Team Experience66

Software Testing Plan

March 20, 2026

Team Members:

Ethan Meyer, Manjot Kaur, Moraa Morara, Nasya Valenzuela

Client:

Mike Talyor, Mark Manone

Mentor:

Scott LaRocca

Overview:

The purpose of this document is to outline how the Experience66 application will be tested to ensure it works correctly and meets user needs. It defines the testing approaches used to identify issues and improve overall system quality.

Table of Contents

Introduction

The Experience66 application is an Android mobile app designed to help travelers explore historic Route 66 locations through an interactive, map-based interface. It provides users with curated points of interest (POIs), including descriptions, images, and location-based features such as distance tracking and geofencing. The primary users include tourists and road trip travelers with varying technical experience, often interacting with the app in real-world environments. Therefore, the key quality goals are reliability, accuracy of location-based data, and strong usability, ensuring users can access information easily and without errors. Software testing is essential to ensure these goals are met. The system includes several core components: the Android frontend (UI and ViewModels), a repository and data layer, a structured POI dataset, Mapbox integration, and location services. External dependencies include the Mapbox SDK and Android location services, while testing is conducted using emulators and physical devices with both real and simulated location data. The test scope covers all core functionality, including data processing, map interaction, search and filtering, POI display, and location-based features. Out of scope are low-level Android framework behavior, third-party SDK internals, and large-scale production environments. This scope aligns with the project backlog and demo flight plan, focusing on validating features required for Alpha demos and final acceptance. The testing strategy combines multiple levels of testing throughout development. Unit testing validates core logic such as distance calculations, data parsing, and filtering, and is performed continuously during development. Integration testing ensures that system components work together correctly, especially interactions between the frontend, dataset, map layer, and location services, and is conducted before major milestones such as Alpha I, Alpha II, and the Acceptance Demo. Usability testing evaluates how easily users can navigate and interact with the app, with sessions planned after key development phases to refine the user experience. While performance and security testing are not primary focuses, basic stability and safe data handling are monitored during testing. This strategy is driven by the application's reliance on accurate, real-time user interaction. Features such as distance calculations, map navigation, and geofencing require more intensive testing due to their complexity and high impact on user experience. Simpler components require less focus. By prioritizing high-risk areas, the team ensures efficient testing while maintaining overall quality. The following sections provide a detailed breakdown of each testing approach used in the Experience66 project

Unit Testing

Unit testing for the Experience66 application focuses on verifying the correctness of core logic at the smallest level before it is integrated into the full system. In our project, this means testing individual Kotlin classes, helper functions, and data processing components that power features like distance tracking, POI filtering, and geofencing behavior. Since the app relies heavily on location data and structured datasets, ensuring that these computations are accurate and stable is critical to the overall reliability of the application. By validating these units early and consistently, we reduce the risk of hidden bugs carrying into later stages of development where they are harder to diagnose.

The primary goal of unit testing in this project is to confirm that key logic behaves correctly under a wide range of conditions. This includes verifying that calculations such as distance between coordinates are accurate, that search and filtering logic returns expected results, and that data retrieved from the POI dataset is handled correctly. We also focus on input validation to ensure that invalid or unexpected data does not cause crashes or incorrect outputs. Because Experience66 is designed for real world travel scenarios, even small inaccuracies in logic could negatively affect the user experience, so precision and consistency are important.

To implement unit testing, we use JUnit as the primary testing framework for Kotlin. This allows us to write structured test cases and assertions for each unit of code. In addition, we use Mockito where necessary to mock dependencies, especially for components that rely on Android services such as location providers. This helps isolate logic from external systems so that each unit can be tested independently. Tests are executed locally during development through Android Studio, and they are also designed to be compatible with automated execution if continuous integration is expanded later.

We track test results primarily through pass and fail status, ensuring that all critical logic passes before new features are merged. While full test coverage is not required for every line of code, we prioritize coverage for logic-heavy components that directly impact user functionality. This includes distance calculations, filtering systems, and data parsing. The focus is on meaningful coverage rather than achieving a high percentage for its own sake. In other words, we test what matters most for system correctness.

Units Under Test

The following components have been identified as the most important units for testing based on their complexity and impact on the user experience:

Distance Calculation Logic

This is one of the most critical pieces of the application. It determines how far a user is

from a selected point of interest based on latitude and longitude values. Since this directly affects user decision making while traveling, accuracy is essential.

POI Data Parsing and Model Handling

The application reads structured POI data that includes names, descriptions, coordinates, and media references. Unit tests verify that this data is correctly parsed into usable objects and that missing or malformed fields are handled safely.

Search and Filtering Logic

The search feature allows users to find specific Route 66 locations. This involves filtering the dataset based on user input. Testing ensures that results are accurate, relevant, and consistent across different inputs.

Geofencing Trigger Logic (Testing Mode)

Although geofencing is simulated during development, the logic that determines whether a user has entered a POI region must be validated. This includes checking boundary conditions and ensuring correct triggering behavior.

Repository and Data Access Layer

The repository layer acts as an intermediary between the dataset and the rest of the application. Unit tests ensure that data retrieval is consistent and that the correct data is returned for different queries.

ViewModel Logic (Non UI State Handling)

While UI rendering itself is not unit tested, the logic inside ViewModels that prepares data for display is tested. This includes state updates, transformations, and handling of user interactions at a logical level.

We intentionally do not unit test certain parts of the system. UI rendering, Mapbox SDK interactions, and Android framework components are excluded because they depend heavily on external systems and are better suited for integration or instrumentation testing. Similarly, simple data holders or trivial getters and setters are not prioritized since they do not contain meaningful logic.

Unit Test Design Approach

Our approach to unit testing is structured around systematically exploring how each unit behaves under different conditions. Rather than only testing expected use cases, we include boundary and invalid inputs to ensure robustness. Each unit is evaluated across three main categories of test cases: valid inputs, boundary conditions, and invalid inputs.

1. **Valid Inputs**
   These tests confirm that the unit behaves correctly under normal conditions. They represent realistic scenarios that users are likely to encounter.
2. **Boundary Conditions**
   These tests explore edge cases such as minimum and maximum values. For example, zero distance, extremely large coordinate differences, or empty datasets. These cases are important because they often reveal hidden issues in logic.
3. **Invalid Inputs**
   These tests verify that the system handles incorrect or unexpected inputs gracefully. Instead of crashing, the application should return safe defaults or clear error states.

Below are examples of how this approach is applied to key units in the system.

Unit Under Test: calculateDistance(userLat, userLng, poiLat, poiLng)

Purpose:

- Computes the distance between the user's current location and a selected POI using geographic coordinates.

Test Case Categories:

- Valid inputs: typical coordinates within reasonable distance
- Boundary cases: identical coordinates, extremely large distances
- Invalid inputs: null values, out of range latitude or longitude

Sample Tests:

- User and POI at different locations return a positive distance value
- User and POI at the same coordinates return a distance of 0
- Invalid coordinate values are handled without crashing

Unit Under Test: filterPOIs(searchQuery, poiList)

Purpose:

- Filters a list of POIs based on a user's search input.

Test Case Categories:

- Valid inputs: matching keywords, partial matches
- Boundary cases: empty search query, full dataset match

- Invalid inputs: null query, null dataset

Sample Tests:

- Search for a known POI name returns the correct result
- Empty query returns the full list of POIs
- Null dataset does not cause a crash and returns an empty result

Unit Under Test: parsePOIData(jsonInput)

Purpose:

- Converts raw dataset input into structured POI objects.

Test Case Categories:

- Valid inputs: correctly formatted dataset
- Boundary cases: missing optional fields
- Invalid inputs: malformed JSON, missing required fields

Sample Tests:

- Valid dataset produces a correctly structured list of POIs
- Missing image field results in a default placeholder value
- Malformed input is handled without crashing the application

Unit Under Test: checkGeofenceTrigger(userLocation, poiRegion)

Purpose:

- Determines whether the user has entered a defined POI region.

Test Case Categories:

- Valid inputs: user inside region, user outside region
- Boundary cases: user exactly on the boundary
- Invalid inputs: null location data

Sample Tests:

- User inside the region triggers a positive result

- User outside the region does not trigger
- Null input is handled safely without failure

Overall, unit testing in Experience66 is focused on ensuring that the logic driving the application is accurate, reliable, and resilient. By isolating key components and testing them under a wide range of conditions, we reduce the likelihood of defects appearing in later stages of development. This approach also makes debugging easier since issues can be traced back to specific units rather than complex system interactions.

Integration Testing

Integration testing for Experience66 focuses on verifying that independently developed components work correctly when combined into a complete system. While earlier testing ensured that individual features such as the map, POI dataset, search functionality, and distance calculations worked in isolation, this phase ensures that these components interact properly to support real user workflows.

Following Alpha Demo I and Alpha Demo II, the application now includes multiple connected components, including the Android frontend, Mapbox-based map rendering, the POI dataset, media handling, search functionality, and location-based features. At this stage, issues are more likely to occur at the boundaries between these components rather than within individual features. For example, a POI may exist in the dataset but fail to display correctly on the map, or search results may not properly align with map navigation.

Our approach to integration testing focuses on identifying and validating these critical boundaries. Priority is given to interactions that directly affect the user experience and system stability, such as loading and displaying POIs, selecting locations, searching for content, and calculating distances. These areas are essential because any failure would interrupt the core purpose of the app — helping users explore Route 66 locations in a reliable and intuitive way.

Key Integration Points

The following integration points were identified as critical to system functionality:

- **Frontend (Android UI) ↔ POI Dataset**
  Verifies that POI data (titles, descriptions, coordinates, and images) is correctly retrieved and displayed in the app.

- **Frontend ↔ Mapbox Map Layer**
  Ensures that POIs are accurately represented as markers and that Route 66 is displayed as a highlighted path. It also validates map interactions such as zooming, panning, and marker selection.

- **Map Marker Selection ↔ POI Detail View**
  Confirms that selecting a marker displays the correct POI information, including associated images and descriptions.

- **Search Bar ↔ Filtering Logic ↔ Map Navigation**
  Verifies that search input filters POIs correctly, presents relevant results, and moves the map to the selected location.

- **POI List View ↔ POI Data ↔ Action Buttons**
  Ensures that the list view loads valid POIs and that user actions (such as viewing details or opening a map location) behave as expected.

- **Location Services ↔ Distance Calculation ↔ UI Display**
  Confirms that the app calculates and updates the user's distance to a POI based on real-time location data.

- **Geofencing Logic (Testing Mode) ↔ POI Content Triggering**
  Validates that entering a POI region (in development testing mode) triggers the correct content and media.

Test Environment

Integration testing is conducted in a controlled and repeatable environment using both Android emulators and physical devices. This allowed the team to evaluate app behavior under realistic usage conditions while maintaining consistency across test runs.

A cleaned and structured dataset of Route 66 POIs was used to ensure predictable and verifiable results. This dataset included known locations, descriptions, and image references, making it easier to confirm whether the app displayed accurate information.

For location-based features such as distance tracking and geofencing, mock locations were used within the emulator, along with controlled movement on physical devices. This approach allowed the team to simulate real-world travel scenarios without relying on actual geographic movement.

To maintain consistency, the same test data and scenarios were reused across multiple test runs. This made it easier to identify issues such as incorrect coordinates, broken image links, mismatched search results, or inaccurate distance calculations. Any failures were identified through UI inconsistencies, incorrect data display, or unexpected behavior, and were logged for further analysis.

End-to-End Integration Scenarios

Integration Point: Frontend → Map → POI Dataset

Feature: Display Route 66 points of interest on the map

Scenario Description:
A user opens the app and views Route 66 POIs on the map.

Integration Steps:

- User launches the app

- The app retrieves POI data from the dataset

- The map initializes and displays POI markers

- Route 66 is shown as a highlighted path
- The user interacts with the map

Expected Results:

- The map loads successfully
- POI markers appear at correct locations
- Displayed markers match the dataset
- No crashes or rendering errors occur

Failure Handling:

- Missing data results in a clear fallback state
- Invalid coordinates are skipped and logged
- Map loading issues are captured for debugging

Integration Point: Map Marker → POI Detail View → Media/Data

Feature: Display location details with images

Scenario Description:
A user selects a POI and views its details.

Integration Steps:

- User taps a POI marker
- The app retrieves the associated POI data
- The POI detail card is displayed with image and description
- The user interacts with multiple POIs

Expected Results:

- Correct POI information is displayed

- Images load properly
- No duplicate or invalid entries appear
- The app remains stable during repeated interactions

Failure Handling:

- Missing images display placeholders
- Data mismatches are logged
- The app does not crash due to incomplete data

Integration Point: Search → Filter Logic → Map Navigation

Feature: Search for POIs

Scenario Description:
A user searches for a specific Route 66 location.

Integration Steps:

- User enters a keyword in the search bar
- The app filters POIs based on input
- Relevant results are displayed
- User selects a result
- The map navigates to the selected POI

Expected Results:

- Search input is handled smoothly
- Only relevant results are displayed
- Selected results match the correct map location
- Clearing search restores the full POI set

Failure Handling:

- No-results cases are handled clearly
- Search does not freeze or crash
- Incorrect results are flagged and logged

Integration Point: POI List → POI Data → User Actions

Feature: Browse POIs in list view

Scenario Description:
A user views and interacts with the POI list.

Integration Steps:

- User opens the POI list screen
- The app loads POI data
- The user scrolls through entries
- The user selects an action (e.g., view details)

Expected Results:

- The list loads correctly
- Each POI displays relevant information
- Actions perform as expected
- Navigation remains consistent

Failure Handling:

- Missing data does not break the layout
- Incorrect action behavior is logged
- Navigation errors are identified and tracked

Integration Point: Location Services → Distance Calculation → UI

Feature: Distance tracking

Scenario Description:
A user views the distance to a POI while traveling.

Integration Steps:

- The app retrieves the user's location
- The system calculates distance to a POI
- The distance is displayed
- The value updates as the user moves

Expected Results:

- Distance is calculated accurately
- Units are clear (miles/feet)
- Updates occur in real time
- The app remains stable

Failure Handling:

- Location errors are handled with clear messaging
- Stale data does not produce incorrect values
- Calculation issues are logged

Integration Point: Geofencing (Testing Mode) → Content Trigger

Feature: Trigger POI content based on location

Scenario Description:
A user enters a POI region in a controlled test environment.

Integration Steps:

- A simulated location enters a geofenced area
- The system detects the event

- The app retrieves the corresponding POI
- Content is displayed

Expected Results:

- Correct POI content is triggered
- Content matches the user's location
- No duplicate triggers occur

Failure Handling:

- Missed triggers are logged
- Incorrect matches are flagged
- The app remains stable even if detection fails

With integration testing confirming that all major components of Experience66 work together reliably, the next step is to evaluate how the app performs from a user's perspective. While the system is now functionally complete, usability testing focuses on how easily users can navigate the app, understand its features, and interact with content without confusion.

Since Experience66 is designed for travelers who may be on the move, it is especially important that the app feels intuitive and easy to use. This phase ensures the app is not only working correctly, but also providing a smooth and engaging user experience.

Usability Testing

Overview and Goals

For Experience66, usability testing is critical because the application is designed for travelers who may be on the move, distracted, or unfamiliar with the system. The primary goals of usability testing are to:

- Identify confusing navigation or unclear interface elements
- Validate that geolocation-based content delivery is intuitive
- Ensure that users can easily discover and interact with historical content
- Reduce user errors and friction during key workflows
- Improve overall engagement and satisfaction

Context and Assumptions

The target users for Experience66 include travelers, tourists, and Route 66 enthusiasts with varying levels of technical experience. Many users will interact with the app quickly and in real-world environments where attention may be limited.

Because the app relies on a map-based interface and locally stored content, it is important that users can easily understand how to navigate the map and access information without confusion. Poor usability could lead to frustration or difficulty accessing important content.

Usability Testing Methods

The team will use a combination of lightweight, practical usability testing approaches:

- **Task-Based User Testing:**

  Small usability sessions will be conducted with representative users. Participants will be asked to complete realistic tasks within the app.

- **Informal Client Feedback:**

  Regular feedback sessions with the project client will help validate design decisions and ensure alignment with project goals.

- **Expert Reviews:**

  Team members and mentors will perform heuristic evaluations to identify usability issues based on best practices.

Planned Sessions:

- 3–5 users representing general travelers
- 1–2 team members acting as expert reviewers
- Ongoing client feedback sessions

Sample Tasks:

- Launch the app and view current location on the map
- Discover nearby points of interest
- Select a location and view associated historical content (images, descriptions, etc.)
- Navigate between different locations on the map
- Access content while offline

Measures Collected:

- Task completion success rate
- Time to complete tasks
- Number of errors or confusion points
- User feedback and comments

Integration into Development Cycles

Usability testing will be conducted continuously throughout the development lifecycle:

- **After Alpha I**: Initial usability feedback on basic navigation and map interaction
- **Before Alpha II:** Refinement of workflows and UI improvements based on early feedback
- **Before Final Acceptance:** Validation that all core user flows are intuitive and complete

All usability findings will be documented and categorized based on severity and impact. High-priority usability issues (e.g., blocking navigation or unclear workflows) will be addressed in the next development cycle, while lower-priority improvements will be scheduled as time permits.

Usability testing is treated as an iterative feedback loop, ensuring that the Experience66 application evolves to meet user needs and provides a smooth, engaging experience for travelers along Route 66.

Testing Workflow and Quality Controls

To ensure consistent quality throughout development, the Experience66 team follows a structured workflow for identifying, reporting, and resolving defects. All bugs discovered during unit, integration, or usability testing are documented and tracked using the team's sprint backlog and task board, ensuring visibility across all team members. Additionally, real-time communication and coordination are maintained through Discord, where issues can be quickly discussed, clarified, and assigned.

Each defect report includes a clear title, description of the issue, steps to reproduce, expected vs. actual behavior, associated user story or feature, and any relevant screenshots or logs. This ensures that defects are well-documented and can be efficiently reproduced and resolved.

Defects are prioritized based on severity and impact on the user experience :

- Critical: Causes crashes, major data errors, or blocks core functionality (e.g., map not loading, incorrect POI data). Must be fixed immediately.
- High: Significant feature malfunction affecting usability but not completely blocking use (e.g., incorrect search results). Fixed before the next milestone.
- Medium: Issues that may confuse users but have workarounds (e.g., minor UI inconsistencies).
- Low: Cosmetic issues that do not impact functionality (e.g., styling or layout improvements).

Each defect is linked to a specific user story or use case from the backlog to maintain traceability and ensure all requirements are validated. Bugs must be reproducible before prioritization; unclear issues are discussed through Discord and refined before assignment.

The development cycle follows an iterative process aligned with Alpha demos and the final acceptance milestone. A cycle begins once sprint tasks are defined in the backlog and ends when the following criteria are met:

- All critical and high-severity defects are resolved
- Core features for the milestone are fully functional
- Unit tests for critical logic pass
- Integration testing confirms system components work together correctly
- No crashes or blocking issues occur during normal use

While full test coverage is not required, all essential functionality must be stable. Known issues may be accepted if they are low or medium severity, do not impact core functionality, and are documented in the backlog for future resolution. This workflow, supported by structured backlog tracking and consistent team communication through Discord, ensures that defects are managed efficiently, quality is maintained, and the application is ready for each development milestone and final delivery.

Conclusion

The Experience66 testing plan provides a structured and comprehensive approach to ensuring the application meets its core quality goals of reliability, accuracy, and usability. By combining unit testing, integration testing, and usability testing, the team is able to validate both the technical functionality of the system and the overall user experience. This layered approach allows issues to be identified early, reduces risk, and supports the development of a stable and effective application. Through the use of a sprint backlog for task tracking and Discord for team communication, the team maintains clear organization and efficient collaboration when managing defects and testing progress. Defined workflows for reporting, prioritizing, and resolving issues ensure that critical problems are addressed promptly while allowing for continuous improvement across development cycles. Overall, testing is an ongoing and iterative process that supports each stage of development, from initial builds to final acceptance. By focusing on high-impact features such as location-based services and map interactions, the team ensures that the Experience66 application delivers a reliable and user-friendly experience for travelers exploring Route 66.