GPS Based Campus Room Finder

Sprint 3
10-28-2025

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

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 defines the boundaries, commitments, and outputs required to deliver the GPS-Based Campus Room Finder. This scope covers all activities necessary to design, implement, test, and document a mobile application that meets the client's expectations while remaining usable and maintainable beyond the project timeline.

Deliverables & Outcomes:

- **Written Reports:** Detailed organizational and technical documents submitted at the conclusion of each of the four sprints.
- **Presentations:** A presentation delivered at the end of each sprint to summarize progress and demonstrate results.
- **Evaluations:** Peer evaluation forms submitted individually by team members after each sprint.
- **Final Product:** A fully tested, documented, and maintainable Android mobile application that provides GPS-based navigation to campus buildings and rooms.

Work Required:

- **Tasks:** All development tasks including source code creation, user interface design, system integration, testing, and documentation. Additional requirements will be integrated as identified throughout the project.
- **Team:**
  - Kaden Hunt — Project Manager
  - Aaron Downing — Documentation Draft
  - Ryerson Brower — Research Coordinator, Task Manager
- **Time Commitment:** Work will be divided across four sprints. Each team member will contribute 8–10 hours per week to development, meetings, and documentation.
- **Resources:** GitHub will serve as the version control system and task management platform. The documentation will be written collaboratively in Texmaker. The development will take place on personal laptops running Windows 10 or later which will meet the requirements of Android Studio. of Android Studio.
- **Schedule:** Deliverables align with the four milestone deadlines outlined on Blackboard. Weekly client meetings occur on Tuesdays at 12:35 p.m. in Snell Hall B104. Internal team meetings will take place on Thursdays at 2:00 p.m.

Altogether, this scope establishes what will be delivered, the benefits it provides, and the foundation for successful implementation

1.3 Technical Requirements

1.3.1 Functional Requirements

Mandatory Functional Requirements
The application will use GPS coordinates to determine the user's current location within the campus boundaries.
The application will allow the user to search for a specific building and room number using a text-based input.
The application will generate a step-by-step navigation route from the user's current location to the selected room.
The application will have an interactive display to navigate the user to the building and room.
The application will provide an estimated travel time based on the mobile location of the user.
Extended Functional Requirements
The application will provide voice-guided navigation for hands-free use.
The application will allow users to bookmark or "favorite" frequently visited rooms for quicker searches.
The application will provide a "recent searches" history so users can quickly reselect prior destinations

The functional requirements for the WKU GPS-based campus room finder are designed to help WKU students and faculty easily locate rooms across campus. By using GPS coordinates to determine the user's current location, the application provides accurate, real-time directions, allowing users to navigate campus quickly and effectively. The interactive display offers clear step-by-step guidance to the desired building and room number, featuring a user-friendly interface that makes input, ensuring easy accessibility for all users. To get rid of any other unnecessary confusion, the application will provide an estimated travel time based on the mobile location of the user. This feature also allows users to make better decisions about which route to take depending on their time constraints between classes. The applications goal is to address common problems such as getting lost or arriving late to class, enhancing convenience and creating a smoother, more reliable navigation experience across the WKU campus.

1.3.2 Non-Functional Requirements

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

The mandatory non functional requirements for the WKU GPS-based campus room finder ensure the application performs reliably, efficiently, and securely while providing users with a positive application experience. By requiring location updates with an accuracy of within 5 meters under clear sky conditions, the app guarantees precise position for navigating campus. Delivering route generation results within 2 seconds ensures that users receive routes efficiently without unnecessary delays. Visual and text-based route guidance enhances accessibility and makes navigation possible for all users. Requiring all project source code must be developed by the CS 360 project team helps achieve the desired learning outcomes of the class and encourages accountability throughout the team in a real-world setting. Using a database enables efficient storage and retrieval of all building and rooms on campus. Additionally, gathered and optimized security metrics will ensure the application remains fast, safe, and easy to use, while also meeting quality standards set out by the client. Collectively, these requirements provide a reliable and high-quality tool for campus navigation.

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 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

Our wire-frame shows the basic interactions one can do with our app. The user enters the app and either choose to search a location to go to or to use history to go to a previous location. Once one of these 2 options is chosen then a arrow pop ups on screen and starts pointing in the direction to go along side the arrow is some text instructions telling the user where to go. Once the user makes it near the location the app with tell them they are near and to look at room numbers. It will then give the option to put in a new location.

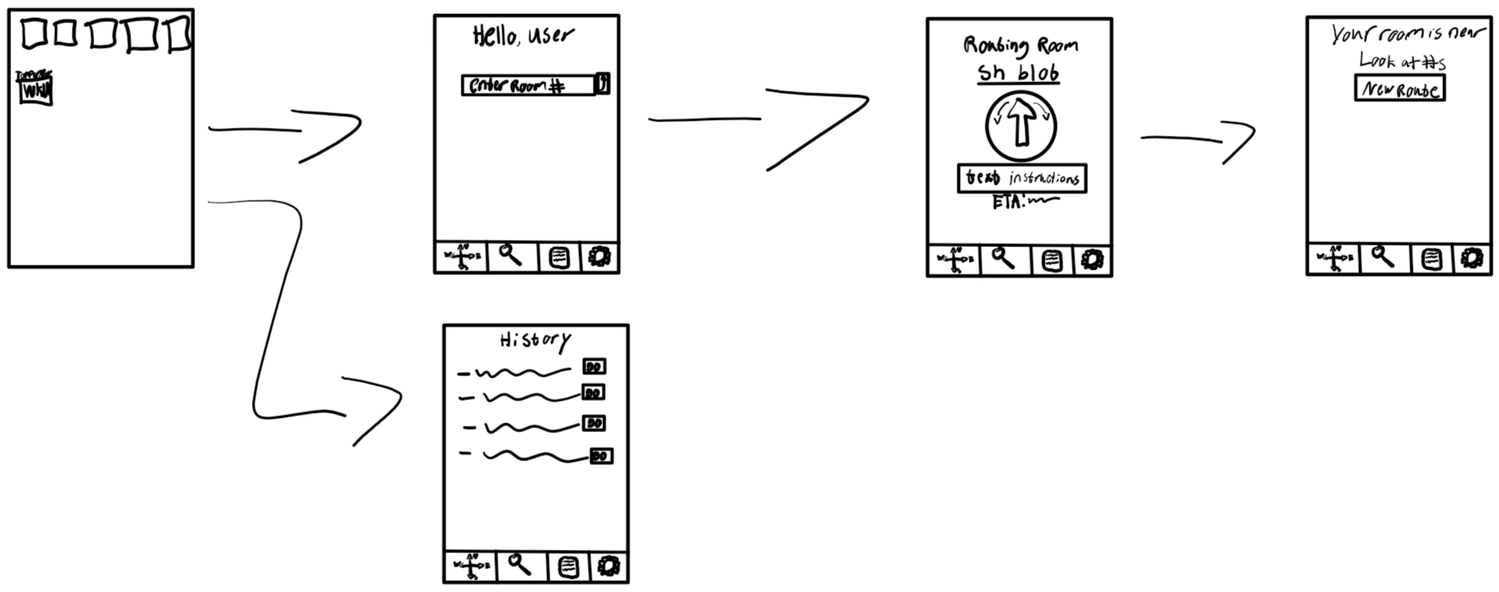


Figure 1: The wireframe of the basic use of our GPS app.

2.3 UML

2.3.1 Class Diagrams

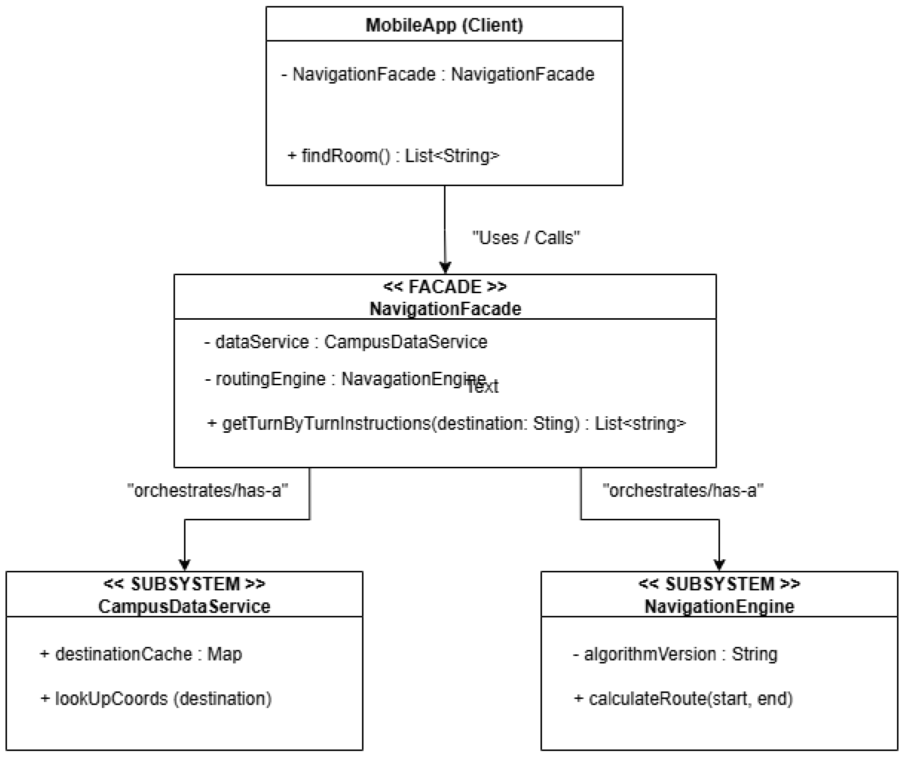

MobileApp (Client)
- NavigationFacade : NavigationFacade
+ findRoom() : List<String>
"Uses / Calls"
<< FACADE >>
NavigationFacade
- dataService : CampusDataService
- routingEngine : NavagationEngine
Text
+ getTurnByTurnInstructions(destination: Sting) : List<string>
"orchestrates/has-a"
"orchestrates/has-a"
<< SUBSYSTEM >>
CampusDataService
+ destinationCache : Map
+ lookUpCoords (destination)
<< SUBSYSTEM >>
NavigationEngine
- algorithmVersion : String
+ calculateRoute(start, end)

Figure 2: Structural Design Pattern: *Facade* for routing & map services.

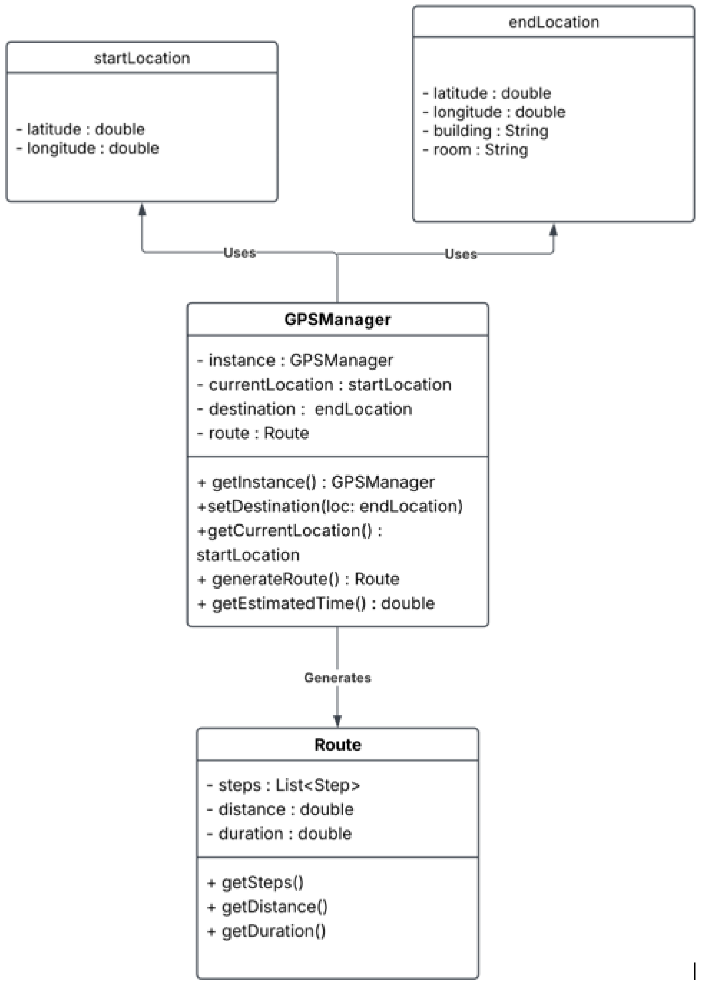


Figure 3: Creational Design Pattern: *Singleton* for AppConfig.

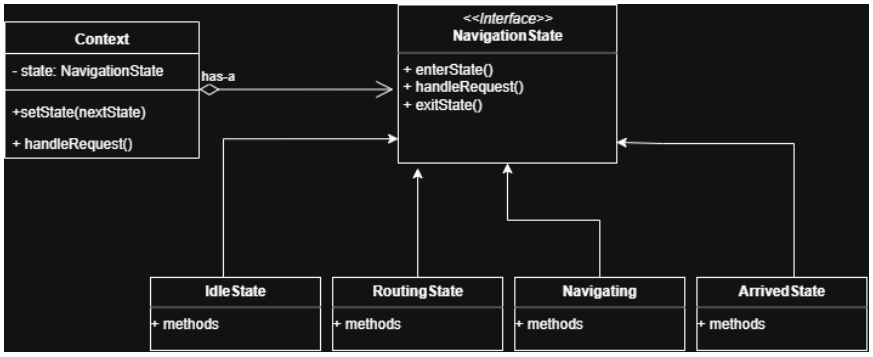


Figure 4: Behavioral Design Pattern: State.

2.3.2 Use Case Diagrams

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

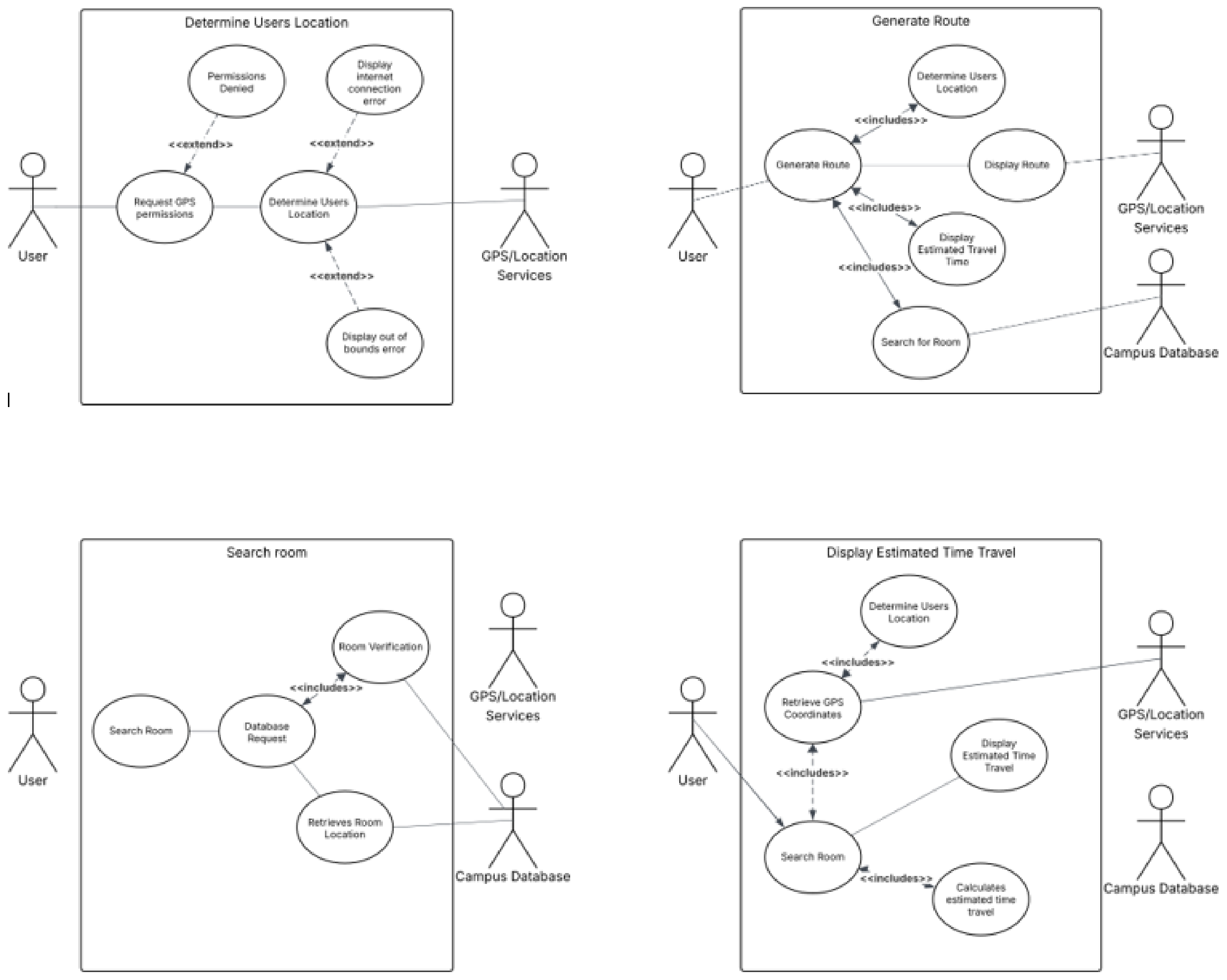
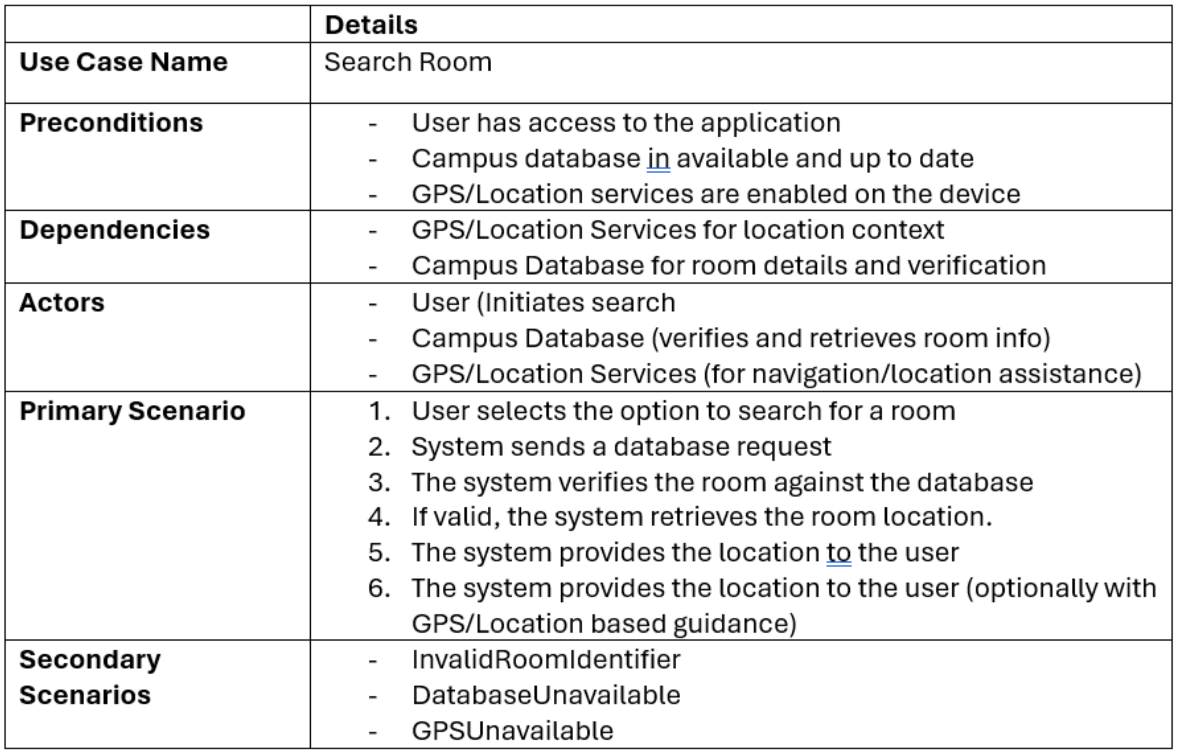


Figure 5: Use Case Diagrams.

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



	Details
Use Case Name	Search Room
Preconditions	- User has access to the application - Campus database in available and up to date - GPS/Location services are enabled on the device
Dependencies	- GPS/Location Services for location context - Campus Database for room details and verification
Actors	- User (Initiates search - Campus Database (verifies and retrieves room info) - GPS/Location Services (for navigation/location assistance)
Primary Scenario	1. User selects the option to search for a room 2. System sends a database request 3. The system verifies the room against the database 4. If valid, the system retrieves the room location. 5. The system provides the location to the user 6. The system provides the location to the user (optionally with GPS/Location based guidance)
Secondary Scenarios	- InvalidRoomIdentifier - DatabaseUnavailable - GPSUnavailable

Figure 6: 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.

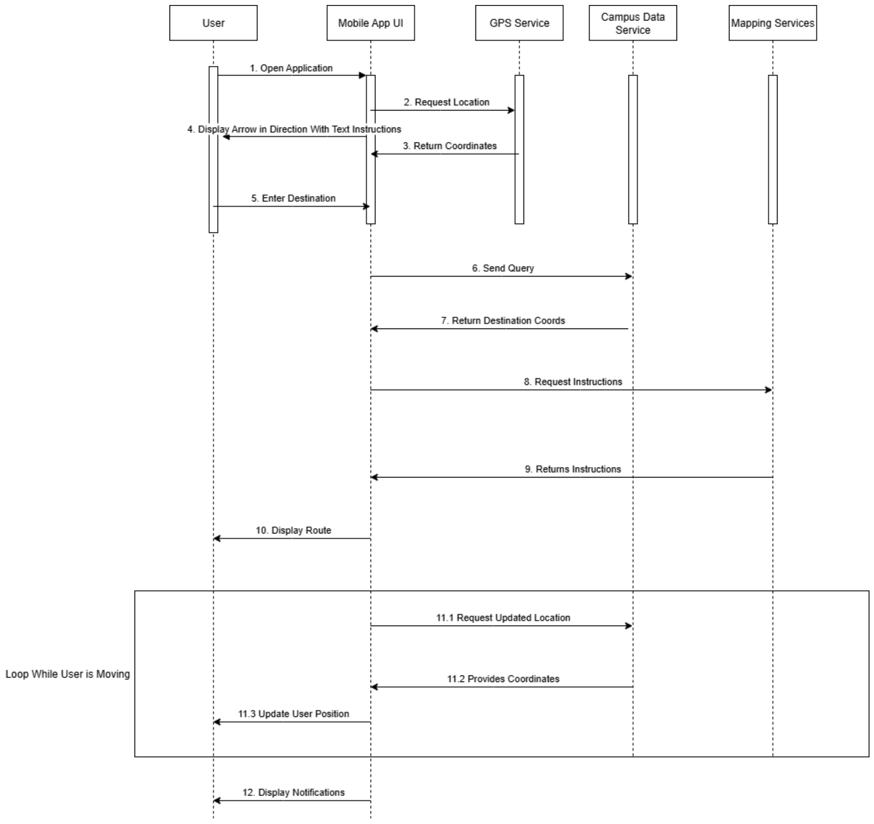


Figure 7: Sequence diagram for the user route request, generation, and update.

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.

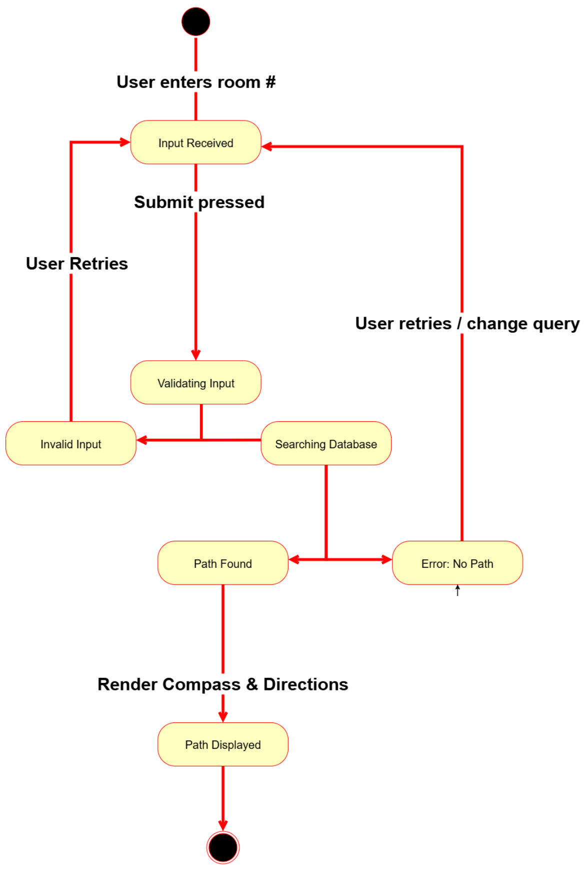


Figure 8: State UML diagram for the room search feature, showing user input, validation, query, error handling, and path display.

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.

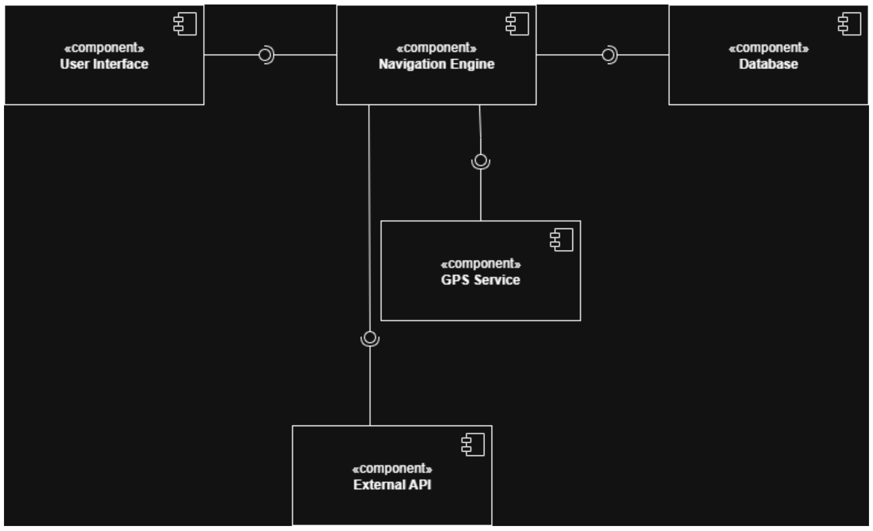


Figure 9: Component diagram.

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.

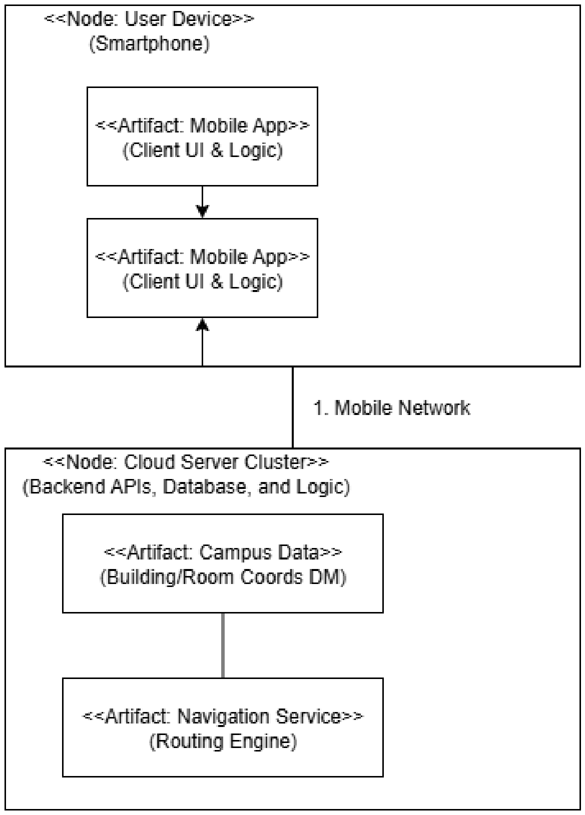


Figure 10: Deployment diagram for mobile + external services.

2.4 Mapping of Source Code to UML Diagrams

Sprint 3 focused on the interactive UI only. The classes and adapters shown in the UML are planned for Sprint 4. This subsection documents the planned mapping so the design stays testable and traceable as implementation begins next sprint.

Planned class and file mapping

1) Facade pattern (Fig. 2) Planned files: core/AppConfig.java, core/RouteFacade.java, core/MapProviderAdapter.java Responsibility: expose a single entry point for route generation and map services so the UI remains decoupled.

2) Singleton pattern for configuration (Fig. 3) Planned files: core/AppConfig.java Responsibility: store app-wide flags such as units and theme, and provide lazy accessors for settings.

3) State pattern for navigation flow (Fig. 4) Planned files: nav/state/BrowsingState.java, nav/state/SearchingState.jav nav/state/NavigatingState.java, nav/state/ErrorState.java Responsibility: model screen-to-screen transitions and error handling without tightly coupling UI logic.

4) Use Case diagrams (Fig. 5) Current UI files: ui/screens/HomeScreen.kt, ui/screens/SearchScreen.kt, ui/screens/NavigationScreen.kt, ui/components/SearchBar.kt, ui/components/HistoryList.kt Coverage: determine location, search for room, generate route, show time estimate.

5) Sequence diagram Planned linkage: ui ViewModel methods will call RouteFacade which calls MapProvider-Adapter, which retrieves cached building and room data from local database access classes. Planned files: ui/viewmodel/SearchViewModel.kt, data/RoomRepository.kt, data/BuildingRepository.kt

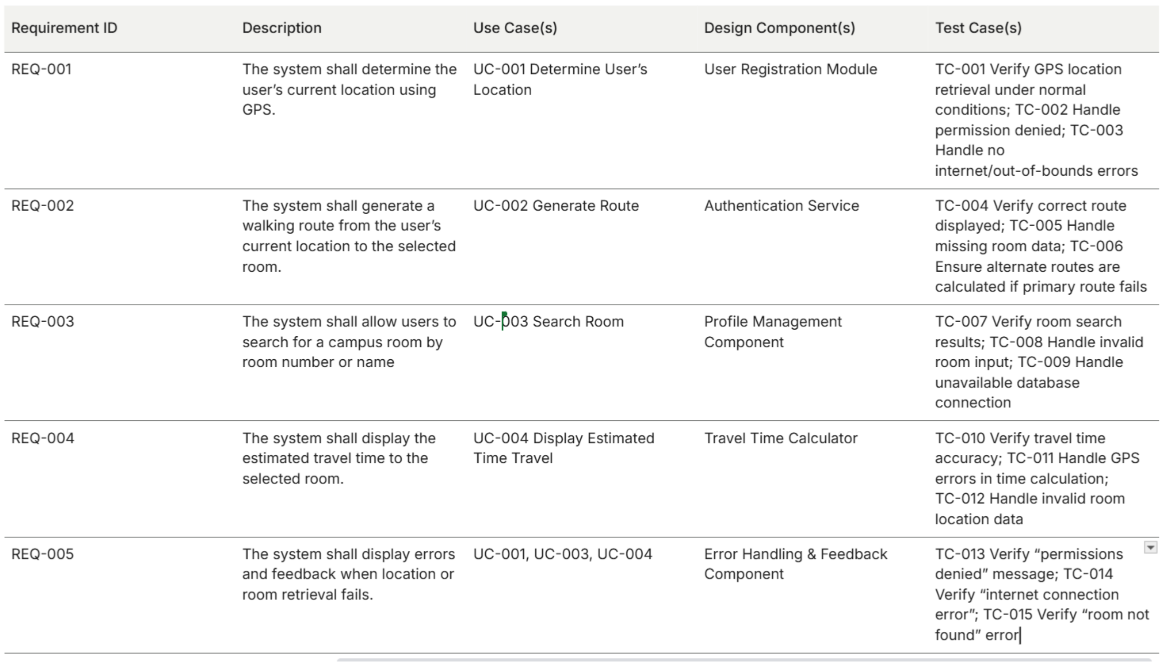
6) State diagram Planned linkage: the state classes listed above will drive which screen renders and which actions are enabled. Planned files: nav/state/* as listed in item 3

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.



Requirement ID	Description	Use Case(s)	Design Component(s)	Test Case(s)
REQ-001	The system shall determine the user's current location using GPS.	UC-001 Determine User's Location	User Registration Module	TC-001 Verify GPS location retrieval under normal conditions; TC-002 Handle permission denied; TC-003 Handle no internet/out-of-bounds errors
REQ-002	The system shall generate a walking route from the user's current location to the selected room.	UC-002 Generate Route	Authentication Service	TC-004 Verify correct route displayed; TC-005 Handle missing room data; TC-006 Ensure alternate routes are calculated if primary route fails
REQ-003	The system shall allow users to search for a campus room by room number or name	UC-003 Search Room	Profile Management Component	TC-007 Verify room search results; TC-008 Handle invalid room input; TC-009 Handle unavailable database connection
REQ-004	The system shall display the estimated travel time to the selected room.	UC-004 Display Estimated Time Travel	Travel Time Calculator	TC-010 Verify travel time accuracy; TC-011 Handle GPS errors in time calculation; TC-012 Handle invalid room location data
REQ-005	The system shall display errors and feedback when location or room retrieval fails.	UC-001, UC-003, UC-004	Error Handling & Feedback Component	TC-013 Verify "permissions denied" message; TC-014 Verify "internet connection error"; TC-015 Verify "room not found" error

Figure 11: Requirements Traceability Table.

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

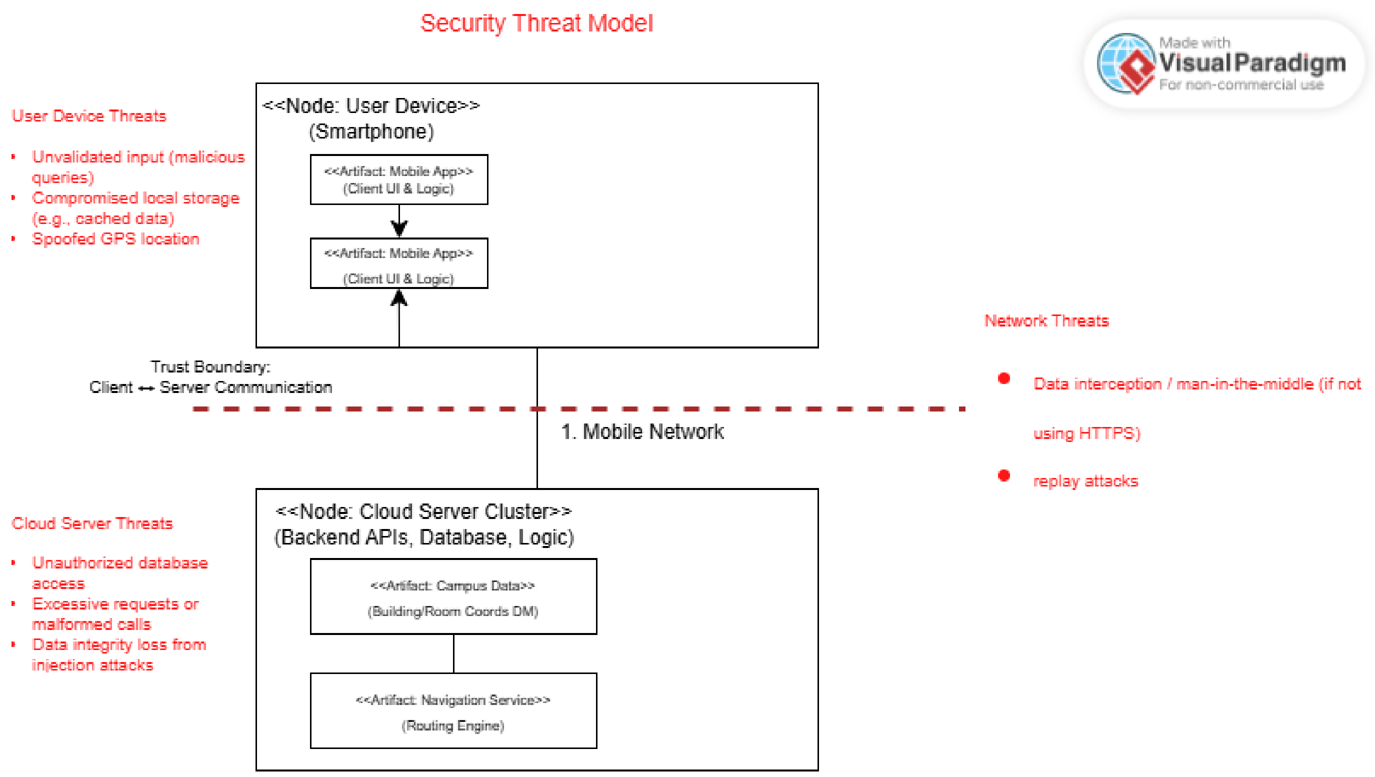

Security Threat Model
Made with Visual Paradigm For non-commercial use
User Device Threats
Unvalidated input (malicious queries)
Compromised local storage (e.g., cached data)
Spoofed GPS location
<<Node: User Device>> (Smartphone)
<<Artifact: Mobile App>> (Client UI & Logic)
<<Artifact: Mobile App>> (Client UI & Logic)
Trust Boundary: Client ↔ Server Communication
1. Mobile Network
Network Threats
Data interception / man-in-the-middle (if not using HTTPS)
replay attacks
Cloud Server Threats
Unauthorized database access
Excessive requests or malformed calls
Data integrity loss from injection attacks
<<Node: Cloud Server Cluster>> (Backend APIs, Database, Logic)
<<Artifact: Campus Data>> (Building/Room Coords DM)
<<Artifact: Navigation Service>> (Routing Engine)

Figure 12: 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 $\leq$ 3 seconds
- Location accuracy within ± 5 meters
- User interface response delay $\leq$ 200 ms
- Database query response $\leq$ 1 second
- Support for $\geq$ 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 $\leq 3\,\mathrm{s}$ (median), UI tap responsiveness $\leq 200\,\mathrm{ms}$, navigation refresh 1–2 Hz with no spikes ($<1\%$ frames $>16\,\mathrm{ms}$). 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:
Introduction:
Test item:
Features to test/not to test:
Approach:
Test deliverables:
Item pass/fail criteria:
Environmental needs:
Responsibilities:
Staffing and training needs:
Schedule:
Risks and Mitigation:
Approvals:

4.2 Unit Testing

Text goes here.

4.2.1 Source Code Coverage Tests

Text goes here.

4.2.2 Unit Tests and Results

Text goes here.

4.3 Integration Testing

Text goes here.

4.3.1 Integration Tests and Results

Text goes here.

4.4 System Testing

Text goes here.

4.4.1 System Tests and Results

Text goes here.

4.5 Acceptance Testing

Text goes here.

4.5.1 Acceptance Tests and Results

Text goes here.

5 Conclusion

This sprint delivered a working UI flow for the GPS-based Campus Room Finder: the home screen, search and history interactions, and the navigation view with time estimates displayed. The goal was to validate usability and interaction design before committing to routing and data layers. That objective was achieved: the app boots, the screens render reliably, and the primary user journeys are clear and fast to operate.

Several rubric sections are documented as plans because the underlying code is not yet implemented by design this sprint. Specifically, the mapping from source to UML, performance tests, and database-backed routing are described as prospective work starting in Sprint 4. The workload plan, and bottleneck analysis outline how we will measure and improve latency, refresh cadence, and resource usage on real devices.

Shortcomings in this iteration include lack of live routing, no persistent building and room dataset, and no measured performance results. The next sprint will implement the Facade and State layers, connect the local database, instrument timers, and produce graphs and traceable mappings back to the diagrams. With the UI foundation in place, the project is positioned to meet the remaining performance, security, and documentation requirements on schedule.

6 Appendix

6.1 Software Product Build Instructions

Project state This sprint only includes the user interface prototype of the TopperNav Android app. No database or routing logic is connected yet. The app currently runs on the Android emulator within Android Studio.

Build setup 1) Install Android Studio (Jellyfish or newer). 2) Ensure Android SDK Platform 34 and Build Tools are installed. 3) Clone the TopperNav GitHub repository. 4) Open the project in Android Studio and allow Gradle to sync. 5) In the AVD Manager, create an emulator (for example, Pixel 6 – Android 14). 6) Click Run to launch the app on the emulator.

Future deployment In Sprint 4, the app will be deployed to a physical Android device for on-device testing and performance measurement. This will allow testing of GPS features, latency, and resource usage.

6.2 Software Product User Guide

General user 1) Open the app to the home screen. 2) Type a building and room (for example, Snell Hall B104) or open History to select a recent search. 3) The navigation view shows direction cues and a time estimate based on the mock flow in Sprint 3. 4) Use back to return to home and start another search.

Administrator (development team) Use Android Studio to install debug builds on an emulator or lab device.

6.3 Source Code with Comments

This section includes the full Kotlin source files that define the current user interface for the TopperNav mobile application. Since Sprint 3 focused exclusively on UI development, these files represent the four core screens in the app: Search, Navigation, History, and Settings. Future sprints will integrate these interfaces with the underlying route engine, GPS access, and SQLite database. Each file is shown in full below and includes inline comments for clarity.

6.3.1 C.1 SearchScreen.kt

```
package edu.wku.toppernav.ui

import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Button
import androidx.compose.material3.Card
import androidx.compose.material3.ListItem
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp

@Composable
fun SearchScreen(
    query: String,
    onQueryChange: (String) -> Unit,
    onEnter: (String) -> Unit
) {
    val rooms = listOf(
        "SH 202", "SH 210", "SH 305",
        "MMTH 100", "MMTH 212",
        "HC 101", "HC 220",
        "DSU 1033", "DSU 3002"
    )
    val results = rooms.filter { it.contains(query.trim(), ignoreCase = true) }

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        Text("Find a room", style = MaterialTheme.typography.headlineSmall)

        OutlinedTextField(
            value = query,
            onValueChange = onQueryChange,
            modifier = Modifier.fillMaxWidth(),
            singleLine = true,
            label = { Text("Building and room (e.g., SH 202)") },
            placeholder = { Text("Type building or room number") },
        )
        Button(
            onClick = { onEnter(query) },
            enabled = query.isNotBlank()
        ) {
            Text("Enter")
        }

        LazyColumn(
            modifier = Modifier.fillMaxSize(),
            contentPadding = PaddingValues(bottom = 24.dp),
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            items(results) { item ->
                Card(
                    modifier = Modifier
                        .fillMaxWidth()
                        .clickable { onEnter(item) }
                ) {
                    ListItem(
                        headlineContent = { Text(item) },
                        supportingContent = { Text("Tap to navigate") }
                    )
                }
            }
        }
    }
}
```

6.3.2 C.2 NavigationScreen.kt

```
package edu.wku.toppernav.ui

import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Navigation
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.unit.dp
```

```
@Composable
fun NavigationScreen(
    destination: String?,
    etaText: String,
    travelTimeText: String,
    steps: List<String>
) {
    Surface(modifier = Modifier.padding(16.dp)) {
        Column(verticalArrangement = Arrangement.spacedBy(12.dp)) {
            Text(
                text = destination?.ifBlank { "No destination selected" } ?: "No destination selecte
                style = MaterialTheme.typography.headlineSmall
            )

            // Compass placeholder: circle + upward navigation arrow
            Box(
                modifier = Modifier
                    .fillMaxWidth()
                    .height(280.dp),
                contentAlignment = Alignment.Center
            ) {
                Box(
                    modifier = Modifier
                        .height(220.dp)
                        .fillMaxWidth(0.6f)
                        .clip(CircleShape)
                        .background(MaterialTheme.colorScheme.surfaceVariant),
                    contentAlignment = Alignment.Center
                ) {
                    Icon(
                        imageVector = Icons.Filled.Navigation,
                        contentDescription = "Compass arrow",
                        tint = MaterialTheme.colorScheme.primary
                    )
                }
            }

            Text("ETA: $etaText", style = MaterialTheme.typography.bodyLarge)
            Text("Travel time: $travelTimeText", style = MaterialTheme.typography.bodyLarge)

            Text("Steps :", style = MaterialTheme.typography.titleMedium)
            steps.forEach { s ->
                Text("• $s", style = MaterialTheme.typography.bodyMedium)
            }
        }
    }
}
```

6.3.3 C.3 HistoryScreen.kt

```
package edu.wku.toppernav.ui
```

```
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Card
import androidx.compose.material3.ListItem
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp

@Composable
fun HistoryScreen(
    items: List<String>,
    onSelect: (String) -> Unit
) {
    var filter by remember { mutableStateOf("") }
    val filtered = items.filter { it.contains(filter.trim(), ignoreCase = true) }

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        Text("History", style = MaterialTheme.typography.headlineSmall)
        Text("Search history", style = MaterialTheme.typography.labelLarge)

        OutlinedTextField(
            value = filter,
            onValueChange = { newValue -> filter = newValue },
            modifier = Modifier.fillMaxWidth(),
            singleLine = true
        )

        LazyColumn(
            modifier = Modifier.fillMaxSize(),
            contentPadding = PaddingValues(bottom = 24.dp),
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            items(filtered) { item ->
                Card(
                    modifier = Modifier
                        .fillMaxWidth()
                        .clickable { onSelect(item) }
                ) {
                    ListItem(
                        headlineContent = { Text(item) },
                        supportingContent = { Text("Tap to load into Search") }
                    )
                }
            }
        }
    }
}
```

6.3.4 C.4 SettingsScreen.kt

```
package edu.wku.toppernav.ui

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp

@Composable
fun SettingsScreen(
    name: String,
    onNameChange: (String) -> Unit
) {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.spacedBy(12.dp)
    ) {
        Text("Settings", style = MaterialTheme.typography.headlineSmall)
        OutlinedTextField(
            value = name,
            onValueChange = onNameChange,
            label = { Text("Display name for greeting") },
            placeholder = { Text("Leave blank for \"Hilltopper\"") },
            singleLine = true
        )
        Text(
            "UI-only now. Later you’ll wire GPS, ETA, step guidance, and a database."
        )
    }
}
```