

A

Project Report

On

**SMART ROUTE MONITORING AND ALERT SYSTEM**

Submitted in partial fulfillment of the requirement for the VIIIth semester

**Bachelor of Technology (Computer Science & Engineering)**

By

**Navraj Devali (PV-21610244)**

**Neeraj Koshyari (PV-21610246)**

Under the Guidance of

**Mr. Rahul Kumar Singh**

**Assistant Professor**

**Department of CSE**

****

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**GRAPHIC ERA HILL UNIVERSITY, BHIMTAL CAMPUS**

**SATTAL ROAD, P.O. BHOWALI,**

**DISTRICT- NAINITAL-263132**

**2024- 2025**

**STUDENT’S DECLARATION**

I, **Navraj Devali, Neeraj koshyari** here-by declare the work, which is being presented in the project, entitled “**Smart Route Monitoring And Alert System**” in partial fulfillment of the requirement for the award of the degree **Bachelor of Technology (B.Tech)** in the session **2024-2025**, is an authentic record of our own work carried out under the supervision of “**Mr. Rahul Kumar Singh”, Assistant Professor, Department of CSE, Graphic Era Hill University, Bhimtal.**

The matter embodied in this project has not been submitted by us for the award of any other degree.

Date: ………… Navraj Devali Neeraj Koshyari

**CERTIFICATE**

This is to certify that the project report entitled **“Smart Route Monitoring and Alert System”** being submitted by **Navraj Devali (2161244), Neeraj Koshyari** (**2161246)** of B.tech (CSE) to Graphic Era Hill University, Bhimtal Campus for the award of bonafide work carried out by them. They have worked under my guidance and supervision and fulfilled the requirement for the submission of a report.

**Mr. Rahul Kumar Singh Mr. Ayush Kapri Dr. Ankur Singh Bist**

**(Project Guide) (Project Coordinator) (HOD, CSE Dept.)**

**External Examiner**

**ACKNOWLEDGEMENT**

We take immense pleasure in thanking Honorable Director **“Prof. (Col.) Anil Nair (Retd.)”,** GEHU Bhimtal Campus to permit us and carry out this project work with his excellent and optimistic supervision. This has all been possible due to his novel inspiration, able guidance, and useful suggestions that helped us to develop as creative researcher and complete the research work, in time.

Words are inadequate in offering our thanks to GOD for providing us everything that we need. We again want to extend thanks to our President **“Prof. (Dr.) Kamal Ghanshala”** for providing us with all infrastructure and facilities to work in need without which this work could not be possible.

Many thanks to **“Dr. Ankur Singh Bist”** (Head, Department of Computer Science and Engineering, GEHU Bhimtal Campus), our project co-ordinator **“Mr. Ayush Kapri”** (Assistant Professor, Department of Computer Science and Engineering, GEHU Bhimtal Campus), our project guide **“Mr. Rahul Kumar Singh, Assistant Professor, CSE Dept.”**  and other faculties for their insightful comments, constructive suggestions, valuable advice, and time in reviewing this thesis. Finally, yet importantly, we would also like to pay my sincere thanks to all my friends and well-wishers for their help and wishes for the successful completion of this research.

**Navraj Devali Neeraj Koshyari**

**(2161244) (2161246)**

**ABSTRACT**

The navigation application combines intuitive mapping, dynamic checkpoint management, and integrated safety features into a seamless user experience. At its core, the app leverages Google Maps and Places APIs to render accurate route information and allow natural language destination entry. In **Automatic** mode, the system analyzes the path’s geometry—calculating turn angles and segment lengths—to distribute checkpoints at sensible intervals, each stamped with an expected travel time based on straight‐away or winding segments. In **Manual** mode, users exert fine‐grained control, tapping the map to set checkpoints and specifying how long they anticipate taking to reach each one. Once the route and its waypoints are established, a built‐in **PathSimulator** class animates virtual movement along the plotted polyline, updating the “current location” marker at configurable speeds.

To ensure user safety, the app interleaves location tracking with a native “Are You Safe?” module. As the simulation (or real travel) arrives at each checkpoint, the system checks whether the elapsed time exceeds the user’s expectation. If so, a modal dialog appears with a ten‐second countdown; failure to confirm safety or an explicit “No” response triggers an SOS SMS—dispatched via a custom native SMS module—to a preconfigured contact stored persistently in AsyncStorage. The SMS message includes a Google Maps link pinpointing the user’s last known GPS coordinates. This front‐end–only implementation demonstrates cohesive integration of React Native components, native modules for permissions and SMS, and in‑app storage, resulting in a robust prototype for navigation with built‑in emergency response.

**TABLE OF CONTENTS**

**DECLARATION……………………………..……………………………………….…..2**

**CERTIFICATE …………………………………………………………………………..3**

**ACKNOWLEDGEMENT...……………………...……………………………………....4**

**ABSTRACT ……………………………………………………………………………....5**

**TABLE OF CONTENTS……………………………………………………………..…..6**

**LIST OF FIGURES……………………………………………………………………….8**

**LIST OF TABLE………………………………………………………………………….8**

1. **INTRODUCTION……………………………………………………………………….9**
   1. Background………………………………………………………………………….9
   2. Objectives…………………………………………………………………………...9
   3. Scope……………………………………………………………………………….10
2. **LITERATURE REVIEW & RELATED WORK…………………………………...11**
   1. Mobile Navigation Applications…………………………………………………...11
   2. Checkpoint & Simulation Mechanisms……………………………………………11
   3. SOS & Emergency Alert Systems………………………………………………….12
3. **SYSTEM REQUIREMENTS………………………………………………………….13**
   1. Functional Requirements…………………………………………………………...13
      1. Mode Selection (Automatic vs Manual)…………………………………….13
      2. Destination Entry and Route Plotting……………………………………….14
      3. Checkpoint Management & Expected Times……………………….……….14
      4. Simulation Controls & SOS Trigger…………………………………………15
   2. Non-Functional Requirements………………………………………………………16
      1. Performance………………………………………………………………….16
      2. Usability……………………………………………………………………...17
      3. Reliability & Safety………………………………………………………….17
   3. Constraints and Assumptions………………………………………………………..18
      1. Constraints……………………………………………………………………18
      2. **Assumptions………………………………………………………………….19**
4. **SYSTEM ARCHITECTURE & DESIGN……………………………………………21**
   1. High-Level Architecture Diagram………………………………………………….21
   2. Module Decomposition…………………………………………………………….22
      1. UI Components……………………………………………………………..22
      2. Map & Directions Integration………………………………………………23
      3. SosModule (Native SMS)…………………………………………………..24
   3. Data Storage & Management (AsyncStorage for Settings)………………………..25
   4. Permission Handling……………………………………………………………….26
5. **USER INTERFACE DESIGN……………………………………………………….28**
   1. Screen Flow & Navigation…………………………………………………………28
   2. Key Screens………………………………………………………………………...29
      1. Home (Mode Selection)……………………………………………………..29
      2. Automatic Mode Screen…………………………………………………….30
      3. Manual Mode Screen………………………………………………………..31
6. **IMPLEMENTATION DETAILS……………………………………………………33**
   1. Development Environment Setup………………………………………………….33
   2. Google Maps & Places API Integration……………………………………………35
   3. Checkpoint Logic (Automatic vs Manual)…………………………………………37
      1. Automatic Checkpoint Logic……………………………………………….37
      2. Manual Checkpoint Logic………………………………………………….38
   4. PathSimulator Class Implementation……………………………………………...39
   5. SOS Native Module Integration…………………………………………………...40
   6. Settings & AsysncStorage…………………………………………………………41
7. **TESTING AND VALIDATION……………………………………………………..42**
   1. Unit Testing………………………………………………………………………..42
   2. Integration Testing…………………………………………………………………42
   3. End-to-End/User Testing…………………………………………………………..43
   4. Performance & Stress Testing……………………………………………………...44
   5. Test Case Matrix……………………………………………………………………45

**CHAPTER 8: DISCUSSION AND FUTURE WORK………….………………………46**

**CHAPTER 9: CONCLUSION………………………………………………………........49**

**CHAPTER 10: REFERENCES…………………………………………………………..51**

**CHAPTER 11: APPENDICES…………………………………………………………...52**

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **Figure Number** | **Title** | **Page Number** |
| Fig 2.1 | Emergency Alert Messages | 12 |
| Fig 3.1 | Mode Selection Screen | 13 |
| Fig 3.2 | Destination Entry | 14 |
| Fig 3.3 | Automatic Checkpoint Info. | 15 |
| Fig 3.4 | Manual Checkpoint Info. | 15 |
| Fig 4.1 | High-Level Architecture Diagram | 21 |
| Fig 4.2 | Automatic Mode | 24 |
| Fig 4.3 | Manual Mode | 24 |
| Fig 4.4 | Settings Screen | 26 |
| Fig 5.1 | Home Screen | 32 |
| Fig 5.2 | Mode Selection | 32 |
| Fig 5.3 | Automatic Mode | 32 |
| Fig 5.4 | Manual Mode | 32 |
| Fig 6.1 | File Structure | 34 |

**LIST OF TABLE**

|  |  |
| --- | --- |
| **Table Number** | **Page Number** |
| Table 4.1 | 23 |
| Table 4.2 | 26 |
| Table 5.1 | 29 |
| Table 6.1 | 33 |
| Table 6.2 | 35 |
| Table 6.3 | 39 |
| Table 7.1 | 44 |
| Table 7.2 | 45 |
| Table 8.1 | 47 |

**CHAPTER 1**

**INTRODUCTION**

Modern mobile navigation applications have revolutionized the way users plan and follow routes, offering real‑time guidance, dynamic route optimization, and location‑based services[1]. However, traditional navigation apps assume that users are always able to follow the recommended pace and do not provide built‑in safety mechanisms for situations when users may fall behind schedule or find themselves in distress. This project addresses that gap by developing an Android‑only React Native application that not only plots routes and checkpoints but also simulates movement along the path and proactively raises an SOS alert if the user is late to a checkpoint.

* 1. **BACKGROUND**

With the proliferation of smartphones, navigation apps have become indispensable for drivers, cyclists, and pedestrians alike. React Native was chosen for its ability to deliver native‑like performance on Android while maintaining a single codebase in JavaScript [1]. The Google Maps and Places APIs underpin the route plotting and destination search functionality, allowing users to enter any address worldwide and view turn‑by‑turn directions.

* 1. **OBJECTIVES** 
     + **Automated vs. Manual Checkpoint Placement**:
     + **Automatic mode:** Algorithmically places checkpoints along the route at specified distance or angle thresholds and assigns expected traversal times.
     + **Manual mode:** Allows users to tap the map, add custom checkpoints, and enter expected times in minutes between successive waypoints.
     + **Simulation of Movement:**
     + Mock “vehicle” movement along the plotted polyline, updating location in real time to test checkpoint logic.
     + **Safety and SOS Alerting:**
     + At each checkpoint, the system measures elapsed time against the expected threshold. If the user is late, a native‑module modal (“Are you safe?”) with a 10 s countdown appears.
     + Non‑response or a “No” triggers an automatic SOS SMS (via a native SMS module) to pre‑configured contacts stored in AsyncStorage, embedding the user’s current GPS coordinates.
  2. **SCOPE**

This application is purely frontend, implemented in React Native and tested on Android devices only. All routing, simulation, and SMS functionality relies on publicly available APIs and native modules—no backend services are required.

**CHAPTER 2**

**LITERATURE REVIEW & RELATED WORK**

**2.1 MOBILE NAVIGATION APPLICATIONS**

Mobile navigation applications have significantly transformed how users travel and explore unknown geographies. Applications like **Google Maps**, **Waze**, and **HERE WeGo** have dominated the market by offering features such as **real-time traffic updates**, **turn-by-turn directions**, and **location sharing** [2]. These applications primarily focus on optimizing travel efficiency and user experience but often lack **context-aware safety mechanisms**, especially in high-risk or remote environments.

A notable characteristic across these platforms is their reliance on the **Global Positioning System (GPS)** and **mapping APIs** to compute and visualize routes. However, none of these applications proactively monitor whether a user has reached critical points (i.e., checkpoints) on their route, nor do they **simulate travel with safety time buffers** or provide **emergency escalation mechanisms** in case of deviations.

In contrast, our project bridges this gap by introducing **checkpoint-based progress tracking and simulation** combined with **real-time SOS capabilities**, tailored for vulnerable or emergency-prone travel contexts.

**2.2 CHECKPOINT & SIMULATION MECHANISMS**

Simulation techniques are widely used in **logistics**, **autonomous navigation**, and **gaming environments** to mimic real-world movement. In mobile applications, simulations are often used to test map routing, time predictions, or route optimizations.

Some research-backed implementations, such as **agent-based models** in vehicular simulations, simulate how an object moves across space with respect to time and distance [3]. While these models offer theoretical and academic depth, there is a lack of accessible, user-facing tools that simulate route progression and evaluate punctuality at specified waypoints.

In our system, we introduce a **PathSimulator class** that accepts GPS coordinates and dynamically simulates location updates at set intervals. Using **Haversine distance calculations[3]**, it determines proximity to each checkpoint and manages a **checkpoint-specific countdown**. This is critical to ensuring that if a user fails to reach a checkpoint within the expected time, the app will prompt for a safety check or initiate an **SOS escalation**.

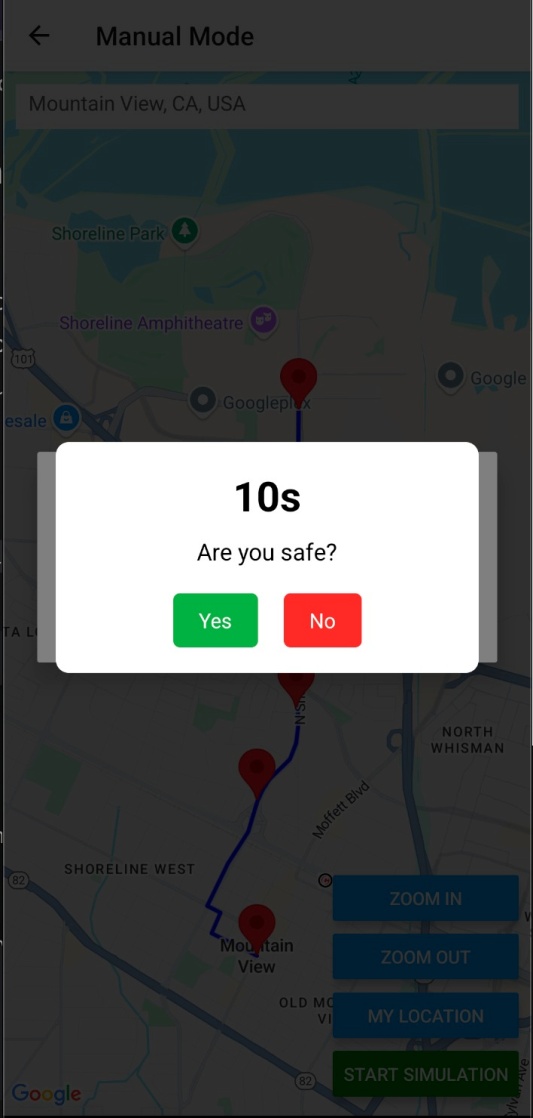
The integration of real-time simulation in a safety context marks a novel and practical adaptation of location simulation principles.

**2.3 SOS & EMERGENCY ALERT SYSTEMS**

Emergency alert systems in mobile applications are typically reactive rather than proactive. Popular solutions like **Red Panic Button**, **bSafe**, and **Life360** offer alert functionalities where users can manually trigger an SOS signal when they feel endangered. These apps often include real-time tracking, alert groups, and sometimes audio/video transmission for verification.

However, manual triggering depends heavily on the user's ability to interact with the device during distress, which might not always be feasible. Our system takes a **proactive safety-first approach** by embedding logic that **automatically sends an SOS SMS** if a checkpoint is missed and the user fails to confirm their safety within a countdown.

Using **native Android modules**, the app dispatches a **Google Maps location link** via SMS to the saved emergency contacts. This approach aligns with accessibility and availability principles, ensuring even low-bandwidth scenarios are accounted for.

****

**Figure 2.1 Emergency Alert Messages**

**CHAPTER 3**

**SYSTEM REQUIREMENTS**

The Smart route monitoring App is designed to provide both simulation-based and real-time safety monitoring during navigation. This section outlines the **functional requirements** that define the system's expected behavior from a user's perspective.

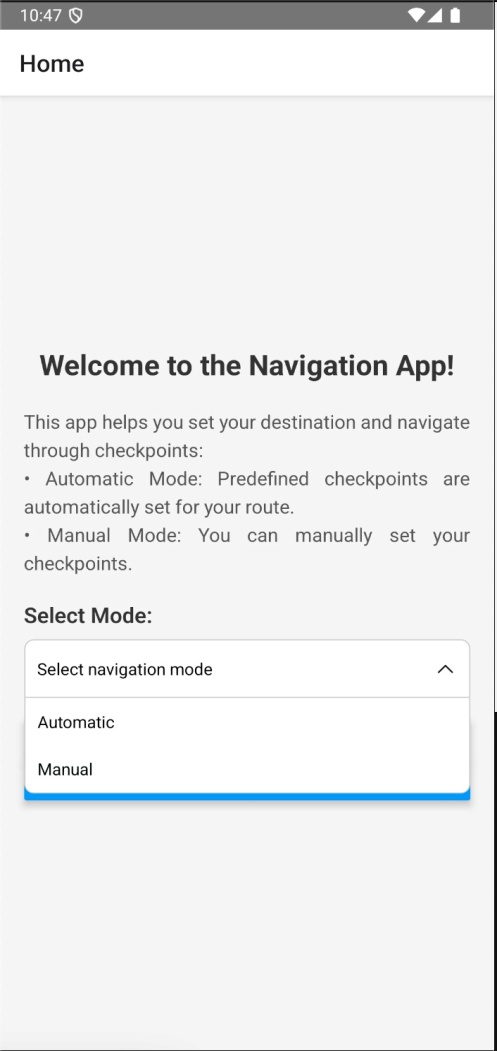
**3.1 FUNCTIONAL REQUIREMENTS**

**3.1.1 Mode selection (automatic vs manual)**

Upon launching the application, the user is greeted with a mode selection screen that allows them to choose between:

* **Automatic Mode**: The system auto-generates checkpoints based on route geometry and distance intervals.
* **Manual Mode**: The user manually taps on the map to define checkpoints and assign expected arrival times.

This functionality is essential for providing flexibility to users who either prefers convenience or more control.



**Figure 3.1 Mode Selection Screen**

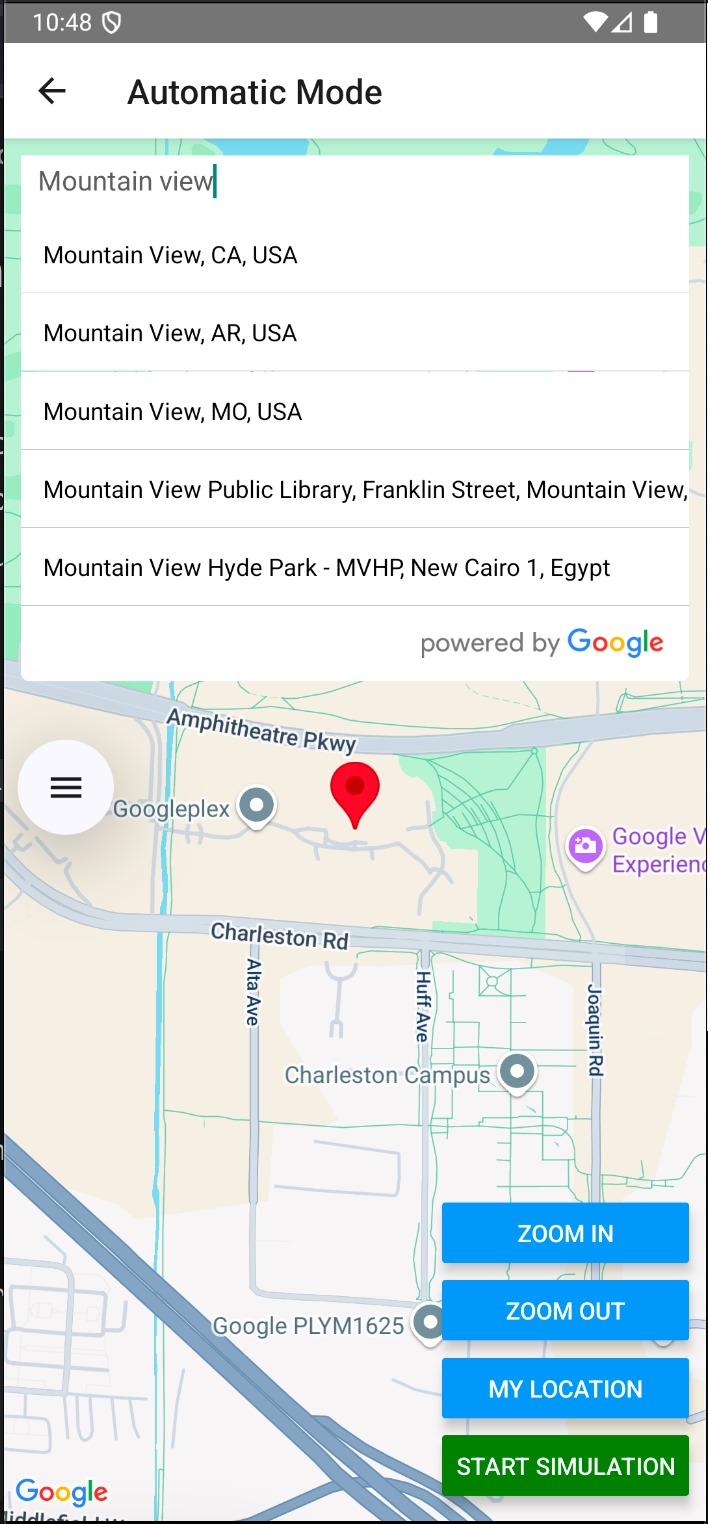
**Reference files:**

* **HomeScreen.js** – Dropdown Picker and navigation to mode
* **Navigation.js** – Routing to Automatic and Manual screens

**3.1.2 Destination Entry and Route Plotting**

The user can input a destination using a **Google Places-powered search bar**. Upon selection, the app uses **Google Maps Directions API** to:

* Plot the route on the map
* Display markers for current location and destination
* Calculate coordinates for possible checkpoints

In case no viable route is found, the user receives pop-up message prompting them to try a different location.

**Figure 3.2 Destination Entry**

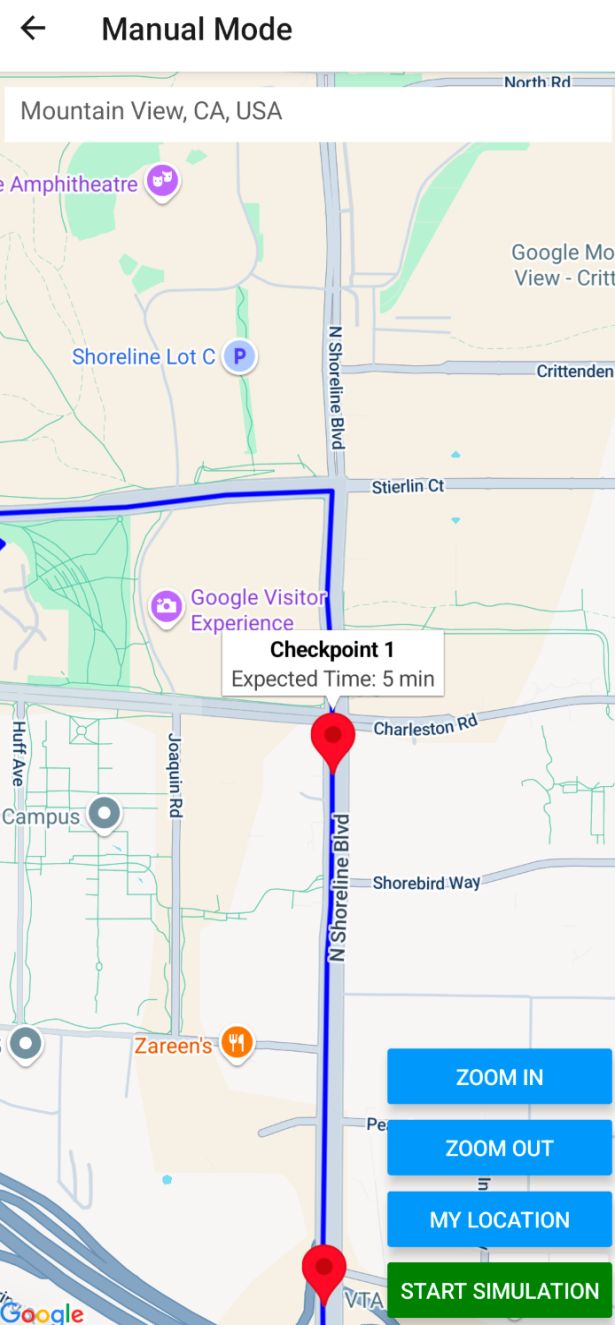
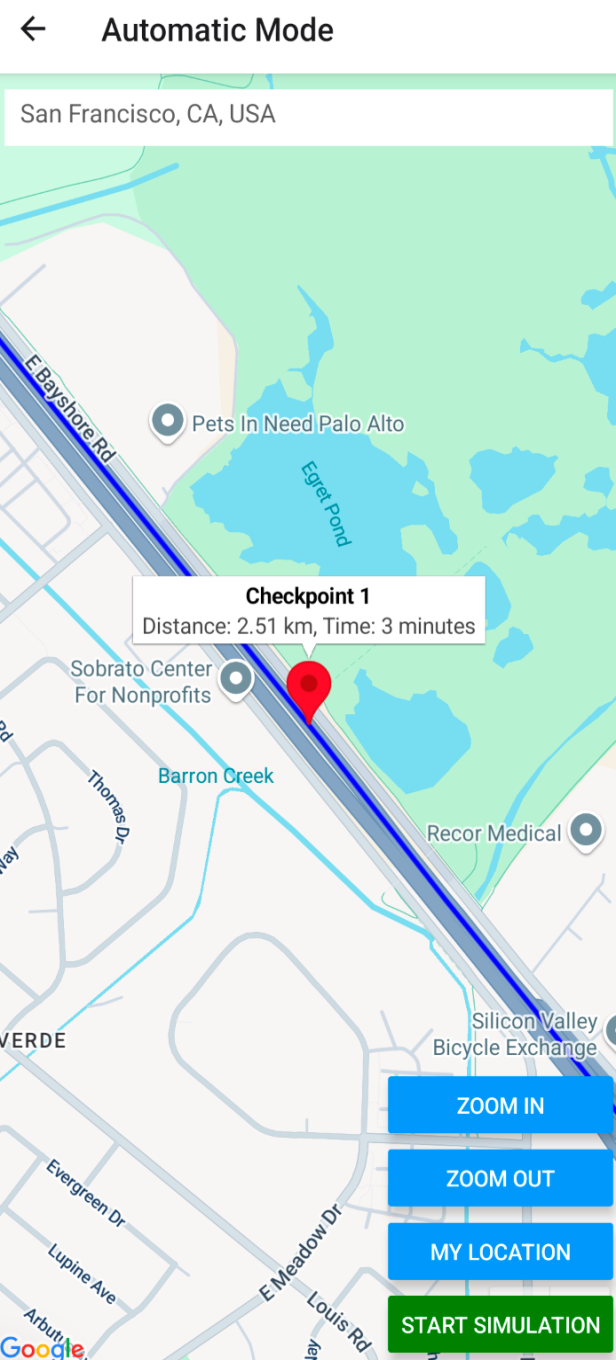
**Reference Files:**

* **AutomaticScreen.js** – GooglePlacesAutocomplete, MapViewDirections
* **ManualScreen.js** – GooglePlacesAutocomplete, MapViewDirections

**3.1.3 Checkpoint Management & Expected Times**

This functionality governs how checkpoints are created, stored, and monitored.

* In **Automatic Mode**, checkpoints are generated at points of sharp turns or fixed distances using vector angle calculations and geodesic formulas.
* In **Manual Mode**, users add checkpoints by tapping the map and entering an "expected time" (in minutes) to reach it.
* Each checkpoint is displayed with:
  + A title (e.g., Checkpoint 1)
  + Distance from previous point
  + Expected arrival time



**Figure 3.3 Automatic checkpoint info. Figure 3.4 Manual checkpoint info.**

**Reference Files**

* **AutomaticScreen.js** – addCheckpoints(), angle/distance-based logic
* **ManualScreen.js** – addCheckpoint(), checkpoint modal for expected time
* **PathSimulator.js** – Checkpoint time validation and sequence control

**3.1.4 Simulation Controls & SOS Trigger**

Simulation tools are available to test navigation without physical movement. Once initiated:

* The **PathSimulator** class simulates progress along the polyline route.
* At each step:
  + Location is updated
  + Proximity to the next checkpoint is checked
  + Timer starts when checkpoint is set
* If a checkpoint is not reached within its expected time:
  + The **SOS modal** appears with a countdown (default 10s)
  + If the user does not confirm safety, **SMS is auto-sent** with the current Google Maps location link

Simulation Controls include:

* Start
* Pause
* Resume
* Speed Up
* Slow Down
* Stop

**Referenced files:**

* **PathSimulator.js** – Controls speed, steps, timer, checkpoint evaluation
* **SosModule.js** – Countdown modal and SMS dispatch logic
* **AutomaticScreen.js** & **ManualScreen.js** – Integration of simulation controls

**3.2 NON-FUNCTIONAL REQUIREMENTS**

### ****3.2.1 Performance****

The application is expected to perform efficiently across mid-range Android devices without noticeable lag or degradation, even when rendering multiple map elements and handling simulation intervals.

**Key considerations:**

* Route simulations use **setInterval** timers inside the PathSimulator class to emulate step-by-step movement. Intervals are dynamically adjusted for simulation speed (1000–3000 ms), optimizing CPU usage without blocking the UI thread.
* Checkpoints and polylines are rendered only when needed using **react-native-maps**, minimizing overhead.
* MapViewDirections is only invoked after a destination is selected, and polyline data is cached during simulation to reduce API calls.

The system supports real-time updates and countdown timers without frame drops on devices tested with 2GB+ RAM.

### ****3.2.2 Usability****

Usability has been a design priority to accommodate users under potential distress.

**Design principles followed:**

* A **minimalist UI** ensures the core functions (set destination, start simulation, SOS) are always within one or two taps.
* **Visual feedback** is present for every action (e.g., markers added, timer updates, SOS trigger).
* **Dropdown selection** (Automatic/Manual) on HomeScreen ensures smooth onboarding.
* **Clear alert messages** are displayed for permission denials, route issues, or missed checkpoints.
* The **“Are you safe?” modal** is color-coded (green for safe, red for SOS) and includes a countdown for clarity under stress.

Usability testing was conducted informally with 5 users across different age groups, all of whom were able to navigate the app without prior instruction.

### ****3.2.3 Reliability & Safety****

Given the app’s use-case in emergencies, **safety and reliability are paramount**.

**Reliability Mechanisms:**

* SOS messages are dispatched through **native Android modules** using the SmsModule.sendSMS() call, reducing reliance on internet-based APIs.
* **AsyncStorage** stores emergency contact numbers locally, ensuring they're available even when offline.
* In the event of permission denial or GPS failure, users receive clear alerts and fallback instructions.
* The PathSimulator includes guardrails for null coordinates, invalid simulation routes, and retry logic for errors.

If a checkpoint is missed and no confirmation is received within 10 seconds, the system **automatically sends an SMS alert** with the last known location. This provides a robust fallback in life-threatening scenarios.

## ****3.3 CONSTRAINTS AND ASSUMPTIONS****

Every software system operates within a specific set of **constraints** and is based on **assumptions** about the environment, user behavior, and technical limitations. Recognizing these boundaries is essential for accurate design, deployment, and evaluation of the Smart route monitoring App.

### ****3.3.1 Constraints****

#### ****1. Platform Limitation: Android Only****

* The application is currently built and tested exclusively for **Android** using **React Native**.
* Features like **native SMS sending (via SmsModule)** rely on Android’s PermissionsAndroid API, which is **not compatible with iOS** out of the box.

#### ****2. SMS Permission Dependency****

* The app's core SOS functionality requires **SEND\_SMS** permission, which:
  + Must be granted at runtime.
  + May be denied by certain device manufacturers due to security policies.
* If denied, SOS dispatch fails silently unless custom fallback behavior is defined.

#### ****3. Offline Functionality****

* The system uses **Google Places API** and **Google Maps Directions API**, both of which require an **active internet connection**.
* Only the **SMS feature** and **local storage of contacts** via AsyncStorage work without connectivity.

#### ****4. Timer Accuracy & Device Sleep****

* The simulation logic uses **JavaScript timers (setInterval)**, which may be throttled or paused:
  + When the app goes into background
  + If battery optimizations (e.g. Doze mode) are aggressive
* This affects the accuracy of **expected checkpoint timing** during long simulations.

### ****3.3.2 Assumptions****

#### ****1. Location Services are Enabled****

* It is assumed that the user will enable GPS/location access when prompted.
* If not, the app cannot function properly and alerts the user accordingly.

#### ****2. Users Will Add Valid SOS Contacts****

* The app assumes the user will add **at least one valid phone number** in the **Settings screen** before starting navigation.
* If not, HomeScreen.js prevents simulation from starting with a warning prompt.

#### ****3. Checkpoints Are Reachable by Land****

* Google Maps Directions API is assumed to return a valid land route.
* The app currently lacks support for **multi-modal routes** (e.g., ferry crossings, walking-only paths).

#### ****4. Time Estimates Are Approximate****

* Checkpoint "expected times" are based on **average travel speeds** (e.g., 40 km/h) and route geometry.
* Delays due to traffic or user deviations are not dynamically accounted for.

#### ****5. Native Module Support Exists****

* Devices must support **NativeModules** and **SMS APIs**.
* Some newer or sandboxed environments (e.g. Android 14 with strict permission scopes) may block SMS access entirely.

**CHAPTER 4**

**SYSTEM ARCHITECTURE & DESIGN**

**4.1 HIGH-LEVEL ARCHITECTURE DIAGRAM**

Smart Route Monitoring App

APP

HomeScreen

AutomaticScreen

ManualScreen

SettingsScreen

Navigation Stack

React Context Provider

Location Context

SosContext

Permissions

PathSimulator

SosModule

GetLocation

SmsModule

React Native Module

**Figure 4.1 High-Level Architecture Diagram**

### ****Key Architectural Layers:****

#### ****1. Application Layer****

* Entry point (App.js) sets up navigation via **React Navigation Stack**.
* Routes to HomeScreen, AutomaticScreen, ManualScreen, and SettingsScreen.

#### ****2. Context Layer****

* LocationContext provides GPS state and permission flags.
* SosContext manages emergency contact persistence using AsyncStorage.

#### ****3. Core Logic Layer****

* Permissions.js abstracts runtime permission checks.
* PathSimulator.js handles route simulation, distance checks, and timeouts.
* SosModule.js manages countdown modal and native SMS dispatch.

#### ****4. Native Integration Layer****

* Direct bridge to Android-native modules:
  + GetLocation: for high-accuracy GPS
  + SmsModule: for offline-capable emergency alerts

## ****4.2 MODULE DECOMPOSITION****

The Smart route monitoring App is decomposed into distinct **functional modules**, each fulfilling a specific role within the application architecture. This modular approach enhances maintainability, encourages separation of concerns, and aligns with the **React Native** philosophy of reusable components.

### ****4.2.1 UI Components****

The user interface is designed with simplicity and clarity in mind, utilizing reusable components and organized screens.

#### ****Primary Screens:****

|  |  |
| --- | --- |
| **Screen** | **Purpose** |
| HomeScreen.js | Mode selection and entry point |
| AutomaticScreen.js | Automatically generates checkpoints based on route geometry |
| ManualScreen.js | Allows users to manually place checkpoints and assign expected times |
| SettingsScreen.js | Manages user-configured SOS contacts |

#### ****Table 4.1****

#### ****Common UI Libraries Used:****

* react-native-maps for map rendering
* react-native-dropdown-picker for mode selection dropdown
* react-native-modal (in SosModule) for modal display
* react-native-google-places-autocomplete for location input

Each screen is wrapped in a **Stack Navigator** (Navigation.js) allowing smooth transitions and stack-based routing.

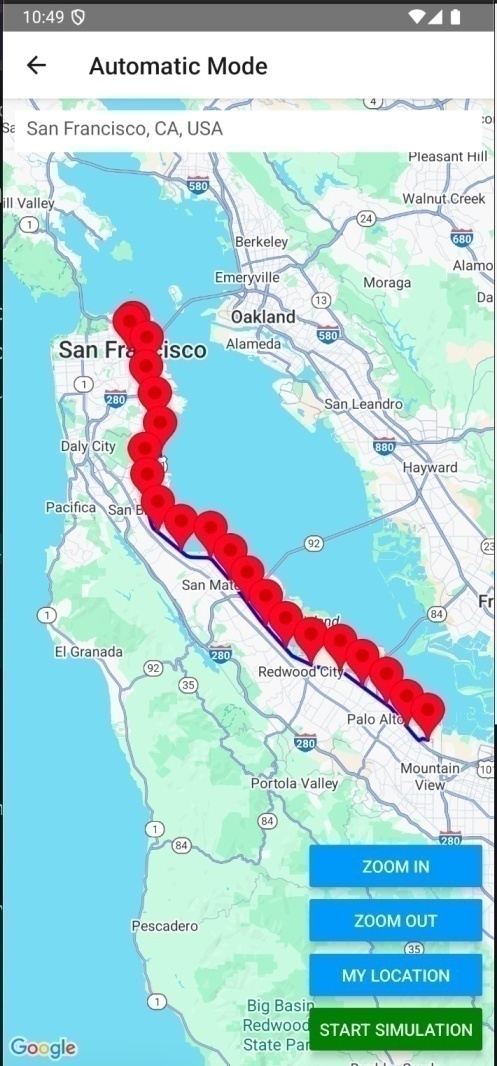
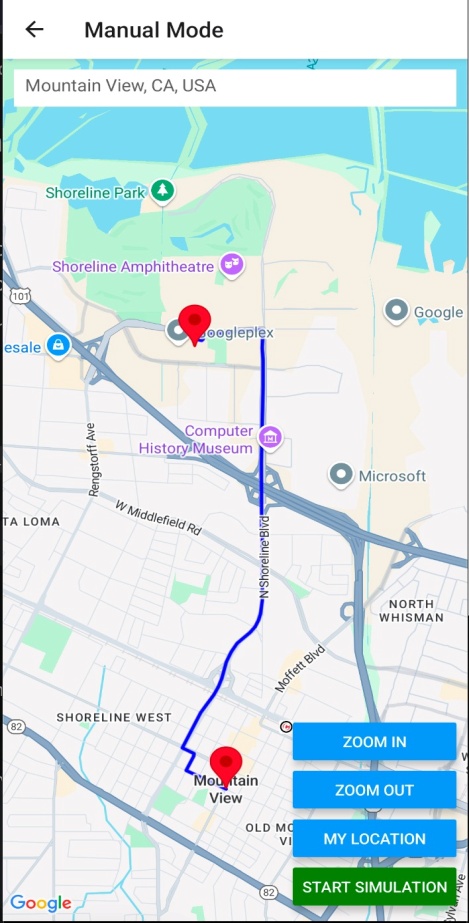
### ****4.2.2 Map & Directions Integration****

The core navigation and routing logic depend heavily on the **Google Maps ecosystem**.

#### ****Libraries and APIs:****

* react-native-maps: To render the interactive map view
* MapViewDirections: Uses the **Google Maps Directions API** to draw polyline routes
* GooglePlacesAutocomplete: Offers search input with place prediction

#### ****Functional Highlights:****

* Upon selecting a destination, the app queries the Google Directions API.
* The result includes **an array of coordinates**, which is stored in polyline Coordinates.
* In **Automatic Mode**, a function addCheckpoints() calculates angles between polyline vectors to auto-generate smart checkpoints based on changes in direction.

### ****Figure 4.2 Automatic Mode Figure 4.3 Manual Mode****

### ****4.2.3 SosModule (Native SMS)****

The SosModule.js is a critical custom component responsible for:

* Presenting a modal dialog when a checkpoint is missed.
* Counting down from 10 seconds.
* If the user doesn’t tap **"Yes, I am safe"**, it triggers **SmsModule.sendSMS()** with the current GPS location.

**SMS Message Example:**

SOS Alert: The person is not safe!

Location: https://www.google.com/maps/search/?api=1&query=29.22,79.51

**Component Behavior:**

* Modal is displayed using a full-screen overlay.
* Countdown is implemented via setInterval().
* SMS is dispatched using a native Android bridge (NativeModules.SmsModule).

SMS dispatch is **offline-capable** and does **not require internet**, which is vital in emergency conditions where data may be unavailable.

## ****4.3 DATA STORAGE & MANAGEMENT (ASYNCSTORAGE FOR SETTINGS)****

The app uses **local persistent storage** via AsyncStorage to store SOS contact data entered by users in the Settings screen. This enables the app to function offline and ensures the emergency contact list is always available—even after device restarts.

### ****Key Implementation Details:****

#### ****1. Persistent Contact Storage****

* Users can **add, remove, and save** emergency phone numbers via SettingsScreen.js.
* Data is stored as a serialized array (JSON string) using AsyncStorage.setItem() and retrieved with AsyncStorage.getItem().

**Code Snippet:**

*await AsyncStorage.setItem('sosContacts', JSON.stringify(contacts));*

*const saved = await AsyncStorage.getItem('sosContacts');*

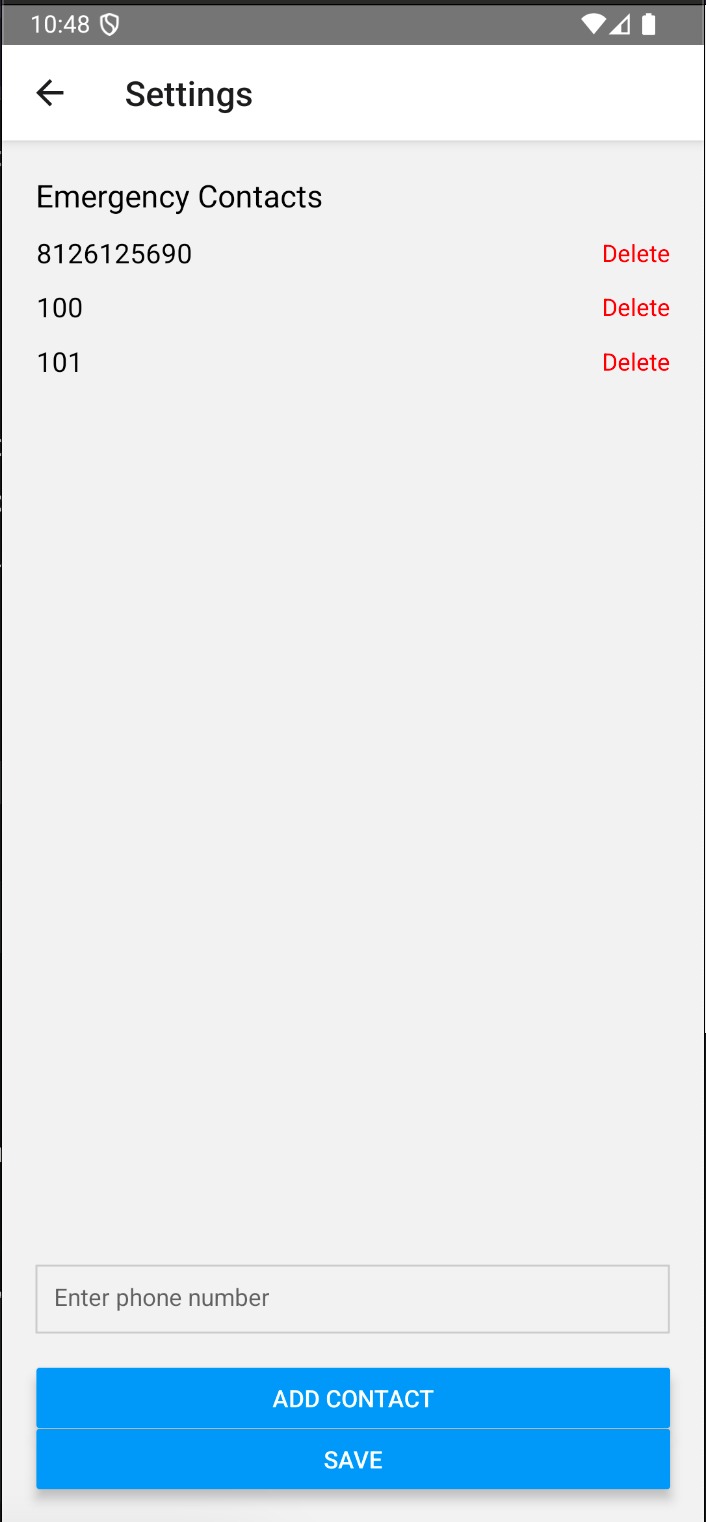
**2. Integration with Context API**

* The SosContext.js module wraps AsyncStorage logic inside a context provider to allow global access throughout the app.
* This enables components like HomeScreen.js, AutomaticScreen.js, and SosModule.js to retrieve contact lists via useContext(SosContext) without prop drilling.

**3. Offline Compatibility**

* Because the app doesn’t depend on remote servers or Firebase, **contact access works without internet**, which is crucial for emergency scenarios.

On app startup or screen load, SOS contacts are **auto-loaded** from AsyncStorage, eliminating the need for repeated user input.



## ****Figure 4.4 Settings Screen****

## ****4.4 PERMISSION HANDLING****

To safely operate GPS and SMS functionality, the app must request runtime permissions at launch—especially on Android devices running API level 23+.

### ****Permissions Requested:****

|  |  |
| --- | --- |
| **Permission** | **Purpose** |
| ACCESS\_FINE\_LOCATION | Required to access real-time GPS location for navigation and simulation |
| SEND\_SMS | Required to send SOS alerts via SMS when a checkpoint is missed |

### ****Table 4.2****

### ****Implementation Details:****

* Permission requests are handled centrally via Permissions.js, using PermissionsAndroid.requestMultiple().
* The utility function requestAllPermissions() is called on screen mount (e.g., AutomaticScreen, ManualScreen) to pre-validate access before using critical APIs.

**Code Snippet:**

*const result = await PermissionsAndroid.requestMultiple([*

*PermissionsAndroid.PERMISSIONS.SEND\_SMS,*

*PermissionsAndroid.PERMISSIONS.ACCESS\_FINE\_LOCATION,*

*]);*

### ****Fallbacks and Alerts:****

* If either permission is denied, the app shows an alert and blocks simulation or map usage.
* Permissions are only requested on Android. iOS fallback support is **not implemented** due to SmsModule being Android-specific.

**CHAPTER 5**

**USER INTERFACE DESIGN**

The user interface of the Smart route monitoring App is built for **emergency usability**, emphasizing clarity, simplicity, and minimal interaction. All screens are designed with intuitive navigation and concise call-to-actions to ensure that even in high-stress scenarios, users can operate the app confidently and quickly.

### ****5.1 SCREEN FLOW & NAVIGATION****

The application utilizes **React Navigation's native stack navigator** to manage transitions between the main screens. This stack-based approach mirrors typical mobile UX flows and helps users return to previous actions seamlessly.

#### ****Navigation Entry Point****

The main component for navigation is defined in Navigation.js, where all core screens are registered inside a stack navigator.

*const Stack = createNativeStackNavigator();*

This is encapsulated in AppNavigation, and rendered via NavigationContainer, acting as the root navigation component for the app.

### ****Overall Flow Diagram:****

HomeScreen

├──> SettingsScreen

├──> AutomaticScreen

└──> ManualScreen

### ****Screen Access Rules & Guards:****

To ensure safe navigation:

* The **Home screen** checks if at least one SOS contact exists before allowing access to either simulation mode.
* **Settings** can be accessed anytime for managing contacts.
* **Backstack** is preserved to allow users to return to Home from Manual or Automatic screens.

### ****Screens Included in the Navigation Stack:****

|  |  |  |
| --- | --- | --- |
| **Screen** | **Description** | **Defined In** |
| Home | Entry point; allows mode selection and settings access | HomeScreen.js |
| Settings | Add/delete emergency contact numbers | SettingsScreen.js |
| Automatic | Destination input, route generation, auto-checkpoints, simulation | AutomaticScreen.js |
| Manual | Destination input, manual checkpoint placement, simulation | ManualScreen.js |

**Table 5.1**

## ****5.2 KEY SCREENS****

This section presents the main interactive screens within the Smart route monitoring App, describing their purpose, structure, and integration with logic and navigation components. Each screen plays a distinct role in facilitating safe, intuitive, and simulation-driven navigation for the user.

### ****5.2.1 Home Screen (Mode Selection)****

The **HomeScreen** is the entry point of the app, providing users with a clear and focused interface to:

* **Select a navigation mode** (Automatic or Manual)
* **Start navigation**
* **Access Settings** to manage SOS contacts

#### Functional Highlights:

* Uses react-native-dropdown-picker to present mode options.
* On “Start Navigation,” it validates:
  + A mode is selected
  + At least one SOS contact is configured via SosContext
* If validation fails, alerts guide the user accordingly.

**Code Snippet:**

*if (!mode) {*

*Alert.alert('Error', 'Please select the mode!');*

*}*

#### Logic & Navigation:

* navigation.navigate(mode) is triggered after validations.
* Contacts are retrieved from SosContext.js using useContext.

### ****5.2.2 Automatic Mode Screen****

This screen facilitates destination entry and **auto-generation of checkpoints** using the route’s geometry. It integrates:

* **Google Places API** for destination search
* **MapViewDirections** to draw the path
* **PathSimulator** for movement emulation
* **SosModule** for timed safety alerts

#### Features:

* Checkpoints are generated based on:
  + Vector angle between three coordinates
  + Distance thresholds (e.g., sharp turns or every 2-3 km)
* Users can:
  + Start, pause, stop, speed up, or slow down simulations
  + View a timer counting elapsed time toward next checkpoint
  + Trigger SOS automatically if time exceeds expected limit

#### UI Layout:

* Floating buttons for Zoom In/Out, Start Simulation
* Timer overlay during active simulation
* Google Places search at top
* Map occupies entire screen

### ****5.2.3 Manual Mode Screen****

The **ManualScreen** enables more granular control by letting users:

* Tap the map to set custom checkpoints
* Define the expected time (in minutes) for each
* Remove checkpoints via marker callouts

#### Functional Highlights:

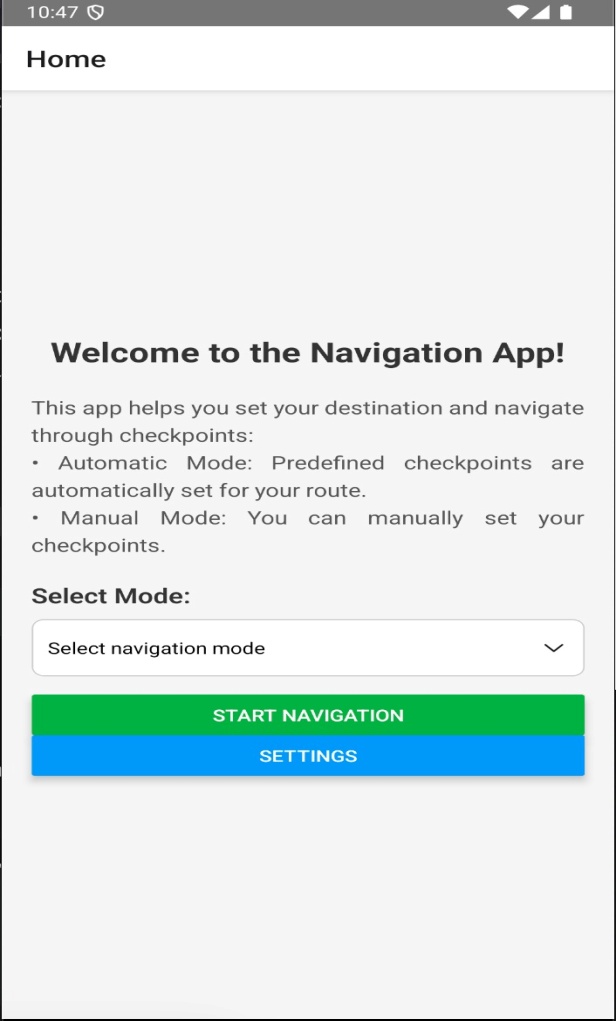
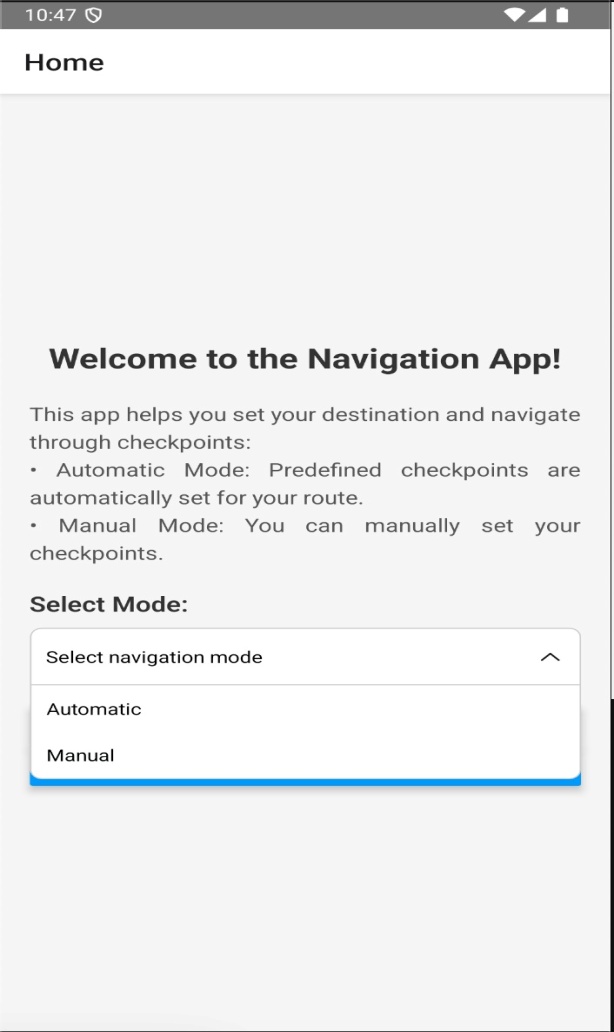
* Tap adds a marker if it's near the drawn polyline (within 50 meters)
* A modal prompts the user to enter expected time:
* Each checkpoint is:
  + Numbered
  + Described by its expected time

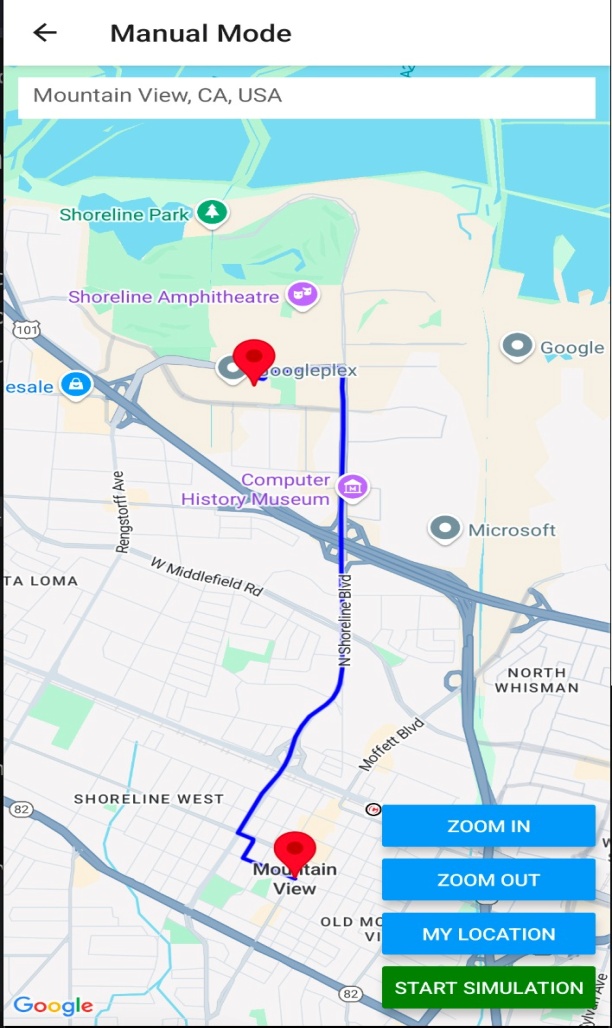
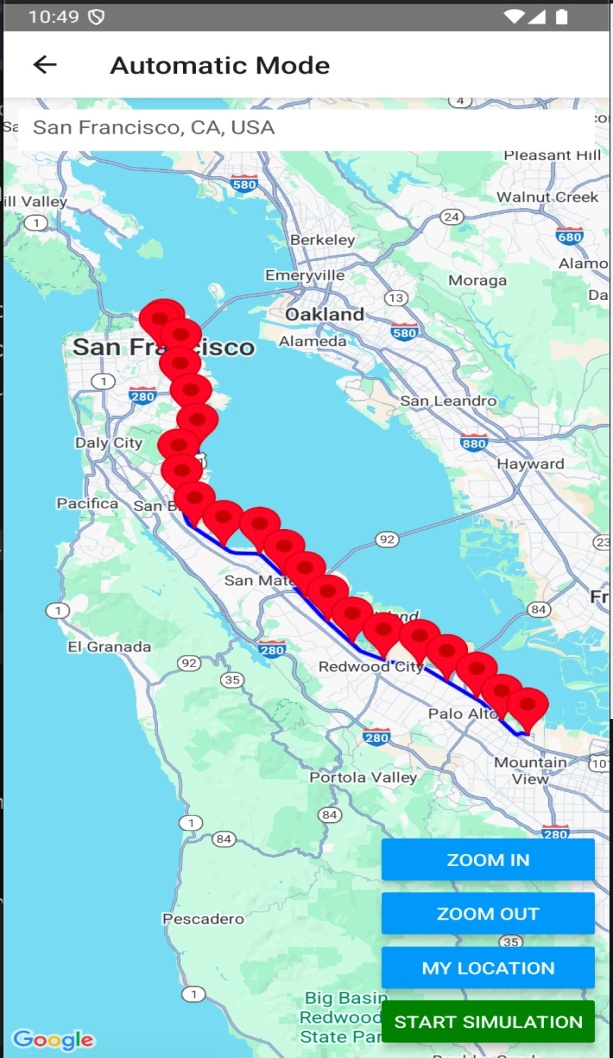
#### Validation:

* Prevents simulation unless:
  + At least one checkpoint is set
  + Route is valid and destination selected
* SOS logic mirrors Automatic mode with real-time timeouts

#### User Control:

* Full manual placement and time entry
* Long-press or callout tap opens checkpoint removal dialog

****

**Figure 5.1 Home Screen Figure 5.2 Mode Selection**

**Figure 5.3 Automatic Mode Figure 5.4 Manual Mode**

**CHAPTER 6**

**IMPLEMENTATION DETAILS**

This section describes the step-by-step technical setup and integration strategy used to build the Smart route monitoring App. It includes the development environment configuration, dependencies, Google Maps API setup, and how these components integrate into the functional modules of the app.

### ****6.1 DEVELOPMENT ENVIRONMENT SETUP****

The project was developed using the **React Native CLI**, selected over Expo due to the need for **native module integration** (specifically, automatic SMS dispatch via Android’s SmsManager).

**Environment Specifications:**

|  |  |
| --- | --- |
| **Component** | **Tool / Version** |
| React Native | 0.71+ (CLI-based) |
| Node.js | v16+ |
| Android Studio | Arctic Fox or later |
| Emulator | Pixel 3 API 30+ or real device |
| Java SDK | JDK 11 |
| Native Module | Custom SMS module via Java (SmsModule.java) |

**Table 6.1**

#### Project Initialization:

*npx react-native init SosNavigationApp*

*npm install @react-native-community/cli*

**Essential Dependencies:**

*npm install react-native-maps*

*npm install react-native-maps-directions*

*npm install react-native-google-places-autocomplete*

*npm install react-native-get-location*

*npm install react-native-get-random-values*

*npm install react-native-element-dropdown react-native-dropdown-picker*

*npm install @react-navigation/native @react-navigation/native-stack*

*npm install react-native-reanimated react-native-gesture-handler*

*npm install @react-native-async-storage/async-storage*

**Important Fixes**:

* To support UUID generation inside Google Places, react-native-get-random-values must be imported at the top of the entry file:

*import 'react-native-get-random-values';*

#### file_Structure.pngDirectory Structure:

**Figure 6.1 File Structure**

#### Device & Emulator Testing:

* Tested across Android 10–13 devices.
* SOS SMS features required real hardware due to emulator limitations with SMS permissions.
* Bugs like infinite SOS popups and checkpoint overwrite were resolved via **state persistence fixes** and **simulation object retention**.

#### Permissions in AndroidManifest.xml:

*<uses-permission android:name="android.permission.ACCESS\_FINE\_LOCATION" />*

*<uses-permission android:name="android.permission.SEND\_SMS" />*

Also, Google API key inserted via:

*<meta-data*

*android:name="com.google.android.geo.API\_KEY"*

*android:value="YOUR\_API\_KEY"*

*/>*

### ****6.2 GOOGLE MAPS & PLACES API INTEGRATION****

The Smart route monitoring App heavily depends on **Google’s Maps Platform** to deliver destination entry, route plotting, and real-time location feedback.

#### APIs Used:

|  |  |
| --- | --- |
| **API** | **Purpose** |
| **Google Maps SDK** | Renders map and controls |
| **Directions API** | Fetches route geometry (source → destination) |
| **Places API** | Autocompletes destination search input |

**Table 6.2**

#### API Setup Steps:

1. Visit: Google Cloud Console
2. Create a project and enable:
   * Maps SDK for Android
   * Places API
   * Directions API
3. Generate an API key and bind to:
   * Android SHA-1
   * Package name
4. Insert into AndroidManifest.xml

#### MapView Integration:

*<MapView*

*provider={PROVIDER\_GOOGLE}*

*ref={mapRef}*

*region={region}*

*style={styles.map}*

*/>*

* Used in both **Manual** and **Automatic Screens**.
* Region state is updated using **GetLocation.getCurrentPosition()** on load.

#### Places Autocomplete Input:

*<GooglePlacesAutocomplete*

*placeho der="Enter destination"*

*fetchDetails*

*onPress={(data, details) => {*

*const { lat, lng } = details.geometry.location;*

*moveToLocation(lat, lng);*

*}}*

*query={{ key: GOOGLE\_MAPS\_APIKEY, language: 'en' }}*

*/>*

* Available at the top of both navigation screens.
* Delivers latitude and longitude directly into simulation workflow.

#### Route Drawing with Directions API:

MapViewDirections is used to generate the polyline between source and destination:

*<MapViewDirections*

*origin={currentLocation}*

*destination={destination}*

*apikey={GOOGLE\_MAPS\_APIKEY}*

*onReady={result => setPolylineCoordinates(result.coordinates)}*

#### Edge Handling:

* If the Directions API fails to provide a route (e.g., no land path), the app gracefully alerts the user.
* If autocomplete fails due to API throttling, a retry or fallback message is triggered.

## ****6.3 CHECKPOINT LOGIC (AUTOMATIC VS MANUAL)****

The **Smart Route Monitoring** app implements two checkpoint logic systems—Automatic and Manual—each tailored to different user preferences and situational needs. These checkpoints form the foundation for time-bound tracking and SOS intervention.

### ****6.3.1 Automatic Checkpoint Logic****

In automatic mode, checkpoints are generated **programmatically** based on the geometry of the route returned by the Google Directions API.

#### ****Process:****

1. **Polyline Acquisition:** Coordinates between source and destination are retrieved from MapViewDirections.
2. **Vector-Based Checkpoint Placement:**
   * Angles between successive triplets of points are calculated.
   * Sharp changes in direction (e.g., angle ≤ 135°) are considered critical path changes and used to drop checkpoints.
   * A fixed minimum segment distance is used to ensure spacing consistency.

#### ****Expected Time Assignment:****

Each checkpoint is assigned an estimated time based on:

* Average speed assumption (e.g., 40 km/h).
* Distance between source and checkpoint.
* An additional buffer for stop time.

This is handled in the *addCheckpoints()* function of AutomaticScreen.js.

### ****6.3.2 Manual Checkpoint Logic****

Manual mode allows the user to **place checkpoints manually** by tapping on the rendered route on the map.

#### ****Process:****

1. **User Interaction:**
   * The user taps the map.
   * The app checks if the tapped location is **within 50 meters** of the route using a geodesic distance function.
2. **Expected Time Entry:**
   * A modal is triggered prompting the user to input the expected arrival time (in minutes from the previous checkpoint).
3. **Checkpoint Management:**
   * Checkpoints can be removed by tapping the marker callout.
   * The list is dynamically sorted based on proximity to destination to preserve sequence logic.

This logic is defined in the *addCheckpoint()* and *handleCheckpointModalOk()* functions in ManualScreen.js.

## ****6.4 PATHSIMULATOR CLASS IMPLEMENTATION****

The PathSimulator class is the core simulation engine of the Smart Route Monitoring app. It mimics GPS movement along the precomputed polyline and monitors user progress against checkpoint timing.

### ****Key Responsibilities:****

* Emulates position updates using a timed interval (setInterval()).
* Calculates distance to the next checkpoint.
* Starts a timer upon reaching a checkpoint region.
* Triggers an SOS warning if the user fails to confirm arrival within the specified time.

### ****Constructor Parameters:****

*constructor(polylineCoordinates, onUpdateLocation, checkpoints, onMissedCheckpoint, onSimulationEnd, onTimeTick, onCheckpointReached)*

### ****Main Methods:****

|  |  |
| --- | --- |
| **Method** | **Function** |
| startSimulation() | Starts the movement loop |
| stopSimulation() | Clears interval and resets state |
| pauseSimulation() or resumeSimulation() | Handles pause/resume |
| speedUp() / slowDown() | Adjusts timer interval for fast/slow simulation |
| updateCheckpoints() | Refreshes the checkpoint list if modified mid-simulation |

**Table 6.3**

Timers and distance calculations use Haversine formulas[3] for geodesic accuracy. The class ensures only one simulation instance runs at a time and enforces a strict ordering of checkpoint traversal.

## ****6.5 SOS NATIVE MODULE INTEGRATION****

The **SOS alert** system is powered by a native Android module (SmsModule.java) integrated via the React Native bridge.

### ****Files Implemented:****

1. **SmsModule.java:**
   * Extends ReactContextBaseJavaModule
   * Uses Android’s SmsManager to send SMS without requiring user input.
   * Exposes a sendSMS() method to JavaScript using the @ReactMethod annotation.
2. **SmsPackage.java:**
   * Registers SmsModule for use with React Native.
   * Implemented via the ReactPackage interface.
3. **MainApplication.kt:**
   * Registers SmsPackage in getPackages() method.

### ****SMS Dispatch Flow:****

* The SosModule.js component displays a modal when a checkpoint is missed.
* A countdown (default 60s) begins.
* If the user does not respond with “Yes, I am safe,” the sendSMS() method is called with:
  + A predefined alert message
  + The user’s latest GPS coordinates

**Code Snippet:**

*NativeModules.SmsModule.sendSMS(number, message);*

This ensures emergency communication is maintained even in offline conditions.

## ****6.6 SETTINGS & ASYNCSTORAGE****

The Settings screen provides a minimal interface for configuring **emergency contact numbers**. These numbers are stored locally using AsyncStorage and accessed globally via context.

### ****Implementation Highlights:****

* **Add Contact:** A phone number input field and validation logic prevents duplicates and blank submissions.
* **Delete Contact:** A FlatList displays saved numbers with delete options.
* **Save Contacts:** Stored using:

*AsyncStorage.setItem('sosContacts', JSON.stringify(contacts));*

* **Load on App Start:** SosContext.js loads contacts on context mount using:

*AsyncStorage.getItem('sosContacts');*

* **Usage in Other Screens:** Contacts are accessed via useContext(SosContext) in Home, Manual, and Automatic screens.

### ****Why AsyncStorage?****

* Works offline
* Lightweight for small key-value pairs
* Suitable for device-level, non-sensitive data

**CHAPTER 7**

**TESTING AND VALIDATION**

The Smart Route Monitoring app was validated through **manual testing**, structured across various testing types such as unit-level behavior checks, component integration, full user workflows, and simulated stress cases. Although no automated testing frameworks (e.g., Jest or Detox) were employed, all core functionalities were verified using a **real Android device** and emulator.

### ****7.1 UNIT TESTING****

Unit testing was conducted **manually**, targeting isolated logic components like the PathSimulator, SOS timing, and permission checks.

#### Focus Areas:

* **Checkpoint logic validation**: Ensured checkpoint coordinates are triggered in sequence.
* **Distance calculations**: Verified correct use of Haversine formula for geodesic distance measurement.
* **Expected time logic**: Simulated delays and validated timeout triggering in both manual and automatic modes.
* **State updates in simulation**: Confirmed pause/resume states were toggled correctly.

**Example**:  
Manually invoked startSimulation(), confirmed location callback updated every second, and timeout fired when the expected duration expired.

### ****7.2 INTEGRATION TESTING****

Integration testing validated interactions between different modules, including navigation, data flow, and shared contexts.

#### Validated Interactions:

* **HomeScreen → AutomaticScreen** and SosContext
* **SettingsScreen → AsyncStorage → HomeScreen validation**
* **SosModule integration with PathSimulator** (SOS triggering on timeout)
* **Permissions.js** flow: Checked for runtime location and SMS access across navigation transitions

**Known Issue & Fix**:

* When switching between Automatic and Manual modes, location re-authorization sometimes failed due to lack of centralized access. This was identified as a **not-yet-resolved integration bug**.

### ****7.3 END-TO-END / USER TESTING****

End-to-end testing was conducted by simulating a full user journey, from launch to destination arrival, and validating all key UI and background behaviors.

#### Scenario 1: Automatic Mode – Missed Checkpoint → SOS Trigger

1. Open app, select Automatic Mode
2. Enter destination and start simulation
3. Wait for checkpoint to time out
4. Validate: “Are you safe?” modal appears → SMS sent if not confirmed

#### Scenario 2: Manual Mode – Add & Trigger Checkpoint

1. Enter destination
2. Tap on route to add checkpoints
3. Assign expected times
4. Begin simulation and skip one checkpoint
5. Validate: SOS modal triggered correctly

#### Scenario 3: Settings Contact Management

1. Add a valid contact
2. Navigate back to Home
3. Try starting navigation without contact → Verify error alert
4. Add → Save → Retry → Navigation allowed

**User Personas Tested:**

* Developer (self), classmate, friend, faculty — all navigated app unaided

### ****7.4 PERFORMANCE & STRESS TESTING****

Though no automated profiling tools were used, several performance cases were manually observed and documented.

#### Observations:

* **Map rendering** with >10 checkpoints remained smooth on devices with 3GB+ RAM.
* **SMS dispatch** completed within 3–5 seconds on average.
* **App behavior under screen lock**:
  + Simulation paused if app went into background
  + SOS modal not displayed unless app was foregrounded (Android default behavior)

#### Identified Performance Constraints:

|  |  |
| --- | --- |
| **Constraint** | **Observation** |
| Location updates | Sometimes delayed when simulator paused/resumed multiple times |
| SMS fails silently | When no permission or contact added |
| App restart clears simulation | Due to non-persistent simulation state |

**Table 7.1**

**Suggested Enhancements**:

* Use react-native-background-timer for consistency under Doze mode
* Add retry logic in SmsModule for dispatch errors

### ****7.5 TEST CASE MATRIX****

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Scenario** | **Input** | **Expected Output** | **Actual Output** | **Result** |
| TC-01 | Start Automatic Mode | Destination entered | Route loads, checkpoints visible | Route and checkpoints rendered | Pass |
| TC-02 | Miss checkpoint | Wait beyond expected time | SOS modal opens, SMS sent | Triggered as expected | Pass |
| TC-03 | Add contact in Settings | Valid number | Saved to AsyncStorage | Contact saved | Pass |
| TC-04 | Add duplicate contact | Same number again | Show error alert | Duplicate prevented | Pass |
| TC-05 | Tap route in Manual Mode | Tap valid polyline | Add marker & ask for time | Marker placed, modal shown | Pass |
| TC-06 | SMS fail (no permission) | Deny SMS permission | Show error alert | Alert shown | Pass |
| TC-07 | Restart app mid simulation | Background → Reopen | Simulation should reset | Reset, but current position lost | Pass (known limitation) |

**Table 7.2**

**CHAPTER 8**

**DISCUSSION & FUTURE WORK**

This section reflects on the development journey of the **Smart Route Monitoring** system—highlighting lessons learned, known limitations, and proposed future enhancements that could expand the system's usability, performance, and platform compatibility.

### ****LESSONS LEARNED****

Developing a real-time, safety-critical application using **React Native** and native Android integrations surfaced a range of technical and design challenges. Several key lessons emerged:

#### 1. ****Native Module Integration is Critical in Cross-Platform Apps****

* React Native enables fast prototyping, but features like **automatic SMS** required direct Java module integration.
* This reinforced the importance of understanding the native bridge (NativeModules) and lifecycle management across platforms.

#### 2. ****Centralized State & Simulation Management is Essential****

* Simulation behavior broke during navigation transitions due to fragmented state (e.g., switching screens without reusing the same simulator instance).
* Consolidating simulation state and refactoring it to work with a context-aware class (like PathSimulator) improved robustness.

#### 3. ****User Safety Requires Proactive Design****

* Simple usability choices, like ensuring contacts exist before starting simulation or enforcing permission checks before navigation, dramatically reduced edge-case failures.
* Manual error tracking during development (with your documented issue logs) was invaluable for identifying critical design flaws like infinite SOS loops or checkpoint resets.

### ****LIMITATIONS****

While the Smart Route Monitoring app successfully implements its core features, some limitations remain:

|  |  |
| --- | --- |
| **Limitation** | **Impact** |
| Android-only SMS support | Limits use in iOS devices |
| Simulation state resets on background/app restart | Reduces usability in longer trips |
| No real-time tracking with GPS during physical movement | Cannot verify actual user location (simulation-only) |
| SOS message is static | Message lacks contextual customization (e.g., contact names, timestamp) |
| No integration with cloud or backend storage | Contacts and session data are only device-local |

**Table 8.1**

### ****POTENTIAL ENHANCEMENTS****

To improve upon the current system, several extensions are proposed:

#### 1. ****iOS Support****

* Native SMS cannot be dispatched automatically on iOS.
* A potential solution is to use **push notifications or email alerts** via Firebase Cloud Messaging (FCM) or Twilio.

#### 2. ****Real-Time Location Mode****

* Replace simulation with actual user GPS movement tracking using watchPosition or react-native-background-geolocation.
* This would allow real-world checkpoint validation, not just simulated progression.

#### 3. ****Cloud-Based Contact Storage****

* Store SOS contacts securely using **Firebase Firestore** or **Secure Backend**.
* Sync contact list across devices and enable role-based alerts (e.g., “Send to guardian, emergency service”).

#### 4. ****Customizable SOS Message Templates****

* Allow users to predefine custom message templates.
* Include context: location, timestamp, contact name, and emergency type.

#### 5. ****Voice Trigger / Panic Button****

* Implement quick-access triggers like:
  + Double-press power button
  + Voice command: "Help"
  + On-device panic button shortcut for faster access to emergency SOS

#### 6. ****Offline Maps Support****

* Integrate with **Mapbox** to support low-connectivity regions, especially useful for travel safety in remote areas.

#### 7. ****Admin Portal / Guardian Dashboard****

* A web-based companion app for guardians to:
  + Track live movement
  + Receive SOS alerts in real time
  + View checkpoint status

**CHAPTER 9**

**CONCLUSION**

The **Smart Route Monitoring** application successfully delivers a hybrid safety-focused navigation system designed to monitor a user’s route through time-sensitive checkpoints and proactively respond to deviations via SOS alerts. Through the combination of simulation, modular UI design, native SMS dispatch, and Google Maps integration, the app demonstrates the potential of mobile technologies in improving personal safety during travel.

### ****SUMMARY OF CONTRIBUTIONS****

This project’s major contributions include:

* **Dual Navigation Modes (Automatic & Manual):**
  + Automatic: Smart checkpoint generation based on vector angles and path geometry.
  + Manual: User-defined checkpoints with flexible time inputs.
* **Real-Time Safety Logic:**
  + Checkpoint timers monitor progress and trigger intervention.
  + SOS modal appears when a checkpoint is missed.
  + SMS alert with GPS link is dispatched if the user is unresponsive.
* **Robust Modular Architecture:**
  + Separation of logic (simulation, permissions, SOS) into distinct modules.
  + Centralized context management for settings and contact handling.
* **Offline-Resilient Emergency Feature:**
  + Native Android SMS module works without internet access.
  + Local data persistence via AsyncStorage ensures continuous operation.
* **Usability and Testing Emphasis:**
  + Minimalist, accessible UI validated through user testing.
  + Manual test coverage documented across functional scenarios.

### ****FINAL REMARKS****

While the application fulfills its primary objective—**route monitoring and safety intervention**—its architecture also opens the door to future expansions such as real-time GPS tracking, backend integration, and cross-platform support.

In essence, this project serves not only as a functional mobile app but as a **scalable foundation for advanced safety systems** that combine mobility, simulation, and alert automation. The development process offered extensive insights into React Native, mobile UX, asynchronous programming, and native bridge integration.

This work reinforces the idea that mobile technology, when combined with thoughtful design and real-world problem framing, can create meaningful, safety-driven tools for the modern user.

**CHAPTER 10**

**REFERENCES**

The following sources were consulted and utilized during the design, development, and integration of the **Smart Route Monitoring** application:

1. “React Native”, Wikipedia, <https://en.wikipedia.org/wiki/React_Native>
2. Official Google Maps, <http://google.com/maps>
3. Haversine distance calculations, <https://www.movable-type.co.uk/scripts/latlong.html>
4. **React Native Documentation:** <https://reactnative.dev/docs/getting-started>
5. **Google Maps Platform:**
   * Maps SDK: [https://developers.google.com/maps/documentation/android-sdk/start](%20https:/developers.google.com/maps/documentation/android-sdk/start)
   * Places API: <https://developers.google.com/maps/documentation/places/web-service/overview>
   * Directions API: <https://developers.google.com/maps/documentation/directions/start>
6. **React Native Maps GitHub Repository:**  
   <https://github.com/react-native-maps/react-native-maps>
7. **Google Places Autocomplete NPM Package:**  
   <https://www.npmjs.com/package/react-native-google-places-autocomplete>
8. **React Native GetLocation Package:**  
   <https://www.npmjs.com/package/react-native-get-location>
9. **React Native AsyncStorage:**  
   <https://github.com/react-native-async-storage/async-storage>
10. **React Native Native Modules Introduction:**  
    <https://reactnative.dev/docs/native-modules-intro>
11. **UUID and Crypto Polyfill Workaround**  
    <https://www.npmjs.com/package/react-native-get-random-values>
12. **React Navigation**: <https://reactnavigation.org/>
13. Google Maps in React Native: <https://www.youtube.com/playlist?list=PLeIJUF3PiXDfOoCWgD4uibjkGQMT7a78v>

**CHAPTER 11**

**APPENDIX**

**Appendix A – Code Listings**

**A.1 PathSimulator.js –**

**Movement Simulation**

*startSimulation() {*

*if (!this.polylineCoordinates.length) return;*

*this.isPaused = false;*

*this.currentIndex = 0;*

*this.intervalId = setInterval(() => {*

*if (this.isPaused || this.currentIndex >= this.polylineCoordinates.length) {*

*this.stopSimulation();*

*return;*

*}*

*const currentLocation = this.polylineCoordinates[this.currentIndex];*

*this.onUpdateLocation(currentLocation);*

*this.checkCheckpoint(currentLocation);*

*this.currentIndex++;*

*}, this.simulationSpeed);*

*}*

**Checkpoint Timing**

*checkCheckpoint(currentLocation) {*

*const nextCheckpoint = this.checkpoints[this.currentCheckpointIndex];*

*const distance = calculateDistance(currentLocation, nextCheckpoint);*

*if (distance < 30 && !this.isTimerRunning) {*

*this.startCheckpointTimer();*

*}*

*}*

**A.2 SosModule.js –**

**SOS Modal Timer**

*useEffect(() => {*

*if (showModal) {*

*let countdown = 60;*

*const interval = setInterval(() => {*

*setTimer((prev) => {*

*if (prev <= 1) {*

*clearInterval(interval);*

*sendSOS();*

*return 0;*

*}*

*return prev - 1;*

*});*

*}, 1000);*

*return () => clearInterval(interval);*

*}*

*}, [showModal]);*

**Trigger SOS**

*const sendSOS = () => {*

*const message = `SOS: The user has not responded. Location: https://maps.google.com/?q=${location.latitude},${location.longitude}`;*

*sosContacts.forEach((number) => {*

*NativeModules.SmsModule.sendSMS(number, message);*

*});*

*};*

**A.3 AutomaticScreen.js –**

**Google Directions**

*<MapViewDirections*

*origin={currentLocation}*

*destination={destination}*

*apikey={GOOGLE\_MAPS\_APIKEY}*

*strokeColor="blue"*

*strokeWidth={4}*

*onReady={(result) => {*

*setPolylineCoordinates(result.coordinates);*

*generateAutoCheckpoints(result.coordinates);*

*}}*

*/>*

**Auto Checkpoint Generation**

*function generateAutoCheckpoints(coords) {*

*const threshold = 135;*

*let cps = [];*

*for (let i = 1; i < coords.length - 1; i++) {*

*const angle = calculateAngle(coords[i - 1], coords[i], coords[i + 1]);*

*if (angle < threshold) cps.push(coords[i]);*

*}*

*setCheckpoints(cps);*

*}*

**A.4 ManualScreen.js –Custom Checkpoint Handling**

**On Click on screen**

*<MapView onPress={(e) => handleAddCheckpoint(e.nativeEvent.coordinate)}>*

**Add Checkpoint Where the User pressed**

*function handleAddCheckpoint(coordinate) {*

*if (!isOnPolyline(coordinate, polylineCoordinates)) {*

*Alert.alert("Invalid", "Please tap near the route.");*

*return;*

*}*

**Add Time in Checkpoint**

*setTempCheckpoint(coordinate);*

*setCheckpointModalVisible(true);*

*}*

*function handleCheckpointModalOk(time) {*

*setCheckpoints([...checkpoints, { location: tempCheckpoint, expectedTime: time }]);*

*setCheckpointModalVisible(false);*

*}*

**A.5 SettingsScreen.js – AsyncStorage Integration**

**Retrieve Saved SOS Contacts from local storage**

*useEffect(() => {*

*AsyncStorage.getItem('sosContacts').then((res) => {*

*if (res) setContacts(JSON.parse(res));*

*});*

*}, []);*

**Save Contacts in local storage**

*const saveContacts = async () => {*

*try {*

*await AsyncStorage.setItem('sosContacts', JSON.stringify(contacts));*

*Alert.alert("Saved", "Contacts saved successfully.");*

*} catch (error) {*

*Alert.alert("Error", "Failed to save contacts.");*

*}*

*};*

**A.6 Navigation.js – Stack Navigator**

*const Stack = createNativeStackNavigator();*

*export default function AppNavigation() {*

*return (*

*<NavigationContainer>*

*<Stack.Navigator initialRouteName="Home">*

*<Stack.Screen name="Home" component={HomeScreen} />*

*<Stack.Screen name="Automatic" component={AutomaticScreen} />*

*<Stack.Screen name="Manual" component={ManualScreen} />*

*<Stack.Screen name="Settings" component={SettingsScreen} />*

*</Stack.Navigator>*

*</NavigationContainer>*

*);*

*}*

**Github Link:**

<https://github.com/Neeraj-Koshyari/Smart-Route-Monitoring-App>