GPS Based Campus Room Finder

Sprint 4
11-25-2025

Name	**Email Address**
Aaron Downing	aaron.downing652@topper.wku.edu
Ryerson Brower	ryerson.brower178@topper.wku.edu
Kaden Hunt	kaden.hunt144@topper.wku.edu

Client: Michael Galloway
CS 360
Fall 2025
Project Technical Documentation

Contents

List of Figures

1 Introduction

This document is the final, cumulative technical report for TopperNav — a GPS-based campus room-finder developed during CS 360. It pulls together the requirements, design decisions, code mapping, and test planning produced across Sprints 1–4, and it explains what was implemented, what was deferred, and why.

In short: we built a working Android prototype that supports text search for building/room, a local Room-backed data store for building/room records and recent searches, and a live navigation UI that displays distance, bearing, and an ETA updated at a walking-friendly cadence (target 1–2 Hz). The project emphasizes maintainable abstractions (a 'LocationProvider' interface, a testable 'NavigationViewModel', and reusable 'GeoUtils') so future features can be added without major refactors.

There were deliberate scope reductions made to keep the project achievable in the class timeline. In particular, full indoor turn-by-turn routing, multi-floor map modelling, and a production voice-recognition/guidance system were scoped out and are documented as future work. Those omissions are explicit design choices discussed in the Project Scope and Conclusion sections.

The rest of this document follows the course template: an overview of the system, the UML artifacts and their mapping to code, non-functional requirements and performance plans, testing approach (unit, integration, system, acceptance), and an appendix with build and usage notes.

1.1 Project Overview

The GPS Based Campus Room Finder is a mobile application designed to simplify navigation for WKU students and faculty. The primary purpose of this project is to create a consistent and easy-to-use tool that addresses the common problem of navigating a large and unfamiliar campus environment. Using GPS technology, the application will help users quickly determine their current location and find the most efficient route to any building and room number on campus. This tool will eliminate the need for paper maps and provide an important resource for new and current members of WKU.

The final product will be a user-friendly mobile application that gives real-time guidance and an estimation of travel times. This software will be a valuable tool for the university with potential for expansion to include additional features that continue to enhance the campus experience.

1.2 Project Scope

The project scope for TopperNav defines the total boundary of work required to deliver a maintainable, campus-focused Android navigation application that lets users search buildings and rooms and receive walking guidance (ETA, cardinal direction). It covers analysis, UI design, data modeling, GPS integration, performance measurement, security review, and documentation across four time-boxed sprints.

Inclusions: (1) Acquisition and normalization of building/room CSV data; (2) Kotlin/Compose-based UI screens (Search, History, Navigation, Settings); (3) A location pipeline using Android location services (balanced accuracy) abstracted behind a `LocationProvider` interface; (4) Navigation state management in `NavigationViewModel` including distance, bearing, ETA, and near-destination floor advice; (5) A Room-based local database for queries and search history; (6) Instrumentation and logging plan (NavTick events) for latency and refresh analysis.

Deliverables: Sprint review decks, four incremental technical docs (merged here), final PDF + repository source, functional prototype APK, test artifacts (JUnit tests, planned instrumentation specs), and a traceability matrix linking requirements to implemented features.

Exclusions: Real indoor path-finding, multi-floor routing, voice guidance implementation (stubbed for future), live remote sync, and cross-platform (iOS) support. These are future enhancement candidates documented in the Conclusion.

Team Allocation: Project/coordination (Kaden), data + repository (Ryerson), documentation + QA passes (Aaron). Each member contributed design reviews, code, and validation sessions. Weekly velocity averaged 8–10 hours per member, matching earlier planning assumptions.

Schedule Summary: Sprint 1: requirements initial data. Sprint 2: scope lock architecture sketches. Sprint 3: full UI prototype. Sprint 4: live location, ETA, and test scaffolding + final consolidation. Final submission date: end of term (Week 16). No budget impacts (all tooling free or academic license).

Outcome Expectation: A self-contained, permission-safe Android app demonstrating a coherent design and providing a foundation for future routing enhancements without refactoring core abstractions.
Scope Note: Due to time constraints, several planned features were not implemented: indoor routing, multi-floor support, and voice guidance. The current scope focuses on stable, outdoor navigation between buildings.

1.3 Technical Requirements

1.3.1 Functional Requirements

Mandatory Functional Requirements
The application will determine user location (campus bounds) using device GPS or network providers.
The application will allow searching by building name and room number with partial matching.
The application will generate navigation metrics (distance, bearing, cardinal direction, ETA) to the selected room.
The application will display a live-updating navigation screen (arrow & status string) at 1–2 Hz.
The application will store recent searches locally and allow re-selection from History.
Extended Functional Requirements
The application will allow bookmarking favorites (design placeholder; data schema prepared).
The application will provide optional voice guidance (future; interface reserved).
The application will offer mock-location demo mode for classrooms without GPS signal.

The finalized functional requirements reflect progressive refinement across sprints. Early goals (basic search and display) expanded into a cohesive navigation flow emitting real-time distance and bearing, packaged in a ViewModel to keep UI declarative and testable. Adding History supports rapid re-navigation between successive campus destinations. The live update requirement (1–2 Hz) emerged from usability feedback: slower refresh felt unresponsive; faster provided negligible benefit and increased battery usage. Extended requirements are intentionally scoped for future work—favorites, voice guidance, and a polished mock/demo mode—each supported by existing data or configuration structures so later teams can implement without architecture changes. Together these requirements ensure users can reliably locate rooms, estimate walking time, and re-use past queries, addressing campus orientation challenges for new or transitioning students.

1.3.2 Non-Functional Requirements

Scope Notes: iOS support and multi-user concurrency (100+ users) are future scalable objectives; current implementation is single-device, offline-first. Accuracy target outdoors is $\leq$ 10 m typical; $\leq$ 5 m ideal. Maps rendering replaced by navigation metric computation ($<$ 300 ms typical). Performance/security/UI metrics are sampled via CSV nav tick logging and manual observation.

The cumulative non-functional set guarantees that TopperNav remains dependable, performant, and maintainable. Accuracy (within plus/minus 5 m outdoors) and route metric latency (<= 2 s initial calculation, <= 200 ms UI frame updates) balance user expectations with battery constraints. Offline resilience (local database + cached last known location) enables partial functionality without continuous data. Code ownership and academic integrity are preserved by limiting external code to standard libraries and documented Android APIs. Performance, security, and usability metrics are planned (NavTick logs, basic input validation, permission gating) and can be extended with automated tooling (Jacoco, static analyzers) post-course. Extended goals—battery minimization, graceful degraded-mode indoors, clean UI density—were assessed informally and documented for future quantitative study. The result is a navigation layer that can scale out (cloud sync, multi-user analytics) without rework of core abstractions. Each non-functional requirement now maps to concrete design decisions (provider abstraction, pure math utilities, incremental recompute heuristics), ensuring traceability.

1.4 Target Hardware Details

We will create a mobile app for students and faculty around campus. The target hardware for our mobile app will primarily be for smart phones on Android. The minimum requirements are: The minimum CPU required is a quad-core ARM-based processor (or the processor that's in most smart phones) to ensure real time GPS processing

Mandatory Non-Functional Requirements
The application will provide location updates with an accuracy of at least ±5 meters under clear sky conditions.
The application will deliver route generation results within 2 seconds of the user's search request.
The application will be compatible with either Android or iOS mobile operating systems.
The application will provide visual and text-based route guidance.
The application will support operation in both portrait and landscape orientations without loss of functionality.
All project source code must be developed by the CS 360 project team.
The project must use a database.
Performance metrics should be gathered and optimized.
Security metrics should be gathered and optimized
User interface metrics should be gathered and optimized.
Extended Non-Functional Requirements
The application should maintain functionality with limited or no internet connection
The application should consume minimal battery power while running in the background.
The application should be designed with a clean, intuitive user interface that prioritizes ease of use.
Performance Non-Functional Requirements
The application should load maps and calculate routes within 3 seconds under normal network conditions.
The application should maintain a user interface response delay of no more than 200 ms during normal operation.
The system should handle at least 100 concurrent users without significant degradation in performance.
The database should return query results within 1 second on average.
The application should ensure smooth real-time navigation updates with a refresh rate suitable for walking speed (1–2 F

and navigation. Our test case for the CPU would be to run a continuous navigation to confirm smooth updating with no lag. You would need at least 2 GB of RAM so the product can also run things like GPS tracking at the same time and the rendering of the map. Our test case for the RAM would be to monitor memory usage under a heavy load. A minimum of 200 MB of persistent storage is needed for the application installation, location files, and maps. Our storage test case would be to install the app and see how much it takes up. Network connectivity will be necessary via Wi-Fi or 4G/5G mobile data, with at least 1 Mbps of sustained bandwidth for map updates and routing queries. The targeted output device will be a touchscreen of a smart phone.

We don't have a plan to place this app on PC or computers but it would be something to possible implement in the future. A software we are using called Android studios also has some hardware requirement. These are: A OS of 64-bit Windows 10 or newer, RAM with 16 GB, a CPU with a processor with visualization support (Intel VT-x or AMD-V), micro architecture from after 2017. 16 GB of free disk space, preferably on a Solid State Drive (SSD). A GPU with at least 4 GB of VRAM.

1.5 Software Product Development

The software we are using already are Google doc, TexLive, github, Android Studio, SQL and VScode. Google doc we use to keep up with each others documents and any document we need to print out. For our documentations we are using TexLive to edit both or Organizational and Tech Docs. Github we used to get a repository that is easy to access and easy for us to place our updated docs and code. Git hub also helps use be able to access everything without having to send files back and forth. We already planed on using VScode and the coding language we will use to code our app is JAVA. VS code with the help of github makes it really easy to pull everyone's code when they edit it so once again we are not sending a ton of files and getting them mixed up. One of the more important software we will use is Android Studio. This will allow use to code and Android app easier. This will also let use visualize the app when we don't have a Android phone accessible. For our database to hold all the data for location of rooms and routes we will use SQL.

2 Modeling and Design

2.1 System Boundaries

2.1.1 Physical

The physical system boundaries for the GPS-Based Campus Room finder are limited to mobile devices, primarily Android smart phones used by WKU students and faculty. The application relies on the mobile phones built-in hardware components such as the GPS system, touchscreen interface, and mobile network for accurate navigation. It will not include any external hardware devices not included in the user's smart phone. The system interacts with Google Maps API (or similar) for real-time navigation and requires the campus buildings and room data stored in the project database. Any devices outside of android mobile devices, such as iphones, PCs, and kiosks, lie out of the scope of the project. The application requires minimum resources, 2GB of RAM and 100mb of storage will be enough which is available on most android phones. Security is ensured by relying on the built-in authentication systems on the phone. The application is easily scalable by adding functionality for more android phones and adding a larger database.

2.1.2 Logical

The logical system boundaries for the GPS-Based Campus Room Finder defines the flow of the information and functions managed by the application. Internally, the system handles location detection, room and building searches, and route generation. It manages the retrieval of the campus building and room number data from the project database and uses the GPS coordinates for navigation. Externally, the system communicates with Google Maps API and user-interface to display directions and built-in mobile operating systems for device-level functions such as notifications. Any process beyond navigation, such as class scheduling and campus event times remain outside the logical scope of the project.

2.2 Wireframes and Storyboard

The storyboard begins at the Home/Search screen (input or History re-select), advances to Navigation (live metrics + directional arrow), and optionally transitions to Settings (units, mock mode). Error or empty states (no matches, permission denied) funnel back to Search with clear prompts. This linear but re-entrant flow keeps cognitive load low versus deep menu hierarchies.

2.3 UML

2.3.1 Class Diagrams

[Missing image: images/Facade Design Pattern.drawio.png]

Figure 1: Structural Design Pattern: *Facade* for routing & map services.

[Missing image: images/Singleton Class Diagram.png]

Figure 2: Creational Design Pattern: *Singleton* for AppConfig.

[Missing image: images/ClassDesignPattern.png]

Figure 3: Behavioral Design Pattern: State.

2.3.2 Use Case Diagrams

Actors: Student/Faculty (User). Core use cases: Determine Location, Search Room, Generate Route, Display Estimated Travel Time

[Missing image: images/Use Case Diagrams.png]

Figure 4: Use Case Diagrams.

2.3.3 Use Case Scenarios Developed from Use Case Diagrams (Primary, Secondary)

[Missing image: images/Use Case Scenario.png]

Figure 5: Use Case Scenario Table for the search room use case.

2.3.4 Sequence Diagrams

The following sequence diagram shows the process of the actor (user) getting into the UI and requesting a location. This would then go through the database to get location data. It would then create a route from the users location and the desired room. As the user is moving the route would update in relation to the location of the user.

[Missing image: images/ToppperNav Sequance.drawio.png]

Figure 6: Sequence diagram for the user route request, generation, and update.

2.3.5 State Diagrams

The following state diagram models the process of a user searching for a campus room. The diagram begins when the user enters a room number and submits it. The system then validates the input: if it is invalid, the user is prompted to retry; if valid, the system queries the database. If the room is found, a path is generated and directions are displayed. If the room is not found, or a query fails, the user can correct their input and resubmit. This diagram focuses only on the search and route-display feature in Sprint 2, since implementation has not yet begun.

[State diagram image omitted in PDF build]

Figure 7: State UML diagram for the room search feature, showing user input, validation, query, error handling, and path display.

2.3.6 Component Diagrams

The component model separates the app into five deployable parts: (1) UI Layer, (2) Navigation Engine, (3) GPS Service, (4) Database, and (5) External API. Components communicate via interfaces to keep the UI testable and to allow swapping the routing provider later. In Sprint 3, only the UI Layer is active; the Navigation Engine exposes no-op methods used by the UI so screenshots and flows are demonstrable without live routing.

[Missing image: images/component_diagram.png]

Figure 8: Component diagram.

2.3.7 Deployment Diagrams

The diagram shows the physical side of the system, which is the user device like a smart phone or tablet.Also related to the physical side there is the client UI. Then there is also the virtual that has database, and routing engine that makes the route. The virtual also holds all the logic and back end APIs.

[Missing image: images/ToppperNav Deployment Diagram.drawio.png]

Figure 9: Deployment diagram for mobile + external services.

2.4 Mapping of Source Code to UML Diagrams

Extended Mapping Table:

- **Facade Pattern (planned):** Placeholder for external routing service; current stub implicit in separation of `LocationProvider` and future cloud sync (no concrete file yet—design reserved).
- **Singleton Pattern:** `core/AppConfig.kt` (global configuration + feature toggles).
- **State Pattern:** `viewmodel/NavigationViewModel.kt` (`NavState` data class models navigation lifecycle).
- **Use Case Diagram Mapping:** "Search Room" → `domain/usecase/SearchRoomsUseCase.kt`; "Determine Location" → `data/local/FusedLocationProviderImpl.kt`; "Generate Guidance" → `viewmodel/NavigationViewMo` "Display ETA" → `ui/screens/NavigationScreen.kt`.
- **Use Case Scenarios Table:** Each scenario row (search success, not found) corresponds to logic branches in `SearchRoomsUseCase` (empty result vs non-empty) and UI state in `SearchScreen.kt`.
- **Sequence Diagram:** Method chain: `setDestination()` → `recompute()` emission → UI recomposition (NavigationScreen collects `state`).
- **State Diagram:** Transitions: Input → Validation (string length check) → Query (UseCase) → Path Metrics (ViewModel) → Arrival (distance <= near threshold triggers floor advice).
- **Component Diagram:** Layers map to folders: UI (`ui/screens`), ViewModel (`viewmodel`), UseCases (`domain/usecase`), Data (`data/local`, `data/repository`), Utilities (`util`).
- **Deployment Diagram:** Physical node = Android device (`MainActivity.kt`); virtual nodes: local DB (Room), GPS provider (LocationManager), potential future API—currently absent.

2.5 Version Control

Our team uses GitHub as the central version control platform to manage all project files and source code. Each member maintains a local copy of the repository and synchronizes changes through frequent commits to ensure consistency and prevent merge conflicts. Commits are pushed after completing small, testable units of work, such as app updates, documentation edits, or minor code adjustments. The main branch serves as the stable build for demonstration and submission. Version numbers follow a milestone-based convention, our current build aligns with version 0.5, representing partial functionality with a completed interface. Final integration and full feature implementation will mark version 1.0. GitHub's built-in commit history are used for transparency, collaboration, and rollback support. This process ensures our documentation, source code, and assets remain synchronized throughout each sprint.

2.6 Requirements Traceability Table

The table ties requirements to use cases to verify coverage.

Table 1: Requirements Traceability Table

ID	Requirement	Use Case(s)	Impl.	Source / Notes
FR1	Determine user location using GPS/network providers	Determine Location	Y	data/local/FusedLocationProviderImpl.k data/local/LocationProvider.kt
FR2	Search by building name and room number (partial match)	Search Room	Y	domain/usecase/SearchRoomsUseCase.kt ui/screens/SearchScreen.kt, data/repository/NavigationRepositoryIm
FR3	Generate navigation metrics (distance, bearing, ETA)	Generate Guidance / Display ETA	Y	util/GeoUtils.java, viewmodel/NavigationViewModel.kt
FR4	Live-updating navigation display (arrow & status) at 1–2 Hz	Display ETA / Navigation	Y (1–2 Hz target)	ui/screens/NavigationScreen.kt, viewmodel/NavigationViewModel.kt; NavTick logging instrumented
FR5	Store recent searches locally and allow re-selection	History	Y	data/local/db (Room), data/repository, ui/screens/HistoryScreen.kt
EFR1	Bookmark / favorites	Favorites	Partial	Data schema prepared; feature placeholder in AppConfig / schema
EFR2	Mock-location demo mode for classrooms	Demo / Test Mode	Y (mock support)	core/AppConfig.kt (mock toggles), test scaffolding
NF1	Location accuracy (target ± 5 m outdoors)	Determine Location	Partial	Depends on device/GPS; fused provider implementation (FusedLocationProviderImpl) and permission handling
NF2	Route generation latency $\leq 2s$	Generate Guidance	Planned / Measured	viewmodel/NavigationViewModel.kt; latency instrumentation via NavTick CSVs
NF3	Cross-platform compatibility (Android/iOS)	Platform	Deferred	Current implementation Android-only; iOS planned as future work
NF4	Visual and text-based guidance	Display ETA	Y	ui/screens/NavigationScreen.kt supports visual and textual cues
NF5	Offline capability; use local DB	Storage / Offline	Y	data/local (Room DB), CSV import for building/room data
PERF1	Navigation refresh rate 1–2 Hz	Navigation	Y (target)	viewmodel instrumentation, NavTick logs for verification
SEC1	Input validation / basic database protection	Security	Planned	Input validation in usecases; secure storage via platform sandbox (to be documented / extended)

2.7 Data Dictionary

This data dictionary defines the essential data elements used by the GPS-Based Campus Room Finder and establishes a shared reference for how information is structured within the application. For this sprint, it focuses on the key elements that support the current UI components, buildings, rooms, search history, and user settings. Each entry outlines the entity, field, type, and a description, ensuring consistency between the database design and the user interface. The dictionary acts as a blueprint for further database integration, guiding how information such as building names, room numbers, and user preferences will be stored, retrieved, and displayed. By defining the fields now, the team can align tasks and maintain naming conventions across classes, SQLite tables, and API calls. This structure helps prevent redundancy, supports scalability, and guarantees that all parts of the system use consistent data definitions moving into Sprint 4.

Entity	**Field**	**Type**	**Description**
Building	buildingId (PK)	INTEGER	Unique building identifier.
Building	name	TEXT	Full building name (e.g., “Snell Hall”).
Building	lat	REAL	Latitude coordinate.
Building	lng	REAL	Longitude coordinate.
Room	roomId (PK)	INTEGER	Unique room identifier.
Room	buildingId (FK)	INTEGER	References Building.
Room	roomNumber	TEXT	Room label (e.g., “B104”).
SearchHistory	historyId (PK)	INTEGER	Unique ID for each search entry.
SearchHistory	queryText	TEXT	User-entered query.
SearchHistory	createdAt	INTEGER	Timestamp when search was made.
UserSettings	settingsId (PK)	INTEGER	Always 1; single local config.
UserSettings	units	TEXT	Distance units (“imperial” / “metric”).
UserSettings	theme	TEXT	UI theme (“light” / “dark”).

2.8 User Experience

The GPS-Based Campus Room Finder prioritizes a simple, intuitive, and visually appealing user-interface to ensure users can quickly locate rooms with minimal effort. Upon opening the app, users are greeted with a clean home screen displaying a search bar and quick-access icons for common destinations. The navigation flow is designed to streamline the room searching process for a quick and easy to use app. Users can either type a room name or select from recent searches to instantly view directions.

Interactive map allows users to zoom, rotate, and view detailed paths through campus buildings. Visual cues such as estimated travel time, text box directions, and an arrow pointing to the desired destination allows users to stay oriented. The app uses familiar icons and consistent layouts to maintain ease of use.

Overall, the systems UX focuses on speed, clarity, and easy use. Making it an efficient tool for navigating campus.

3 Non-Functional Product Details

3.1 Product Security

3.1.1 Approach to Security in all Process Steps

For our mobile app and the very specific scope we have our app doesn’t have to many security features we don’t have a login but if we had more time or wanted to continue this project later on it would be something we would implement. But we chose secure practices from the "Secure Coding Practices Checklist" to help us keep everything secure. These are Database security, Input validation, File Management, Memory Management, Error handling/logging, and Output Encoding. These are some of the security features we want to implement into our mobile app. The biggest one that we need is the database security because our app uses a database to store User location data and building/rooms location data. Something that will help us keep our database secure are some of the other options which are input validation and output encoding. The input validation will mainly making

sure that the request for the locations go to the right database and are not sent elsewhere or intercepted by other person. Output encoding would make sure that the results and the route generation from the database would be easily hacked into. File management would really just help us keep more organized but with well organized files this will help us be able to see any codes that aren't ours and that could be harmful. memory management would make sure that we don't have to much memory usage so that it doesn't make users phone slow and cause problems. Error handling will make sure that any errors we get will be immediately be sent to us so that we will be able to fix it on the other side we will have logging which will keep a log of our errors and everything that happens so that we can prevent any errors from occurring.

3.1.2 Security Threat Model

[Missing image: images/securitythreatmodel.png]

Figure 10: Security Threat Model with client–server trust boundary and representative threats.

The security threat model for the GPS-Based Campus Room Finder is derived from the deployment diagram, which includes a planned cloud service for future expansion. However, in the current sprint, all data and logic are stored locally on the user's Android device. The primary trust boundary is between the application's user interface and its local data store rather than over a live network. Current threats include tampering with locally stored building or route data, GPS spoofing, and unauthorized access to cached searches. These are mitigated through Android's built-in sandboxing, secure SQLite storage, and validation of all input fields. Although the cloud cluster in the deployment model represents potential future capabilities (e.g., live map or database sync), it is not active in the Sprint 3 implementation. Future online versions will extend the threat model to include API authentication, HTTPS encryption, and backend access control.

3.1.3 Security Levels

We don't have to many security levels for our project because we are keeping the scope small and for now only on one phone so that its easier. But some of the basic security levels we would have are the User level. This level will get basic entry to our mobile app so that they can go into the app and use it by getting a location. This level will not have access to the database or the code. The next level would be the Administrators or us the people creating the app. This level would give us access to everything from the database to the source code. Something we could implement later could be a third level the is in between called location manager. This would be a level that could add locations to the database for areas that haven't been added. This level will have access to the database and be able to add locations but won't have access to the source code.

3.2 Product Performance

3.2.1 Product Performance Requirements

The GPS Campus Room Finder must perform efficiently usual university network conditions. The application should loud the navigation tool and calculate the navigation routes within 3 seconds, while also maintaining a smooth UI responsiveness with less than 200 ms delay for location updates. It must support at least 100 concurrent users without service degradation. The database should respond within 1 second on average. These requirements ensure reliable, real-time navigation to keep users satisfied. By defining performance metrics, the system guarantees an easily scalable, fast, and reliable experience.

- Navigation load time $\leq$ 3 seconds
- Route generation time $\leq$ 3 seconds
- UI response delay $\leq$ 200 ms
- Database query response $\leq$ 1 second
- Supports $\geq$ 100 concurrent users

3.2.2 Measurable Performance Objectives

The GPS-Based Campus Room Finder must meet specific performance objectives to ensure its smooth and reliable for users. System response times, accuracy, and scalability will be monitored during testing. The application should provide optimal routes and render navigation tools quickly to support real-time navigation. Location tracking must remain accurate when on campus, even with below optimal network latency. These measurable objectives ensure the system is consistent across devices even with sub-par conditions. The following metrics define the measurable performance goals to be achieved during testing and deployment. These metrics are response time, accuracy, interface responsiveness, query speed, and multi-user scalability.

- Route and map generation time ≤ 3 seconds
- Location accuracy within ± 5 meters
- User interface response delay ≤ 200 ms
- Database query response ≤ 1 second
- Support for ≥ 100 concurrent users without degradation
- Navigation refresh rate of 1–2 Hz during movement

3.2.3 Application Workload

Our goal for Sprint 4 is to collect measured workload data that reflects how users actually move through the app rather than relying on assumptions. We will instrument the UI with lightweight timers and event logging to app-private storage. Each log record will capture: timestam, sessionId, eventTyp, screen, durationMs, and (when applicable) queryTextLen and resultCount. Targeted events include: SearchStarted, SearchCompleted, HistoryOpened, HistorySelected, NavShown, NavTick, and SettingsChanged.

Methodology: (1) Simulated sessions on the emulator (baseline), (2) at least 5 physical-device sessions on a mid-range Android phone. We will run three scripted scenarios: New User: first-time open, type full query, Repeat User: use History, and Multi-Stop: two sequential searches. For each scenario we will collect at least 20 runs to compute stable medians values.

Primary workload metrics to visualize: (a) percent of time per screen (Search, Navigation, History), (b) keystrokes-per-successful-search, (c) search-to-result latency, (d) re-navigation latency from History, (e) navigation refresh cadence (target 1–2 Hz) and UI frame time distribution. We will aggregate to CSVs and produce bar charts and box plots for the above.

Acceptance gates tied to requirements: Search ≤ 3 s (median), UI tap responsiveness ≤ 200 ms, navigation refresh 1–2 Hz with no spikes ($<1\%$ frames >16 ms). All logs remain local and can be cleared in Settings (opt-out). Results and graphs will be included in Sprint 4.

3.2.4 Hardware and Software Bottlenecks

During Sprint 4, we will perform lightweight profiling to identify both hardware and software bottlenecks that could impact navigation accuracy, responsiveness, or reliability. On the hardware side, the app depends on three key resources: GPS, CPU, and battery. Continuous location polling may strain mid-range phones, so the refresh rate (1–2 Hz) will be adjusted to balance smooth updates with efficient power use. Performance tests will compare different polling intervals and note CPU usage, memory footprint, and battery drain during 15-minute simulated walks. Storage I/O will also be reviewed since the SQLite database must quickly read and write local search history and building data without fragmentation or delays.

On the software side, potential bottlenecks include inefficient query logic, excessive UI recomposition in Jetpack Compose, and redundant database access during route updates. We will instrument critical functions with timers and Android Profiler traces to isolate slow operations. Caching frequently used building and room data in memory and delaying rapid UI events will reduce redundant work. Expected outcomes include faster search responses ($\leq$ 1 s), stable frame rates, and reduced CPU peaks. Documented metrics will verify that each identified issue is mitigated and that the system remains responsive across devices meeting our minimum specifications.

3.2.5 Synthetic Performance Benchmarks

Since our application runs entirely on mobile hardware, traditional benchmarking tools such as Sysbench are not applicable. Instead, the team will use lightweight synthetic benchmarks developed within Android Studio and the app itself to evaluate the performance of CPU, memory, file I/O, and database operations. These tests will be run on both the Android emulator and at least one physical device that meets the minimum hardware specifications.

The synthetic tests are divided into three major groups:

- CPU Benchmark: Executes 10,000 iterative distance calculations to simulate route computation workload. The benchmark will record total computation time and average time per iteration to evaluate CPU throughput.
- Database Benchmark: Inserts, queries, and deletes 1,000 building and room records within an in-memory SQLite database. Average query latency and insertion rate will be logged to confirm sub-second performance under realistic data volumes.
- File I/O Benchmark: Writes and reads a 1 MB test file from the app's local storage to measure sustained I/O throughput and confirm minimal lag during history logging or cache updates.

All results will be collected through Android's built-in profiler and summarized in Sprint 4 using graphs that visualize average and 95th-percentile latencies. These controlled micro-benchmarks will confirm that the application can maintain consistent responsiveness on target hardware without relying on external servers or heavy instrumentation.

3.2.6 Performance Tests

Scope and rationale Sprint 3 delivered the interactive UI. The following performance tests will be executed on a mid-range Android device in Sprint 4 to satisfy the product performance requirements and measurable objectives defined earlier.

Planned test cases

1) Search latency Goal: median time from query submit to results shown is 3 seconds or less. Method: instrument SearchScreen to log timestamps for SearchStarted and SearchCompleted. Run 20 scripted trials across three scenarios (new user, repeat user via history, multi-stop). Output: box plot of latency per scenario plus 95th percentile.

2) Navigation refresh cadence Goal: navigation updates at 1–2 Hz with no visible jitter. Method: log NavTick events and UI frame times during a 15-minute walk simulation. Compare refresh cadence at 1 Hz and 2 Hz. Output: bar chart of average and 95th percentile frame times.

3) Local database query and write Goal: queries return within 1 second on average. Method: seed 1,000 rooms across multiple buildings; measure average read of room by building and number; measure write rate for recent-search entries. Output: table of average and 95th percentile latencies.

4) File I/O for cache Goal: sustained read/write of a 1 MB cache file without blocking UI events. Method: write, read, and delete a test file while capturing frame timing. Output: line chart of frame time distribution during I/O.

5) Power and CPU profile Goal: maintain responsiveness while minimizing CPU spikes. Method: compare GPS polling at 1 Hz vs 2 Hz using Android Profiler. Output: table of average CPU, peak CPU, and battery drop over 15 minutes.

Deliverables CSV logs checked into the repo, figures included in this document, and a short discussion comparing results against the targets. Any regressions will include a mitigation note and owner.

4 Software Testing

4.1 Software Testing Plan Template

Test Plan Identifier: TP-NAV-FINAL-01 (Unit), TP-NAV-FINAL-02 (Integration), TP-NAV-FINAL-03 (System), TP-NAV-FINAL-04 (Acceptance)

Introduction: Validate correctness of search, live navigation metrics, and stability under typical walking usage with deterministic and real GPS inputs.

Test item: TopperNav APK + module source (UI screens, ViewModels, Repository, GeoUtils, Location provider implementation).

Features to test/not to test: Test: search parsing, distance/bearing math, ETA, History recall, permission flow. Not test: voice guidance (stub), cloud sync (future), favorites (UI hidden).

Approach: Mixed strategy: automated JUnit for pure logic and state transitions; manual device runs for GPS accuracy & latency; planned instrumentation for UI rendering sanity.

Test deliverables: JUnit source, test result reports (HTML/XML), NavTick CSV logs, summarized performance metrics, final test report table.

Item pass/fail criteria: All unit tests pass; navigation ticks maintain 1–2 Hz median; no crashes in 15-minute walk; distance monotonically decreases toward destination barring route deviations.

Environmental needs: Android Studio, Java 17, physical device (Android 11+), emulator (API 34), adb tooling, location permission granted.

Responsibilities: Each member executes defined device sessions; test lead aggregates logs and updates documentation tables.

Staffing and training needs: Brief walkthrough of test execution and interpreting HTML reports; no advanced training required.

Schedule: Unit tests implemented immediately post Sprint 4 coding; device sessions over 3 days; final aggregation on day 4; acceptance sign-off day 5.

Risks and Mitigation: Risk: GPS variance indoors → use emulator GPX + outdoor runs. Risk: battery usage at higher polling → balanced accuracy provider. Risk: limited test time → prioritize high-value metrics first.

Approvals: Project manager and instructor sign-off recorded in repository tag notes.

4.2 Unit Testing

This project defines a small set of high-value unit tests that validate pure logic and ViewModel behaviors. These should be implemented under ‘app/src/test/java/...‘ using JUnit and ‘kotlinx-coroutines-test‘ for coroutine control.

Planned unit tests (minimum):

- GeoUtilsTest (JVM unit test): verify ‘distanceMeters‘ and ‘bearingDegrees‘ for known coordinate pairs (three cases: short distance, antipodal-ish, and same point). Target: numerical accuracy within 0.5 m for short distances.
- SearchRoomsUseCaseTest: use a fake repository to assert search returns expected room/building entities for common queries.
- NavigationViewModelTest: inject a fake ‘LocationProvider‘ (e.g., a ‘MutableSharedFlow<Location>‘). Tests:
  1. When provider emits points along a straight line toward destination, ‘NavState.distanceMeters‘ monotonically decreases and ‘etaMinutes‘ updates accordingly.
  2. When provider emits an off-route point (distance to destination increases by > ‘navOffRouteThresholdMeters‘), ViewModel triggers recompute and sets ‘onRoute=false‘ or similar flag.

Source Code Coverage Tests

We will measure unit test coverage using Gradle’s reporting (‘./gradlew test‘ produces test results; use ‘jacoco‘ plugin later if coverage percentage is required). Focus coverage on: ‘GeoUtils‘ (100%), ‘SearchRoomsUseCase‘ (happy path + not found), and ‘NavigationViewModel‘ (state transitions). Keep cyclomatic complexity low by keeping ViewModel logic decomposed into small methods (‘shouldRecompute‘, ‘recompute‘). Basis paths: success (on-route), off-route, near-destination floor advice.

4.2.1 Unit Tests and Results

Status: Unit tests are planned and scaffolding is present. Sprint 4 prioritized implementation of live updates and the provider abstraction; automated tests should be added next. To run unit tests locally:

```
# from project root
./gradlew test

gradle test --tests "edu.wku.toppernav.viewmodel.NavigationViewModelTest"
```

Record test results and paste the JUnit XML outputs into the ‘build/reports/tests‘ folder for inclusion in the final report. When tests are added, update this section with pass/fail counts, timings, and any observed failures and mitigations.

4.3 Integration Testing

Integration tests validate the interaction between the ViewModel and the provider and the repository. Two integration targets:

- Robolectric or instrumentation-like test that runs ‘NavigationViewModel‘ with a real ‘FusedLocationProvider-Impl‘ substitute that uses an emulator mock location feed (device/emulator injection). This confirms the provider-to-ViewModel wiring works as expected on a host JVM or emulator.
- Small connected test that launches ‘MainActivity‘, grants permissions programmatically (UI test harness), injects a GPX or sequences of ‘adb emu‘ location updates, and asserts the Navigation UI displays expected ETA/status.

Integration test commands (connected/device):

```
# run connected instrumentation tests
./gradlew connectedAndroidTest
# or run a single instrumentation test
./gradlew connectedAndroidTest --tests "*NavigationViewModelIntegrationTest"
```

4.3.1 Integration Tests and Results

Status: Integration tests are planned; implement the test harness using AndroidX Test and Compose testing libs. Collect logs (adb logcat) during runs and include the NavTick CSVs in the repository for reproducibility.

4.4 System Testing

System testing is manual and runs on a physical device (recommended) to validate end-to-end behavior with real GPS. Test cases:

- Permission and fix test: Install app, grant location permission, walk a short path and verify the Navigation screen shows live updates and ETA changes.
- End-to-end route test: Search for five sample destinations across campus; for each, start navigation and walk to the destination — confirm the arrow, ETA and floor advice behave sensibly.
- Stress test: Run a 15-minute walk simulation at 1 Hz and 2 Hz polling (two separate runs) and measure CPU and battery usage using Android Profiler.

4.4.1 System Tests and Results

Status: The code is instrumented with `Log.d("NAV", ...)` for quick verification. The team should run at least three device sessions and collect `adb logcat` output filtered on `NAV` and a NavTick CSV produced by the app (or by copying logs). Results will be summarized in Sprint 4 final submission.

4.5 Acceptance Testing

Acceptance criteria (pass/fail):

1. App installs and launches on a test device (Android 11+).
2. Location permission flow works; when permission granted and GPS available the Navigation screen shows distance and cardinal direction within 5 m accuracy.
3. ETA updates while walking; median refresh cadence in logs is between 1–2 Hz.
4. No crashes or uncaught exceptions during 15-minute session.

4.5.1 Acceptance Tests and Results

Status: Basic manual acceptance checks were completed in development (emulator and limited physical-device smoke tests). Full acceptance requires 3+ physical-device sessions with NavTick logs; include the CSVs and screenshots in the final submission.

5 Conclusion

This final document consolidates all sprints: problem framing, architecture selection, UI build-out, and live navigation implementation plus testing scaffolding. ToppperNav achieved core objectives—search + real-time directional guidance—while preserving extensibility for advanced routing and voice features. Constraints (indoor accuracy, full path generation) are transparently documented. Recommended future work: integrate campus GIS for precise building polygons, add floor-level indoor model, implement voice guidance, and introduce analytics for usage patterns. The stable abstractions (LocationProvider, NavigationViewModel, Repository) position successors to extend functionality with minimal refactoring. Academic learning goals (version control discipline, requirements traceability, test planning) were met; codebase remains clean, compartmentalized, and ready for enhancement.

6 Appendix

6.1 Software Product Build Instructions

Prerequisites: JDK 17, Android SDK (API 34), Gradle wrapper.

```
# Clone and build
git clone <repo_url>
cd TopperNavApp
./gradlew assembleDebug

# Run unit tests
./gradlew test

# Install on device
adb install -r app/build/outputs/apk/debug/app-debug.apk
```

6.2 Software Product User Guide

User: Open app, search "Building Room" (e.g., Snell Hall B104), tap result, view live distance + ETA, use History for faster re-navigation. Admin: adjust configuration in `AppConfig.kt` (mock mode, thresholds), run tests, review logs.

6.3 Source Code with Comments

Note: Paths relative to project root. For brevity, only key files included; full listing resides in repository.

C.1 app/src/main/java/edu/wku/toppernav/MainActivity.kt
C.2 app/src/main/java/edu/wku/toppernav/viewmodel/NavigationViewModel.kt
C.3 app/src/main/java/edu/wku/toppernav/viewmodel/SearchViewModel.kt
C.4 app/src/main/java/edu/wku/toppernav/domain/usecase/SearchRoomsUseCase.kt
C.5 app/src/main/java/edu/wku/toppernav/data/local/LocationProvider.kt
C.6 app/src/main/java/edu/wku/toppernav/data/local/FusedLocationProviderImpl.kt
C.7 app/src/main/java/edu/wku/toppernav/data/repository/NavigationRepositoryImpl.kt
C.8 app/src/main/java/edu/wku/toppernav/util/GeoUtils.java
C.9 app/src/main/java/edu/wku/toppernav/core/AppConfig.kt
C.10 app/src/main/java/edu/wku/toppernav/ui/screens/NavigationScreen.kt
C.11 app/src/main/java/edu/wku/toppernav/ui/screens/SearchScreen.kt
C.12 app/src/main/java/edu/wku/toppernav/ui/screens/HistoryScreen.kt
C.13 app/src/main/java/edu/wku/toppernav/ui/screens/SettingsScreen.kt