Team Experience66

Software Design Document

February 6 , 2026

Team Members:

Ethan Meyer, Manjot Kaur, Moraa Morara, Nasya Valenzuela

Client:

Mike Talyor, Mark Manone

Mentor:

Scott LaRocca

Overview:

This document describes the software design of the Experience66 Android application, including its architecture, component structure, and implementation strategy. It translates project requirements and constraints into a concrete technical design that explains component responsibilities, interactions, and key architectural decisions, and serves as a reference for implementation, evaluation, and future maintenance.

Table of Contents

Introduction....3
Implementation Overview....4
Architectural Overview....5
    As-Built Architecture Overview....6
Component-Level Design....7
    Presentation Module....8
    Application Logic Module....9
    Data Management Module....11
    Concerns....13
Implementation Plan....14
    Implementation Overview and Phasing Strategy....14
    Design-Centric Implementation Timeline....15
    Technical Risks, Trade-offs, and Mitigation Strategies....16
Conclusion....18

Introduction

Experience66 is a mobile Android application designed to support exploration and historical engagement along Route 66 through location-aware points of interest (POIs), archival content, and proximity-based notifications. This project addresses the needs of travelers, historians, and casual users who often traverse regions with unreliable connectivity and limited access to contextual historical information while on the road. The core objective of Experience66 is to deliver a reliable, map-centered exploration experience that functions effectively without continuous network access. To achieve this, the system emphasizes offline-first operation, local data persistence, and on-device execution, ensuring usability across remote environments commonly encountered along Route 66. The application must present POIs on an interactive map, surface associated archival records, and notify users when they approach significant locations, all while maintaining responsiveness and clarity in constrained mobile conditions.

From a design perspective, the system is constrained by several environmental and technical factors: execution on Android mobile devices, reliance on third-party SDKs for mapping and geofencing, academic project timelines, and the absence of a dedicated backend service. These constraints directly shape architectural decisions related to data storage, execution model, and dependency management.This Software Design Document presents the architectural structure and component-level design used to meet these requirements. It establishes the context needed to understand the system's design choices, explains how responsibilities are divided across components, and provides a clear foundation for implementation, evaluation, and future extension.

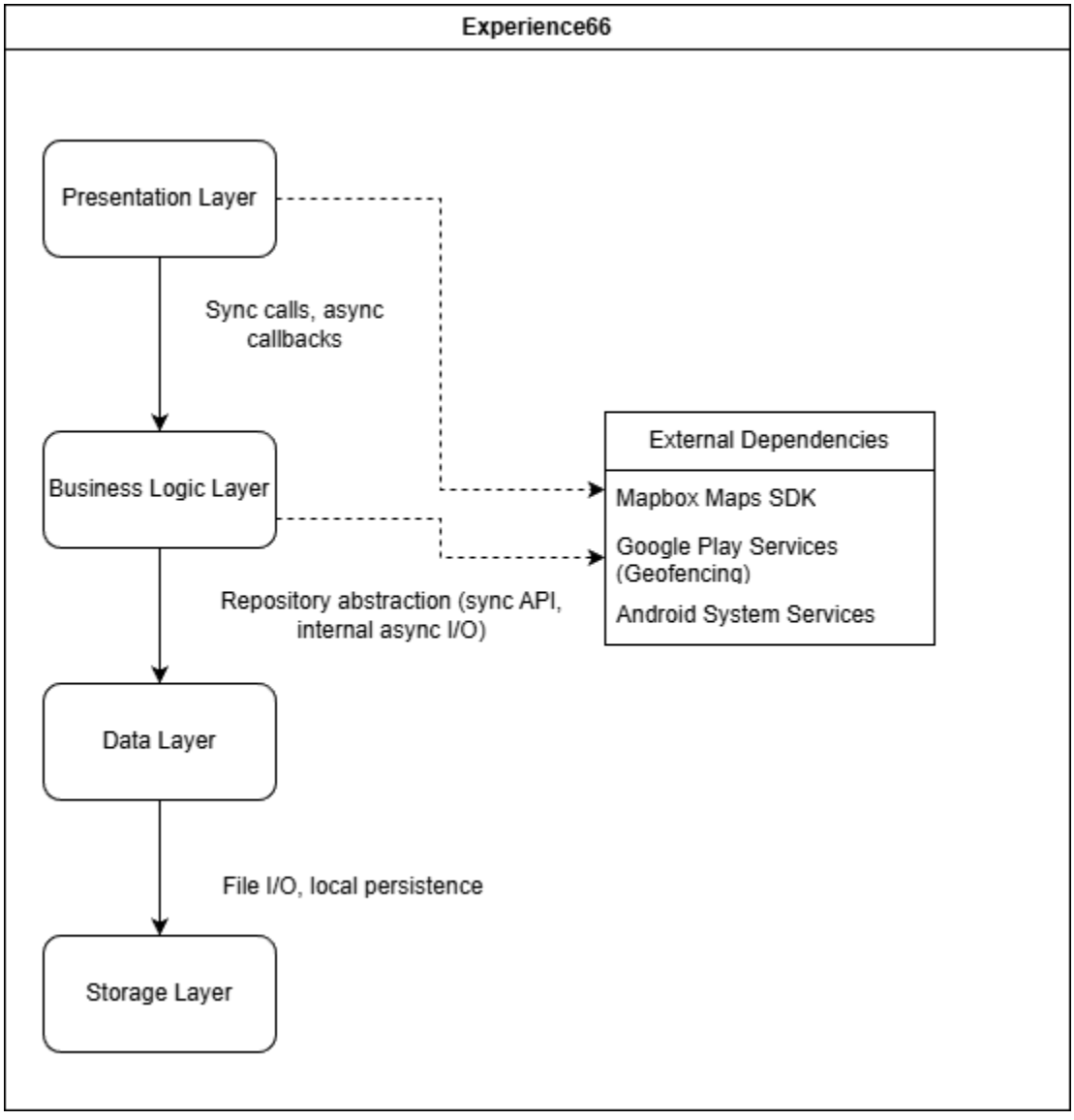
Implementation Overview

Experience66 is implemented as a fully client-side Android application that executes entirely on the user's device. The system is designed to operate without reliance on backend services, reflecting both project constraints and the practical realities of Route 66 travel environments. This design choice prioritizes reliability, offline usability, and predictable performance over centralized data management or real-time synchronization. The solution vision centers on a map-driven mobile experience that integrates static and refreshable historical datasets with location awareness. Users interact primarily through an interactive map interface that displays POIs, supports search and filtering, and responds to geofence triggers as the user moves through physical space. All application behavior is orchestrated locally using Android platform services and SDK-provided APIs. The system adopts a layered architectural approach, separating concerns across presentation, application logic, and data management. The Presentation Layer handles all user-facing behavior, while the Application Logic Layer coordinates workflows such as geofence management, offline decision-making, and UI state updates. The Data Management Layer abstracts file access, parsing, validation, and caching of local datasets. This structure allows design decisions related to platform services, data handling, and user interaction to evolve independently. Key technologies shaping the implementation include the Android SDK, Mapbox Maps SDK for map rendering, and Google Play Services for geofencing. All long-running operations, such as file parsing, dataset refresh, and cache updates, are performed asynchronously to preserve UI responsiveness. The resulting design provides a clear big-picture view of how system components collaborate while remaining grounded in realistic constraints of mobile execution, project scope, and academic timelines.

Architectural Overview

The Experience66 system architecture reflects the project's core constraints: offline-first operation, Android-only deployment, integration with third-party mapping and geofencing SDKs, and the absence of a dedicated backend service. This section explains not only the structural organization of the system, but also the reasoning behind key architectural decisions, the technologies integrated into each layer, and the trade-offs that shaped the final design.

High-Level Architecture of the Experience66 Android Application.



A layered architecture showing Presentation, Business Logic, Data, and Storage layers; arrows indicate synchronous calls, asynchronous callbacks, repository abstraction, and local persistence. External dependencies are integrated via SDK APIs, callbacks, and system services.

As-Built Architecture Overview

The Experience66 Android application follows a **layered, client-side architecture** designed to support offline-first operation while maintaining clear separation of responsibilities. The structure shown in Figure X reflects how the system components interact in practice and how external SDK dependencies are integrated in a controlled manner.

At a high level, the system is organized into four primary layers:

- **Presentation Layer**
- **Business Logic Layer**
- **Data Layer**
- **Storage Layer**

External technologies—including **Mapbox Maps SDK**, **Google Play Services (Geofencing)**, and core **Android System Services**—are integrated through clearly defined boundaries rather than embedded across layers. This design ensures that third-party dependencies remain isolated and manageable.

The **Presentation Layer** is responsible for all user-facing behavior. It renders the interactive map, displays POI markers and detail cards, handles user input, and reflects system state such as offline mode or geofence triggers.

This layer integrates directly with the **Mapbox Maps SDK** to render maps and visual overlays. Map interactions, marker placement, and camera updates are handled here.

Communication with the Business Logic Layer occurs through **synchronous calls**, while long-running operations (such as data loading or geofence callbacks) return through **asynchronous callbacks**, as shown in the diagram.

The Presentation Layer does not perform data parsing, caching decisions, or geofence lifecycle management. Instead, it delegates those responsibilities downward, ensuring UI code remains focused purely on interaction and rendering.

Why this separation matters:

Keeping SDK-driven UI logic here prevents geofencing or storage changes from impacting the visual layer. It also keeps the interface responsive by avoiding direct file I/O or heavy processing in the UI thread.

The **Business Logic Layer** acts as the coordination and decision-making core of the system. It manages workflows that span UI events, data access, and platform services.

This layer integrates with:

- **Google Play Services (Geofencing API)**
- **Android System Services (Location services, broadcast receivers, permissions handling)**

Its responsibilities include:

- Managing geofence registration and lifecycle

- Handling geofence trigger events
- Enforcing offline-first behavior
- Coordinating refresh policies
- Translating structured data into UI-ready state updates

As shown in the diagram, this layer interacts with the Data Layer through a **repository abstraction**, exposing a clean, synchronous API to upper layers while allowing internal operations (such as parsing and file access) to execute asynchronously.

Why Google Play Services was used:

Using the system-managed Geofencing API reduces battery usage and avoids continuous location polling. The trade-off is dependency on Google Play Services and platform constraints such as geofence limits, but the reliability and efficiency benefits outweigh these limitations for a travel-focused app.

The **Data Layer** is responsible for structured data access and transformation. It implements a repository abstraction that shields upper layers from file formats and storage details.

Its responsibilities include:

- Loading bundled POI datasets
- Parsing CSV-based archival records
- Validating and structuring data
- Supporting search and filtering
- Managing refresh logic

As reflected in the diagram, this layer exposes a **synchronous repository API**, while internally performing **asynchronous I/O operations** to prevent blocking higher layers.

This separation ensures that changes in file structure, dataset format, or refresh behavior do not impact UI logic or geofencing workflows.

The **Storage Layer** represents local persistence and file-level storage. It includes:

- Bundled CSV datasets
- Locally cached refresh data
- Lightweight key–value metadata storage
- In-memory runtime collections

All file I/O and local persistence operations occur here. This layer is intentionally isolated from application workflows to keep storage concerns independent.

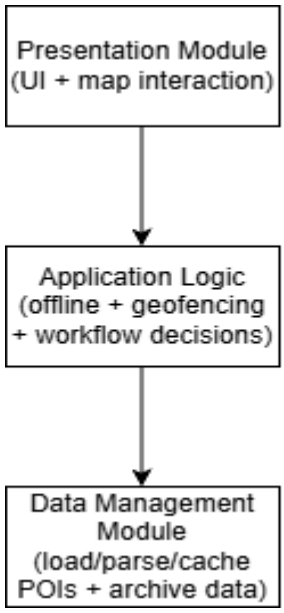
The offline-first design—where all critical content is available locally—directly supports the project goal of delivering Route 66 resources at the point of need during travel, even in areas with limited connectivity.

Trade-off:

A local-only storage model limits scalability and centralized analytics. However, it significantly increases reliability and simplifies deployment, which aligns with the project's scope and academic timeline.

Component-Level Design

The previous sections describe the system at a high level, explaining the layered architecture and the reasoning behind major design decisions. The following section moves from that broad view to a more concrete breakdown of how the application is structured internally. By defining three clear modules and their boundaries, the component-level design shows how responsibilities are separated in practice and how each part of the system interacts without creating unnecessary coupling.

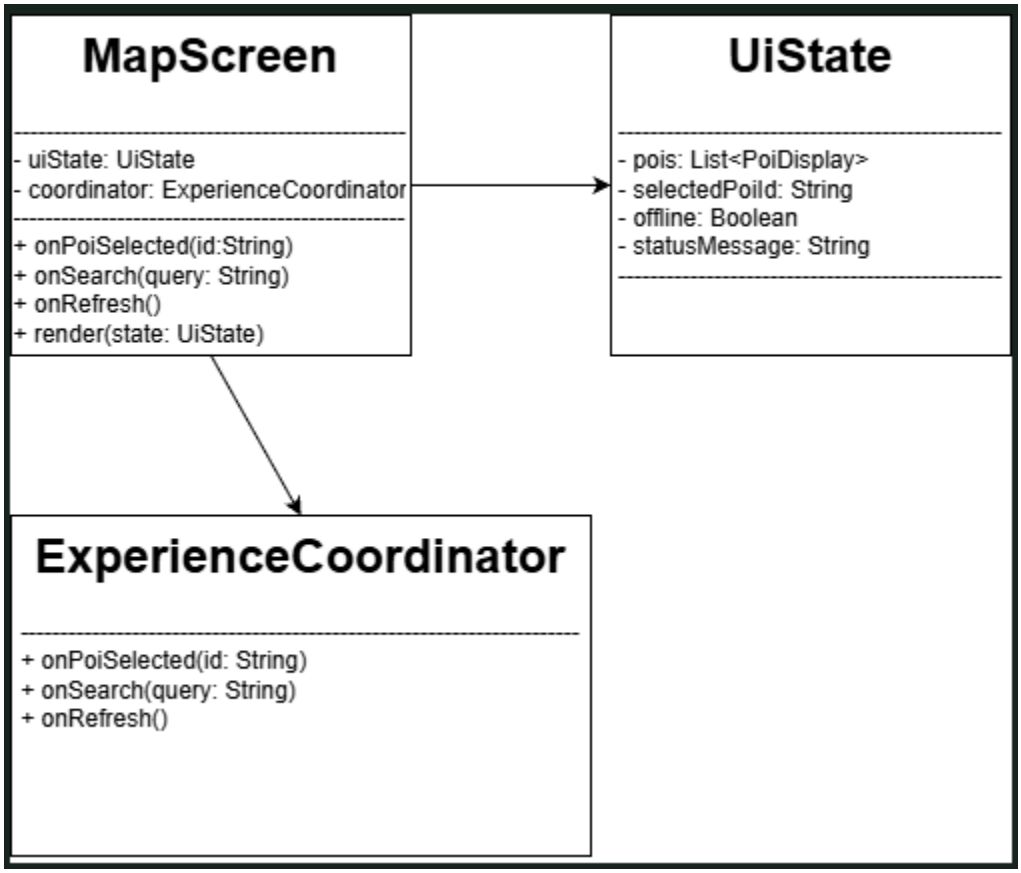


This diagram illustrates the three primary modules of the Experience66 Android application and the allowed dependency flow between them, with user-facing behavior delegated downward through application logic to data management.

The application is organized into three primary modules aligned with the layered architecture described earlier: a Presentation Module, an Application Logic Module, and a Data Management Module. Each module owns a distinct set of responsibilities and interacts with the

others through defined interfaces so that UI concerns, platform workflow concerns, and data concerns do not overlap.

Presentation Module



The Presentation Module is responsible for all user-facing behavior and visual interaction. It manages the map interface, handles user input, and updates the UI in response to system events such as geofence triggers or data availability changes. This module is the coordination point between the user and the rest of the system. It translates user actions into application-level requests and displays results returned by other modules.

This module does not perform data access, offline decision-making, or platform service management directly. Any behavior that involves caching rules, file parsing, or geofence lifecycle decisions is delegated to the Application Logic Module.

Responsibilities

- Render the interactive map and POI markers
- Handle user interactions such as POI selection and search input
- Display system status such as offline mode, refresh progress, and error messages
- Respond to geofence triggers by surfacing alerts or UI updates
- Apply accessibility and visual state changes

Boundaries

- Delegates all non-UI work to the Application Logic Module
- Receives processed data and UI-ready state updates without knowledge of storage or parsing details

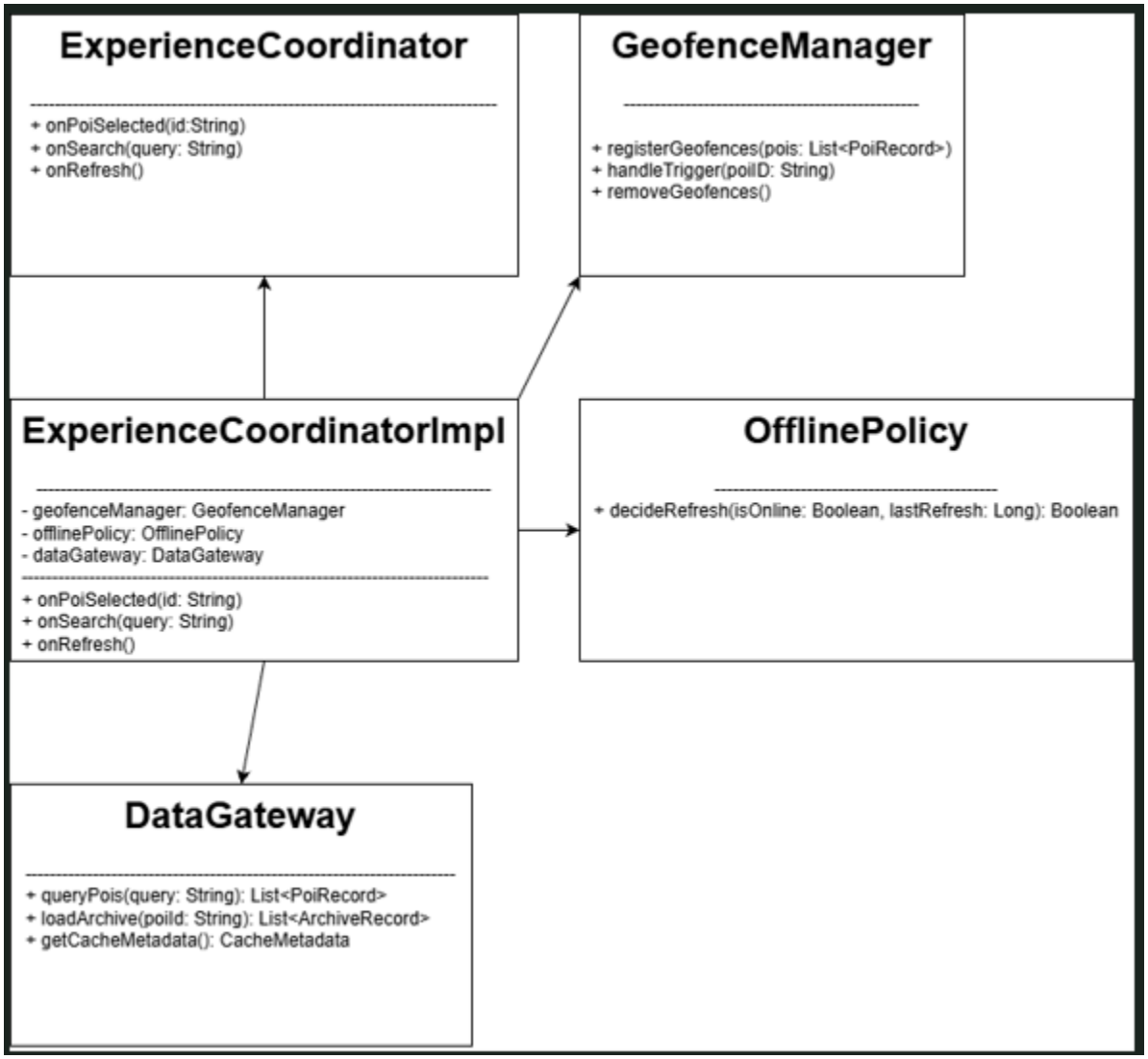
Public Interface

The Presentation Module exposes user intent and consumes UI-ready state.

- **Outputs to other modules**
    - *User intent events:* POI selected, search requested, refresh requested, filter changed
    - *Permission and settings outcomes:* location permission granted or denied, accessibility setting changes

- **Inputs it expects to receive**
    - *Display-ready POI results:* lists of POIs with coordinates and map-ready metadata
    - *Archive/record results:* content associated with a POI for display in the UI
    - *Status updates:* offline or online state, refresh success or failure, geofence trigger notifications

Application Logic Module



The Application Logic Module coordinates core system behavior that is independent of the user interface and storage mechanisms. It serves as the decision-making layer that enforces offline-first behavior, manages geofencing workflows, and orchestrates data requests. This module isolates platform-specific services and policies from both the UI and data layers so that changes to platform behavior do not force rewrites across the entire application.

This module is the owner of application-level workflows such as “user searches for a location, show results, update map markers, and maintain geofence coverage” and “user is offline, uses cached data and communicates limitations clearly.”

Responsibilities

- Coordinate geofence registration, update, and removal behavior
- Handle geofence events and translate them into application-level notifications or UI updates
- Enforce offline and caching policies, including when refresh operations are allowed
- Orchestrate workflows that span UI, data access, and platform services

Boundaries

- Receives user-driven requests from the Presentation Module
- Requests structured data from the Data Management Module without assuming how data is stored or parsed

- Interfaces with external SDKs and system callbacks while shielding other modules from platform details

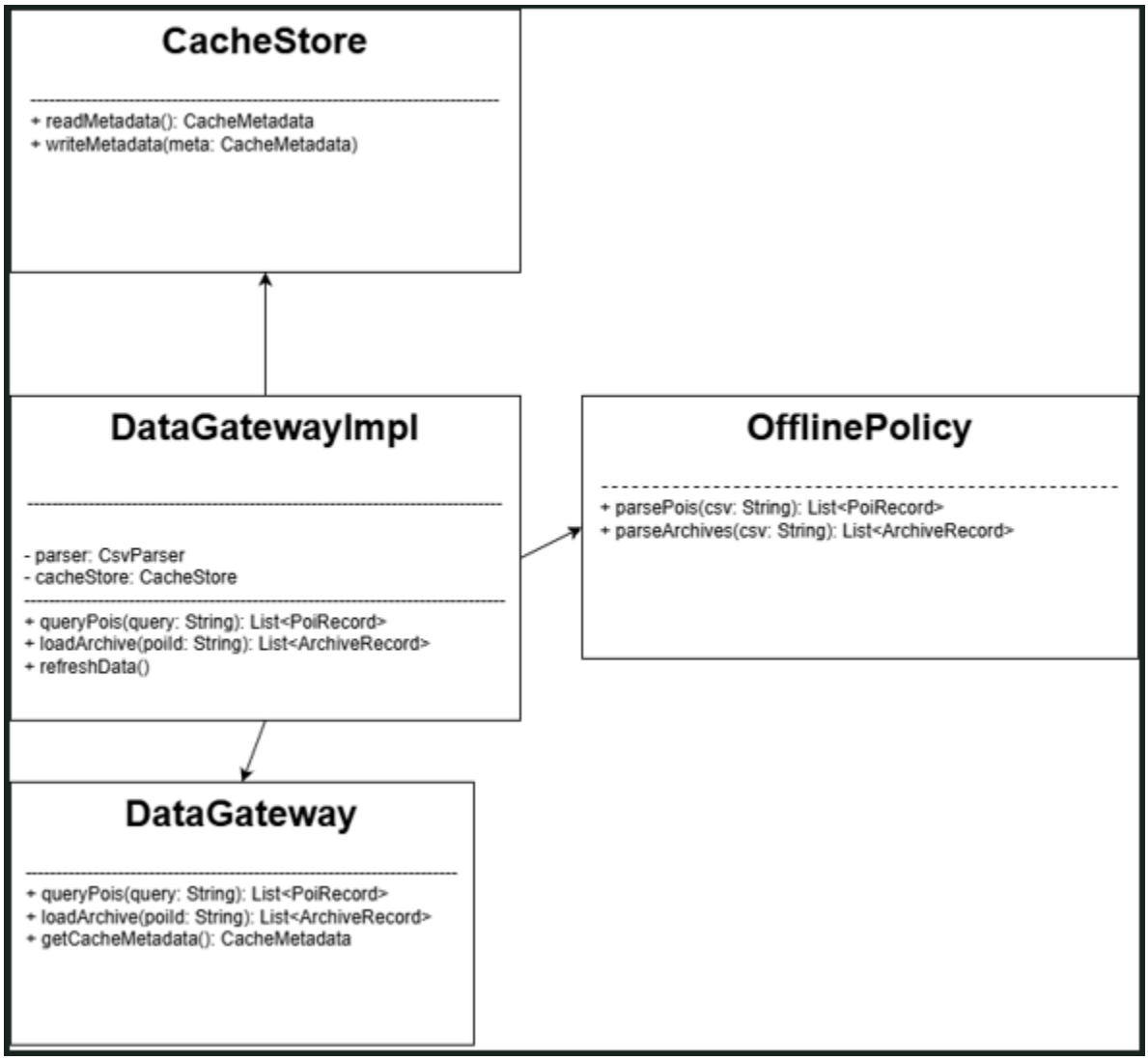
Public Interface

The Application Logic Module exposes workflow-oriented services and consumes structured data.

- **Services it provides**
    - *Geofence control service:* accepts POI location definitions and manages geofence lifecycle
        - Inputs: POI locations (coordinates and identifiers), geofence rules (radius and limits), permission state
        - Outputs: registration success or failure, geofence trigger events, and actionable status messages
    - *Offline behavior and refresh coordination service:* decides whether the app should load cached data or perform refresh operations
        - Inputs: connectivity state, cache metadata (such as last refresh timestamp), user refresh requests
        - Outputs: “use cached data” vs “refresh allowed” decisions, progress or outcome status for the UI
    - *UI state coordination service:* converts raw data into map and UI state changes without embedding UI rendering details
        - Inputs: POI lists, search criteria, geofence triggers
        - Outputs: marker update instructions, selection state, and user-visible notifications

- **Data it expects to receive**
    - Structured POI data suitable for mapping and filtering
    - Archive or record data associated with POIs
    - Cache and freshness metadata needed to make offline-first decisions

Data Management Module



The Data Management Module owns all data access, parsing, validation, and caching behavior. It abstracts the underlying data sources, including bundled CSV datasets and locally cached

refreshed content, and exposes a consistent interface upward. This module ensures data handling remains isolated from UI logic and workflow logic, supporting maintainability and reliable offline operation.

Parsing and file I/O are executed off the UI thread to avoid blocking interaction. The module presents higher layers with validated, structured objects so that the rest of the system does not need to understand file formats, parsing rules, or storage mechanics.

Responsibilities

- Load and parse POI and archival datasets from bundled CSV sources and local cached sources
- Validate and structure data for application use
- Provide search and filtering support over local datasets
- Maintain lightweight persistence for cache metadata and refresh state
- Support incremental refresh and local storage updates when network access is available

Boundaries

- Does not manage geofences, UI state, or offline policy decisions
- Provides structured data through stable interfaces while hiding file formats and parsing details
- Performs parsing and I/O asynchronously to support responsiveness

Public Interface

The Data Management Module exposes data-focused services and returns structured results.

- **Services it provides**
    - *POI query service:* returns POIs for mapping, browsing, and filtering
        - Inputs: optional search text, category filters, proximity constraints, or “all POIs”
        - Outputs: validated POI records with coordinates, identifiers, and display metadata
    - *Archive lookup service:* returns archive or CONTENTdm-associated content tied to a POI
        - Inputs: POI identifier or record key
        - Outputs: archive record list or metadata summaries suitable for UI display
    - *Refresh and cache update service:* updates local datasets and reports cache state
        - Inputs: refresh request, connectivity availability, cache policy constraints
        - Outputs: updated dataset availability, last refreshed metadata, and error conditions if refresh fails

Concerns

Several concerns affect all three modules and are handled consistently across the system. Error handling and logging are centralized so failures in parsing, SDK calls, or refresh operations can be diagnosed without degrading the core offline experience. Performance is protected through background execution for file I/O and parsing, with UI updates occurring only after data is prepared. External services such as Mapbox and geofencing are accessed only through defined boundaries to limit coupling and reduce the risk of SDK-driven design changes impacting the entire codebase.

Implementation Plan

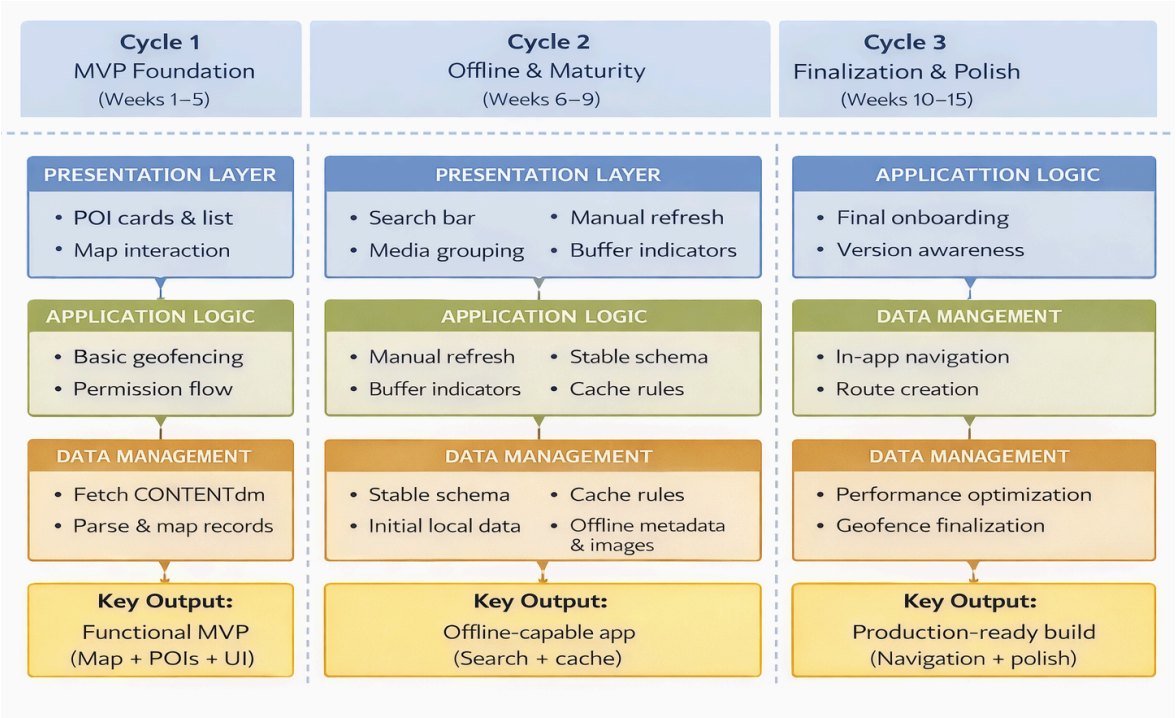
This implementation plan describes how the proposed system architecture will be realized in practice through a phased, incremental development approach. The plan is intentionally design-centric, ensuring that architectural decisions described in the Software Design Document are directly reflected in execution order, integration strategy, and delivery milestones. Development follows a local-first Android architecture with a hybrid data model, emphasizing offline capability, modular integration, and progressive feature expansion.

Implementation Overview and Phasing Strategy

Development is organized into three major cycles aligned with academic milestones. Each cycle delivers a stable, testable increment of the system, reducing integration risk while enabling parallel work across team members. Core architectural components, including the UI layer, local database and cache, geofencing logic, and Mapbox integration, are introduced early and expanded iteratively.

The system executes entirely on-device, with no backend dependency. CONTENTdm data is fetched, parsed, and stored locally using a hybrid model: seed datasets are bundled with the application, while live data can be refreshed and cached when network access is available. Navigation functionality is intentionally deferred to the final cycle to reduce early complexity and dependency risk

Design-Centric Implementation Timeline



Cycle 1 - Core Functionality & MVP Foundation

Focus: Visible functionality, core data flow, and baseline usability

Planned features for this cycle include mapped Route 66 POIs, a clean and functional UI (POI cards and list), simple geofencing, map markers, and basic navigation via Google Maps deep linking. Database work focuses on fetching and parsing CONTENTdm records into a usable local structure. Platform-level concerns such as permissions, error handling, and minimal onboarding are addressed early to ensure stability and usability. Items already implemented (simple geofencing, map markers, and Google Maps navigation) are validated and integrated rather than rebuilt.

Cycle 2 - Offline Capability & Data Maturity

Focus: Reliability without connectivity and improved data organization

This phase introduces offline caching for metadata and images, offline map region downloads, and a stabilized local database schema with validated fields. UI enhancements include search functionality and media grouping for POIs. Manual content refresh and buffer indicators are added to give users control and transparency over data freshness. Testing becomes a shared responsibility across the team to validate offline scenarios and integration points.

Cycle 3 - Feature Completion & Optimization

Focus: Polishing, accessibility, and deferred advanced features

The final cycle integrates deferred functionality, most notably in-app navigation and route creation. Accessibility features are implemented to improve inclusivity, while performance optimizations target map rendering, cache growth, and memory usage. Final onboarding flows and version awareness are added to support long-term maintainability and user experience. Geofencing logic is finalized and tuned based on testing feedback.

Technical Risks, Trade-offs, and Mitigation Strategies

Offline Data Complexity: Caching large datasets and images increases storage and memory pressure. This risk is mitigated through selective caching rules, metadata-first loading, and deferred image downloads.

Mapbox Dependency Risk: Offline map downloads and rendering performance depend on third-party APIs. The design limits exposure by scoping offline regions and deferring advanced navigation until the final cycle.

Deferred Navigation Feature: In-app navigation is postponed to reduce early integration complexity. The trade-off is limited routing capability in early demos, mitigated by Google Maps deep linking in earlier cycles.

Technical Debt from Hybrid Data Sources: Supporting both seed data and live CONTENTdm refresh introduces schema complexity. This is mitigated through early schema stabilization and explicit validation rules in Cycle 2.

Scalability Constraints: A local-only architecture limits dataset size and update frequency. This trade-off is accepted for reliability and offline access, with future backend integration identified as a post-release enhancement.

Conclusion

The design presented in this document provides a clear, cohesive foundation for the Experience66 Android application. By adopting an offline-first, layered architecture and enforcing explicit ownership of responsibilities across components, the system balances usability, maintainability, and technical feasibility within the project's constraints.Key architectural decisions such as local-only execution, repository-based data access, and isolation of platform services directly support the application's core goals of reliability and clarity in environments with limited connectivity. Trade-offs related to scalability, backend omission, and deferred navigation features were intentionally accepted and mitigated through careful modular design and future-ready abstractions.Overall, the proposed design enables efficient implementation, supports incremental development, and minimizes integration risk. It establishes a structure that is both sufficient for current project requirements and adaptable for future enhancement. This Software Design Document therefore serves not only as a description of the system, but as a practical guide for execution, evaluation, and long-term evolution of Experience66.