

Atithisecure - Smart Tourist Safety Monitoring & Incident Response System (Prototype Developer Guide)

Objective

This document provides a **developer-oriented guide** for building a **single cross-platform prototype** of the Smart Tourist Safety Monitoring & Incident Response System. It includes feature explanations, technology stack, examples, and how each component works together.

1. Digital Tourist ID & Verification

Purpose: Securely identify tourists using blockchain, including KYC, itinerary, and emergency contacts.

How it works: - Tourist enters airport/hotel → scans Aadhaar/Passport. - Backend hashes data and stores in **Hyperledger Fabric**. - Tourist receives **QR code digital ID**. - Any authority can scan the QR → verify ID authenticity via blockchain.


Example: - Tourist John Doe arrives in Kolkata. - QR code contains: `TouristID#2025_IN_001` + hashed KYC. - Police scans QR → system returns “Valid ID, active during 25-30 Sep 2025”.

Developer Notes: - Node.js backend uses Hyperledger SDK for ID creation & verification. - React Native handles QR scan + display. - Encrypt sensitive data with AES; store hashes using SHA256.

2. Geofencing & Risk Scoring System

Purpose: Alert tourists when entering high-risk zones; provide safety score based on multiple factors.

How it works: - Map city/district as polygons using **Mapbox**. - Assign risk score 0–100 using factors: - Crime (40%) - Accidents / Road Conditions (15%) - Weather Hazards (10%) - Emergency Facility Availability (15%) - Events / Festivals (10%) - Tourist Feedback (10%) - Zones are color-coded: Green (safe), Yellow (caution), Red (unsafe). - Entering a red zone triggers notification.

Example: - John walks into an area with recent theft reports (crime high) and ongoing Durga Puja festival (crowd risk). Risk score = 75 → app shows Red Alert: “ High-Risk Zone. Proceed with caution.”

Developer Notes: - Backend Python calculates risk scores using sample datasets. - PostGIS/Postgres stores polygons + scores. - React Native + Mapbox visualizes zones; local cache allows offline mode (5–20 km radius).

3. BLE Mesh Communication

Purpose: Enable group communication without internet; track tourist proximity.

How it works: - Devices form **BLE mesh network**. - Predefined emergency messages are broadcast (e.g., SOS, Need Medicine, Lost). - Distance alerts if a tourist goes out of predefined radius (50m). - Messages encrypted (AES) → intermediate nodes cannot read content.

Example: - Tourist group A is exploring forest trail. - John presses “Need Medicine” → message relayed through mesh → received by group members. - John walks too far → alert: “Tourist John is leaving group radius.”

Developer Notes: - React Native BLE library (`react-native-ble-plx`) handles connections. - Node.js optional backend logs messages if internet exists. - AES-128 used for message encryption.

4. Real-Time Translation & Communication

Purpose: Enable tourists to communicate with locals and other tourists from different regions/languages.

How it works: - Speech/text input → translated into local language → TTS output. - Local reply translated back to tourist’s language. - Quick-access emergency phrases for instant communication. - Earphone mode supports hands-free operation.

Example: - John speaks in English: “I need a doctor.” - Local hears Bengali audio: “আমার ডাক্তার প্রয়োজন।” - Local replies in Bengali: “ডাক্তার এই পাশে আছেন।” - John hears English: “Doctor is nearby.”

Developer Notes: - React Native for UI + microphone. - Google/Gemini API for online translation. - Optional offline phrasebook for areas with no internet. - TTS generates audio for phrases.

5. Offline Translation & Local Phrasebook

Purpose: Provide emergency communication in remote areas without internet.

How it works: - Pre-download **phrasebook** for 20 km radius. - Phrasebook stored in SQLite/AsyncStorage with text + audio. - Tourist selects phrase → app shows translation + plays audio. - Auto-refresh phrasebook when moving to new district.

Example: - John enters a remote village in Bengal. - Offline phrasebook includes “Need water” → plays Bengali audio: “আমাকে পানি চাই।” - Tourist can communicate even without network.

Developer Notes: - Backend serves JSON phrasebooks by region. - Audio pre-generated using TTS and stored locally. - Offline mode integrated with BLE & geofencing.

6. Incident Reporting & Emergency Coordination

Purpose: Connect tourists to nearest police/hospitals; track and log incidents.

How it works: - Panic button sends location + alert to nearest authorities. - Incident report form for theft, harassment, missing cases. - Dashboard displays clusters, alert history, last known locations. - Automated mock E-FIR generation.

Example: - John presses Panic Button → dashboard shows: “John Doe, Kolkata, SOS at 3:25 PM.” - Police can visualize location and dispatch help.

Developer Notes: - React Native for mobile UI. - Node.js backend APIs for incident logging. - Mapbox for heatmap & cluster visualization. - Dashboard built in React for admin interface.

AtithiSecure – Digital Tourist ID Using Blockchain

0. Purpose

The **Digital Tourist ID** is the foundation of the AtithiSecure system. Every tourist must have a **secure, verifiable, and tamper-proof identity** before accessing app services.

Objectives for Developers:

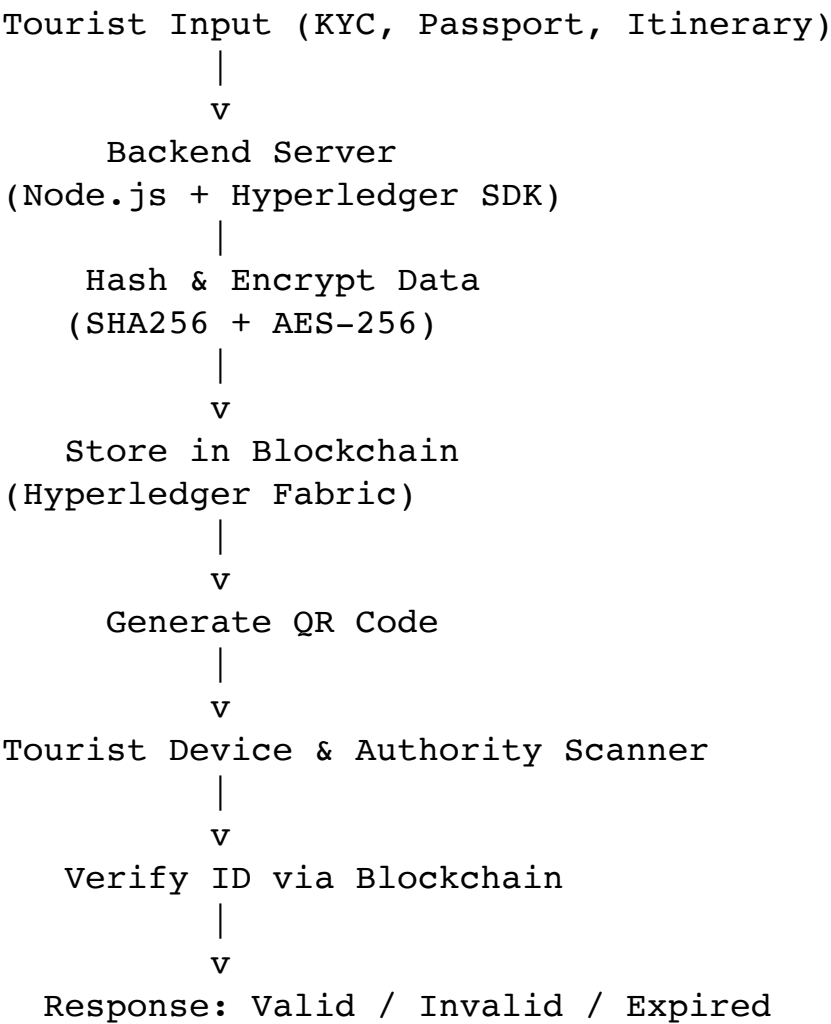
- Implement a **blockchain-based ID system** using Hyperledger Fabric.
- Demonstrate **QR-based verification** at entry points.
- Support **offline verification** for remote areas.

1. System Overview

Workflow:

1. Tourist arrives at airport/hotel/checkpoint.
2. Submits Aadhaar/Passport + itinerary + emergency contacts.
3. Backend hashes and encrypts the KYC data.
4. Data stored on **Hyperledger Fabric blockchain**.
5. Tourist receives **QR code** representing their digital ID.
6. Authorities scan QR → system verifies ID → returns verification result.

Diagram – Digital ID Workflow:



2. Technical Details

Component	Technology / Library	Description
Blockchain Network	Hyperledger Fabric	Stores tourist IDs securely; tamper-proof and permissioned.
Backend Server	Node.js + Express	Handles API requests, ID creation, hashing, and encryption.
Encryption / Hashing	AES-256 (encryption), SHA256 (hashing)	Ensures privacy and tamper-proof KYC storage.
Mobile Frontend	React Native	QR code display and scanning functionality.
QR Code Library	react-native-qrcode-svg / qr-code-generator	Generates digital IDs for scanning by authorities.

Offline Verification	Local hash + QR cache	Allows verification without network in remote areas.
----------------------	-----------------------	--

3. Implementation Steps

Step 1 – Collect Tourist Data

- Input: Name, Aadhaar/Passport, Trip Itinerary, Emergency Contacts.
- Validate format locally before sending to backend.

Step 2 – Hash & Encrypt Data

- Hash sensitive fields using SHA256.
- Encrypt QR payload using AES-256.

Step 3 – Store on Blockchain

- Smart contract `CreateTouristID` stores hashed/encrypted KYC with unique `TouristID`.
- Example blockchain entry:

```
{
  "TouristID": "2025_IN_001",
  "NameHash": "a3f4e5c7...",
  "ItineraryHash": "b7d8e9f1...",
  "EmergencyContactsHash": "c2a4b6d8...",
  "ValidFrom": "2025-09-25",
  "ValidTo": "2025-09-30"
}
```

Step 4 – Generate QR Code

- QR contains `TouristID` + encrypted hash.
- Display QR on tourist device or print for authorities.

Step 5 – Verification by Authority

- Scan QR → send `TouristID` + QR payload to backend.
- Backend decrypts → checks blockchain → returns **Valid / Invalid / Expired**.

4. Example Scenario

1. Tourist John Doe arrives in Delhi. Submits Passport + Trip Details + Emergency Contacts.
2. Backend hashes & encrypts data → stores on Hyperledger Fabric.
3. QR code issued on mobile.
4. Police scan QR → system verifies blockchain → returns *Valid ID, active 25–30 Sep 2025*.
5. Offline verification via local QR cache if network unavailable.

Diagram – Verification Example:

[Tourist Phone QR] ---> [Authority Scanner] ---> [Backend/
Local Cache] ---> [Blockchain Query] ---> [Valid/Invalid]

5. Developer Notes & Best Practices

- Always **encrypt QR payloads**; no plain KYC on device.
- Implement **offline fallback**: maintain local hash cache for remote verification.
- Use **Hyperledger test network** for prototype.
- Enforce **timestamps and validity** for each TouristID.
- Keep modules **modular**: backend, blockchain, QR generation, frontend scanning.
- Provide clear documentation of flows for hackathon demo.

6. Expected Prototype Output

- QR code issuance works on mobile.
- Authority scanner validates ID correctly.
- Blockchain shows entries for each tourist.
- Offline verification functional for small radius simulation.
- Demonstrates a secure, realistic, end-to-end ID verification system.

7. Consolidated Tech Stack

Layer	Technology	Purpose
Frontend (Mobile)	React Native	Cross-platform UI, map, BLE, translation, panic button
Backend / APIs	Node.js + Express	Blockchain, risk scoring, phrasebook, incident reporting
Blockchain	Hyperledger Fabric	Tourist ID creation & verification
Maps & Geofencing	Mapbox GL + PostGIS	Polygon zones, risk heatmaps
BLE Mesh	react-native-ble-plx	Offline group communication, distance alerts
Translation	Google Cloud / Gemini API	Online translation; offline phrasebook for emergencies
Offline Phrasebook	SQLite / AsyncStorage	Local storage of predefined phrases + audio
AI / Risk Scoring	Python + Scikit-learn / TensorFlow Lite	Calculate risk score, predictive alerts
Notifications	React Native Push / Local	Geofence & BLE alerts, panic messages
Hosting / Cloud	AWS EC2 / Lambda / S3	APIs, maps, phrasebook files
Security	AES + SHA256	Encrypt BLE messages, protect blockchain IDs

Example End-to-End Flow

1. Tourist John registers at airport → blockchain ID generated → QR code stored.
2. App shows geofenced zones → enters high-risk area → alert pops.

3. John walks in forest → BLE mesh connects with group → “Need First Aid” sent.
 4. John meets local → speaks English → app converts to Bengali → TTS audio plays.
 5. No internet? Offline phrasebook works in 20 km radius.
 6. Panic button pressed → location sent to dashboard → mock E-FIR logged.
-

Developer Notes & Tips

- Focus on **prototype-level functionality**; no need for full production security at hackathon.
 - Keep BLE and offline phrasebook lightweight for mobile performance.
 - Pre-generate small datasets for risk scoring to simulate AI behavior.
 - Modular design: blockchain, geofencing, BLE, translation are separate modules communicating via Node.js API.
 - Use mock dashboards & sample data for demonstration.
-

AtithiSecure – Geofencing & Risk Scoring System

SIH Hackathon Prototype Developer Guide

0. Purpose

Objective:

The Geofencing & Risk Scoring system ensures tourists are **aware of safety levels in real-time**, preventing accidents, thefts, or crowd-related incidents. It assigns a **dynamic safety score** for areas based on multiple factors and triggers **instant notifications** when tourists enter high-risk zones.

Expectation for Developers:

- Map areas as polygons and assign risk scores.
- Color-code zones: Green (safe), Yellow (caution), Red (unsafe).
- Notify tourists in real-time and offline where possible.

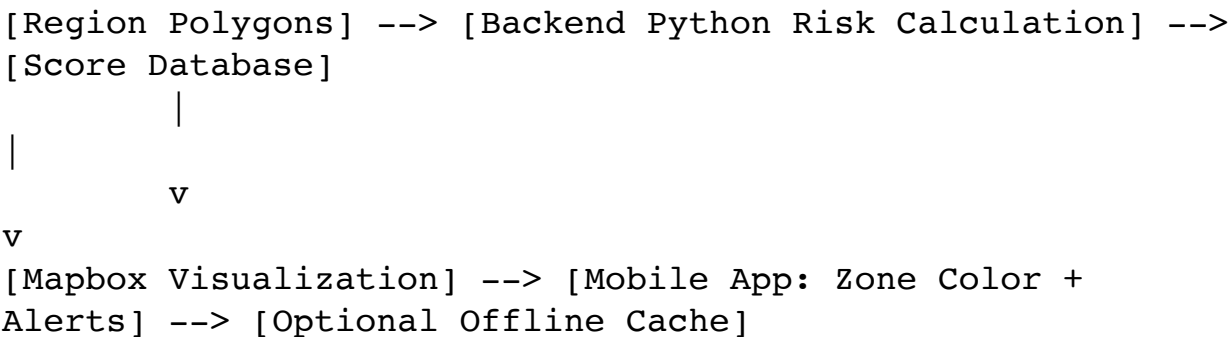
- Integrate AI/ML predictive modeling to anticipate emerging risks.

1. System Overview

Workflow:

- 1. Region Mapping:** Each district/area is mapped as a polygon (Mapbox GL).
- 2. Risk Factor Analysis:** Weighted score based on multiple criteria:
 - Crime: 40%
 - Accidents / Road Conditions: 15%
 - Weather Hazards: 10%
 - Emergency Facility Availability: 15%
 - Events / Festivals: 10%
 - Tourist Feedback: 10%
- 3. Score Calculation:** Python backend calculates 0–100 risk score.
- 4. Color-Coded Zones:** Green (safe), Yellow (moderate), Red (high risk).
- 5. Notifications:** Mobile app triggers alerts on entering red/yellow zones.

Diagram – Geofencing & Risk Score Workflow:



2. Technical Details

Component	Technology / Library	Description
Map Rendering	Mapbox GL / React	Polygon mapping, zoom, and real-time display.

Backend	Python + Flask/Django	Calculate risk scores dynamically based on weighted factors.
Database	PostgreSQL + PostGIS	Store polygon coordinates, risk scores, and
Notification System	Firebase / OneSignal / Local Push	Trigger alerts when tourists enter high-risk zones.
Offline Mode	Local SQLite /	Cache polygon & score data for areas without
AI/ML Predictive Model	TensorFlow / scikit-learn	Optional: predict emerging high-risk areas dynamically.

3. Implementation Steps

Step 1 – Map Regions

- Divide tourist zones into polygons (district/area level).
- Store coordinates in PostGIS database.

Step 2 – Calculate Risk Scores

- Collect datasets for: crime, accidents, weather, emergency facility availability, events, tourist feedback.
- Assign weights: crime 40%, accidents 15%, weather 10%, facilities 15%, events 10%, feedback 10%.
- Calculate cumulative score: $\text{Score} = \sum(\text{weighted factors})$

Step 3 – Assign Zone Colors

- 0–40 → Green (Safe)
- 41–70 → Yellow (Moderate)
- 71–100 → Red (High Risk)

Step 4 – Notify Tourists

- Mobile app receives location → checks against polygons → triggers notification if entering yellow/red zone.
- Use push notifications (online) or local caching (offline).

Step 5 – Update Dynamically

- AI/ML model predicts emerging risks (optional, prototype can use dummy data).

- Risk scores update daily/hourly.

4. Example Scenario


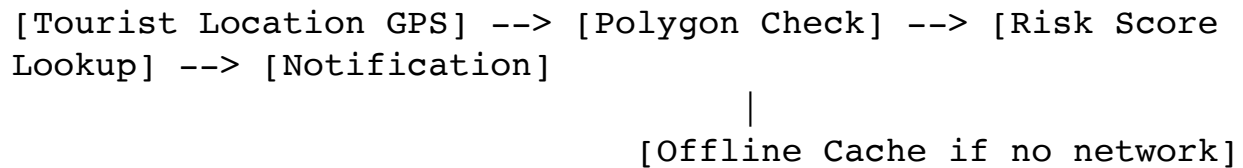
- John enters an area with recent theft reports + ongoing festival.
- Crime factor = high, Event factor = high → Risk Score = 75 → Red Zone.
- App alert: “ *High-Risk Zone. Proceed with caution.*”
- If network unavailable: offline cache still shows Red Zone alert for nearby polygons.

Diagram – Notification Flow:



5. Developer Notes & Best Practices

- **Offline-first:** Ensure polygons & risk scores cached locally for areas without network.
- **Data Accuracy:** Use realistic sample datasets for crime, accidents, events.
- **Polygon Optimization:** Keep polygons lightweight to reduce mobile memory consumption.
- **Push vs Local Notifications:** Prioritize local alerts for offline safety.
- **AI/ML Integration:** Optional for prototype, but structure backend to allow future predictive analytics.
- **Dynamic Updates:** Ensure backend can update risk scores easily without app redeployment.

6. Expected Prototype Output

- Map showing **color-coded risk zones**.
- Mobile app triggers alerts when entering yellow/red areas.

- Offline cached polygons display alerts correctly without internet.
- Demonstrates **weighted scoring logic** for judges.
- Optional AI/ML predictive risk demonstration.

AtithiSecure – BLE Mesh Communication (Detailed Technical Guide)

SIH Hackathon Prototype Developer Guide

0. Purpose

BLE Mesh Communication ensures **offline, secure, and real-time group communication** among tourists. This allows:

- Predefined emergency messages without internet.
- Proximity alerts when someone leaves the group.
- Encrypted communication where intermediate devices cannot read messages.
- Optional backend logging if network is available.

This system is essential for **remote or high-risk tourist areas** where cellular coverage may be poor.

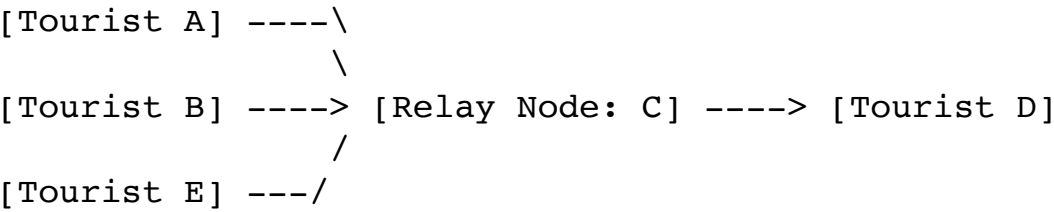
1. BLE Mesh Architecture

Mesh Network Concept:

1. **Nodes (Devices):** Every tourist's smartphone is a BLE node.
2. **Relay Nodes:** Intermediate devices relay messages without decrypting them.
3. **Root Node / Gateway (Optional):** If internet is available, one node can forward messages to backend.

- 4. **Message Broadcasting:** Messages are broadcasted through the mesh; nodes automatically relay messages to reach all recipients.

Diagram – BLE Mesh Network:



2. Device Roles

Role	Responsibility
Normal Node	Sends & receives messages; relays if within range.
Relay Node	Relays messages to extend network; cannot decrypt payload.
Gateway Node	Optional; forwards mesh messages to backend when internet available.
Listener Node	Only listens for emergency messages (for battery optimization).

3. How Messages Flow

Step-by-Step:

1. **Message Creation:**
 - Tourist selects predefined message (e.g., SOS).
 - App encrypts message payload using AES-128.
 - Payload includes: message type, sender ID, timestamp.
2. **Message Broadcast:**
 - Device broadcasts encrypted message to nearby nodes using BLE advertising packets.
3. **Relay by Neighbor Nodes:**
 - Each neighboring node receives the message.
 - Checks if the message was already seen (message ID).

- If new, relays to its neighbors.
4. **Message Reception:**
- Final recipients decrypt the message.
 - Intermediate nodes **cannot read the content** (encryption ensures privacy).
5. **Distance/Proximity Check:**
- Nodes estimate distance using **RSSI (signal strength)** or **hop count**.
 - If a device exceeds the safe radius, an alert triggers: *“Tourist X is leaving group radius.”*
6. **Optional Backend Logging:**
- When internet available, a gateway node forwards messages/alerts to backend server for logging, dashboards, or analysis.

4. Predefined Emergency Messages

Message Type	Example Payload	Use Case
SOS	{ “type”: “SOS”, “sender”: “Tourist001” }	Immediate emergency
Need Medicine	{ “type”: “Medicine”, “sender”: “Tourist001” }	Medical assistance
Lost / Help	{ “type”: “Lost”, “sender”: “Tourist001” }	Lost in area / needs assistance
Need Bandage	{ “type”: “Bandage”, “sender”: “Tourist001” }	Minor injury

5. Technical Implementation

Mobile App (React Native):

- Library: `react-native-ble-plx` for BLE connections.
- Handles scanning, connection, broadcast, and relay.
- Integrates with **geofencing module** to trigger location-based alerts.

Encryption:

- AES-128 for message content.

- Intermediate devices cannot decrypt payload; only recipients can.

Offline Operation:

- Mesh operates entirely without internet.
- Local device cache maintains list of nearby nodes & message history.

Optional Backend Logging:

- Node.js + MongoDB / Firebase collects alerts/messages when gateway has internet.

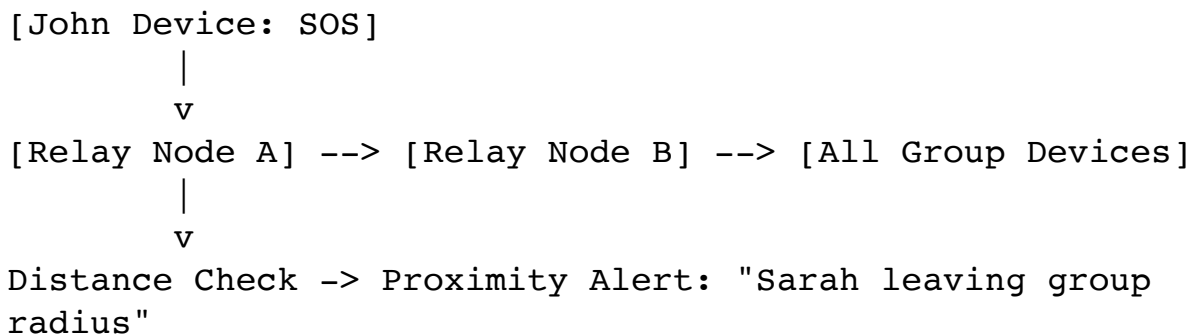
Distance Estimation:

- RSSI signal strength translated to approximate distance (meters).
- Safe radius configurable (e.g., 50–100m).

6. Example Scenario

1. **Group Exploration:** 5 tourists in forest trail.
2. **Emergency Message:** John presses “Need Medicine.”
3. **Broadcast:** John’s device encrypts & broadcasts message to all nearby devices.
4. **Relay:** Neighbor nodes relay message through mesh.
5. **Alert:** All group members’ devices receive and decrypt message.
6. **Proximity Check:** Sarah walks 60m away → alert triggers: “*Tourist Sarah leaving group radius.*”
7. **Backend Logging:** If internet is available, message logged for monitoring dashboard.

Diagram – Message Flow & Proximity Alert:



|
v

[Optional Gateway Node -> Backend Logging]

7. Developer Notes & Best Practices

- **Low-Energy Optimization:** BLE consumes battery; use sleep intervals & scan intervals wisely.
- **Encryption:** All messages must be AES-encrypted; avoid plaintext broadcast.
- **Message Deduplication:** Track message IDs to prevent repeated relays.
- **Offline-first Design:** Mesh must function without network; backend logging optional.
- **Modular Implementation:** Separate mesh layer, encryption, and message handling.
- **Testing:** Simulate multiple devices for proper mesh relaying and proximity alerts.

8. Expected Prototype Output

- Devices auto-connect forming BLE mesh.
- Predefined emergency messages propagate securely through the mesh.
- Proximity alerts trigger when someone leaves the safe radius.
- Optional backend logging demonstrates integration with monitoring dashboard.
- Shows **offline communication & group safety coordination** for judges.

AtithiSecure – Real-Time Translation & Communication

SIH Hackathon Prototype Developer Guide

0. Purpose

Objective:

Enable tourists to **communicate effectively with locals and other tourists** speaking different languages. The system supports:

- Speech-to-speech and text-to-text translation.
- Quick-access emergency phrases.
- Hands-free operation via earphones.
- Offline support for remote areas.

Developer Goals:

- Capture speech/text input and translate in **real-time**.
- Convert translations into **TTS audio output** for local understanding.
- Translate local replies back to the tourist's language.
- Integrate with **offline phrasebooks** for areas with no network.

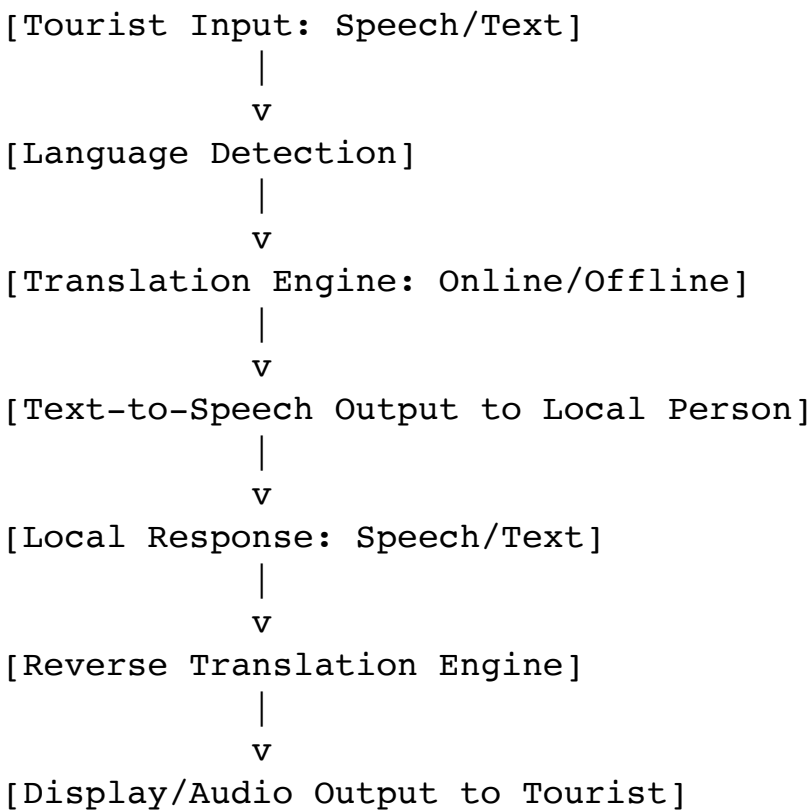
1. System Overview

Workflow:

1. **Input Capture:** Tourist speaks or types a message.
2. **Language Detection:** Detect input language automatically.
3. **Translation Engine:** Translate text/speech to target language.
4. **Text-to-Speech (TTS):** Convert translated text to audio for local person.

- 5. **Local Response:** Local replies via speech/text.
- 6. **Reverse Translation:** Convert local reply into tourist’s language.
- 7. **Output Delivery:** Display text and/or play audio for tourist.

Diagram – Real-Time Translation Flow:



2. Technical Details

Component	Technology / Library	Description
Mobile Frontend	React Native	Capture microphone input, display
Speech Recognition	Google Cloud Speech-to-Text (API)	Convert spoken language to text.
Translation Engine	Google Cloud Translate / Offline	Translate between tourist and local
Text-to-Speech (TTS)	Google Cloud TTS / Offline	Convert translated text to audio for local
Offline Phrasebook	SQLite / AsyncStorage	Preloaded phrases for areas with no
Predefined Emergency Phrases	JSON	Quick-access phrases for SOS, medical, or help requests.

Audio Output	Mobile audio API / Earphone integration	Plays TTS output for tourist/local communication.
--------------	---	---

3. Implementation Steps

Step 1 – Capture Input:

- Tourist speaks or types a message.
- Microphone access required; fallback for text input.

Step 2 – Language Detection:

- Detect tourist's language automatically (e.g., English, Hindi, French).
- Required for both online translation API or offline phrasebook selection.

Step 3 – Translation:

- **Online:** Send text to API (Google Translate / Gemini).
- **Offline:** Search preloaded phrasebook by context.
- Example JSON offline phrasebook:

```
{
  "phrase": "I need a doctor",
  "language": "en",
  "bengali": "আমার ডাক্তার প্রয়োজন",
  "audioFile": "need_doctor_bn.mp3"
}
```

Step 4 – Text-to-Speech (TTS):

- Convert translated text to audio.
- Play audio through device speaker or earphones.

Step 5 – Receive Local Response:

- Local person replies in their language.
- Speech-to-text converts it to text for app.

Step 6 – Reverse Translation:

- Translate local reply back to tourist's language.
- Display text and optionally play audio.

Step 7 – Quick Access & Emergency Phrases:

- Emergency phrases (SOS, Need Medicine, Lost) accessible via a button.
- Offline phrasebook ensures usability in no-network zones.

4. Example Scenario

- John (tourist) in Kolkata wants medical help.
- John speaks in English: “I need a doctor.”
- App detects language → translates to Bengali → TTS output: “আমার ডাক্তার প্রয়োজন।”
- Local replies: “ডাক্তার এখানে আছেন।”
- App translates back to English → audio/text output: “Doctor is nearby.”
- Works offline if phrase is in local preloaded phrasebook.

5. Developer Notes & Best Practices

- **Offline-first:** Preload phrasebooks for remote areas to ensure critical communication.
- **Audio caching:** Cache TTS audio files to reduce latency.
- **Emergency phrases:** Prioritize lightweight, quick-access predefined messages.
- **Language detection fallback:** Use default language if detection fails.
- **Integration with BLE & Geofencing:** Messages can include context (location, risk zone) for safety coordination.
- **UI/UX:** Simple buttons for “Emergency Phrases,” “Speak,” “Listen,” and easy playback control.

6. Expected Prototype Output

- Tourist can speak or type messages in their language.

- Local person hears or reads translation in their language.
- Reverse translation allows tourist to understand reply.
- Works **offline for predefined phrases**, online for free-text translation.
- Demonstrates **multilingual emergency communication** for judges.

AtithiSecure – Offline Translation & Local Phrasebook

SIH Hackathon Prototype Developer Guide

0. Purpose

Objective:

Enable tourists to **communicate in remote areas without internet connectivity** by using **preloaded local language phrases**. This ensures:

- Emergency communication in offline zones.
- Quick access to predefined phrases (SOS, medical help, lost, etc.).
- Audio playback for ease of understanding.
- Seamless integration with BLE mesh and geofencing for safety alerts.

Developer Goals:

- Provide offline text and audio translations for local languages.
- Cache local phrasebooks for specific regions (e.g., 20 km radius).
- Auto-refresh or download new phrasebooks when tourist moves to a new area.
- Maintain small storage footprint using compressed files and vector formats for phrases.

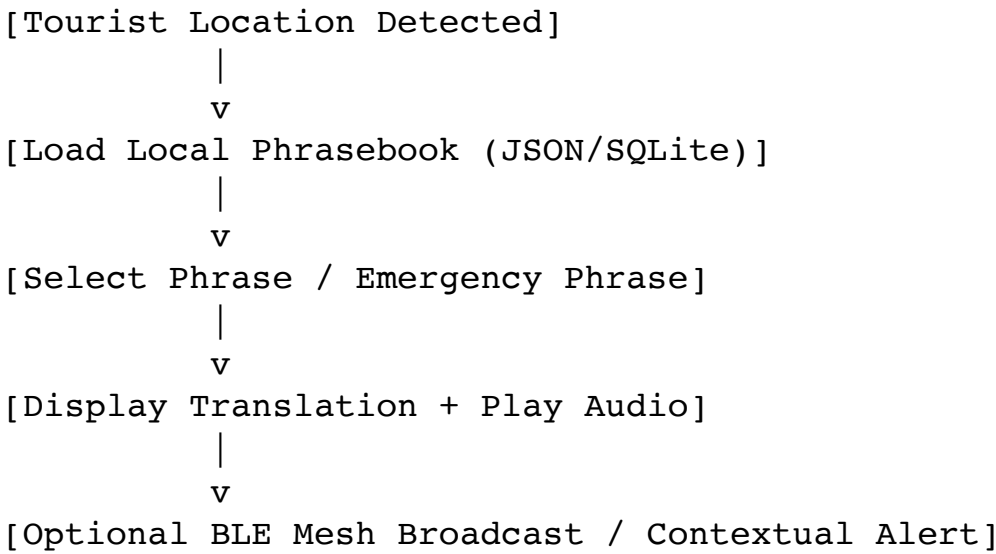
1. System Overview

Workflow:

1. Tourist enters a new region or offline zone.

- 2. App detects location via GPS / geofencing.
- 3. App loads local phrasebook from **device storage** (SQLite / AsyncStorage).
- 4. Tourist selects a phrase → app shows **translated text** and plays **audio**.
- 5. Optionally, BLE mesh messages can include context from the phrasebook for group alerts.

Diagram – Offline Phrasebook Flow:



2. Technical Details

Component	Technology / Library	Description
Mobile Frontend	React Native	UI for phrase selection, playback, and text
Offline Phrasebook	SQLite / AsyncStorage	Stores regional phrases and audio files.
Predefined Emergency Phrases	JSON / CSV	Structured list of commonly used emergency phrases.
Audio Output	Mobile audio API / Earphones	Plays pre-recorded TTS audio for tourist or local person.
Geofencing Integration	Mapbox / GPS	Detects tourist location to load correct local
Optional BLE Integration	react-native-ble-plx	Allows broadcasting emergency messages from phrasebook.

3. Implementation Steps

Step 1 – Define Phrasebooks:

- Structure offline phrasebook in JSON:

```
[
  {
    "phraseID": 1,
    "english": "I need water",
    "bengali": "আমাকে পানি চাই",
    "audioFile": "need_water_bn.mp3"
  },
  {
    "phraseID": 2,
    "english": "I am lost",
    "bengali": "আমি হারিয়ে গেছি",
    "audioFile": "lost_bn.mp3"
  }
]
```

- Include **text + audio file** for each phrase.

Step 2 – Load Phrasebook by Region:

- Detect tourist location via GPS.
- Load pre-downloaded phrasebook for current 20 km radius.
- Cache in AsyncStorage / SQLite for offline use.

Step 3 – Display & Playback:

- Tourist selects phrase → app shows translation.
- Play corresponding audio file.
- Optional: use TTS engine for dynamic phrase audio.

Step 4 – Offline Emergency Use:

- Integrate with BLE mesh for predefined messages.
- Phrasebook IDs map to BLE message types (e.g., SOS, Lost).
- Local translation ensures immediate comprehension without internet.

Step 5 – Auto-Update Phrasebooks:

- When moving to a new region or internet becomes available:
 - Download latest regional phrasebook.
 - Replace or merge with existing offline cache.

4. Example Scenario

- John enters a remote village in West Bengal with no internet.
- App detects location → loads Bengali phrasebook.
- John selects: “I need water.”
- App displays Bengali: “আমাকে পানি চাই” and plays audio: “আমাকে পানি চাই.mp3”.
- BLE mesh can optionally broadcast: “Tourist John needs water.”
- No internet required; message is understood by local and other tourists.

5. Developer Notes & Best Practices

- **File Size Optimization:**
 - Compress audio files.
 - Use vector or lightweight formats to reduce storage.
- **Caching Strategy:**
 - Keep local phrasebooks for current region only (5–20 km radius).
 - Evict old regions to save space.
- **Offline-First Design:**
 - Ensure core phrases always available offline.
 - TTS fallback if audio file missing.
- **Integration with Other Modules:**
 - BLE Mesh → send selected phrase as emergency message.
 - Geofencing → auto-load phrasebooks for current polygon zone.

- **Multilingual Support:**
 - Maintain phrasebooks in 10+ Indian languages.
 - Include English as default language.

6. Expected Prototype Output

- Tourist can **select and hear translated phrases offline**.
- BLE mesh integration allows emergency phrases to be shared in real-time.
- Works in remote locations without internet.
- Demonstrates **multilingual, offline emergency communication** for judges.

AtithiSecure – Incident Reporting & Emergency Coordination

SIH Hackathon Prototype Developer Guide

0. Purpose

Objective:

Enable tourists and authorities to **report, track, and respond to incidents in real-time**. This system ensures:

- Immediate alerting of police, hospitals, and other emergency services.
- Logging incidents for authorities with geolocation and timestamp.
- Automated mock E-FIR creation for missing tourists or other emergencies.
- Integration with geofencing, BLE mesh, and translation modules for comprehensive safety.

Developer Goals:

- Provide a **panic button** and reporting form.
- Integrate **real-time location sharing**.

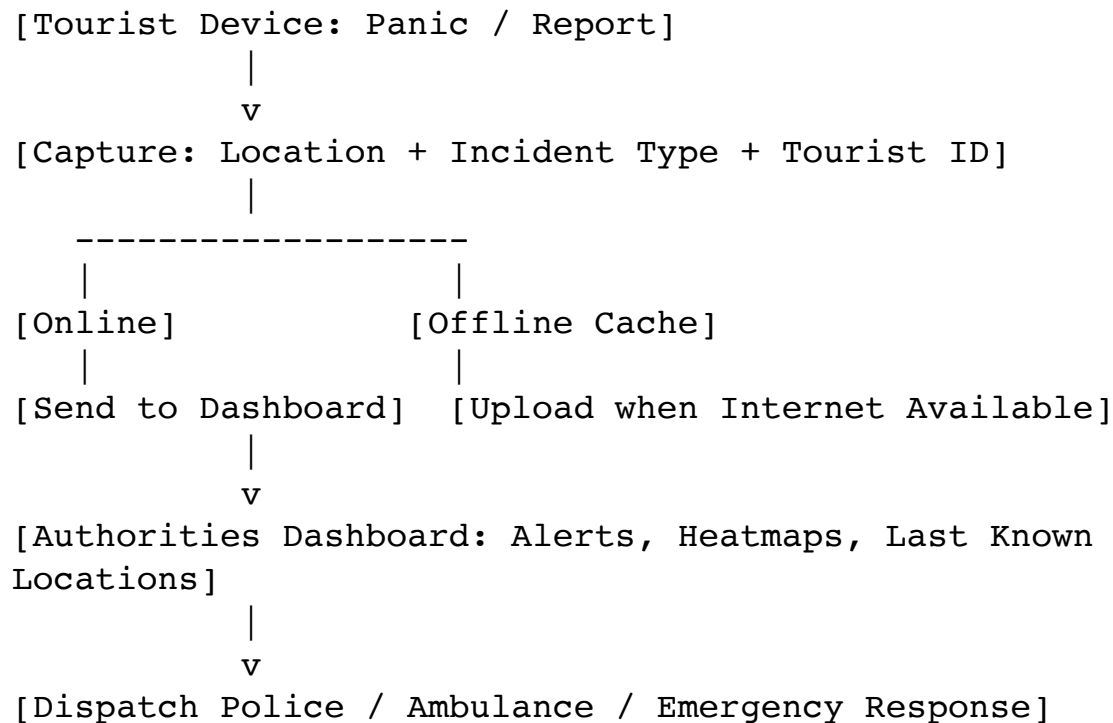
- Display alerts and clusters on **authorities' dashboards**.
- Ensure **offline capability** for remote areas.

1. System Overview

Workflow:

1. Tourist triggers an incident (via Panic Button or report form).
2. App captures **location, timestamp, tourist ID, and incident type**.
3. Data sent to **authorities' dashboard** (online) or stored locally for offline upload.
4. Dashboard shows **heatmaps, clusters, and last known locations**.
5. Optional automated **mock E-FIR generation** for missing tourists.
6. Authorities can **dispatch help** or coordinate emergency response.

Diagram – Incident Reporting Flow:



2. Key Features

Feature	Description
Panic Button	Single press sends SOS with location to authorities & emergency
Incident Report Form	For theft, harassment, medical emergency, or missing tourist
Real-Time Alerts	Push notifications to authorities & group members via BLE mesh or internet.
Location & Timestamp	Accurate GPS tracking with date/time for each report.
Mock E-FIR Generation	Automatic creation of temporary digital FIR for rapid response.
Heatmap & Cluster Visualization	Shows high-risk zones or multiple incidents in an area.
Offline Functionality	Store reports locally if network unavailable; upload when online.

3. Technical Implementation

Mobile App (React Native):

- UI for Panic Button and Incident Form.
- GPS module for geolocation.
- Local SQLite / AsyncStorage for offline caching.

Backend (Node.js + Express):

- REST APIs to receive incident data.
- MongoDB/PostgreSQL to store reports and tourist history.
- Integration with Hyperledger for tourist ID verification.

Dashboard (React / Mapbox):

- Real-time visualization of incidents.
- Heatmaps for high-risk areas.
- Last known location tracking.
- Mock E-FIR creation & download as PDF.

Optional Integrations:

- BLE Mesh → alert nearby group members about the incident.
- Translation module → automatically translate incident details for local authorities.

- Geofencing → tag incidents to specific zones/polygons.

4. Implementation Steps

Step 1 – Panic Button:

- Tourist presses the button → capture location, tourist ID, timestamp.
- Encrypt data (AES) before sending.
- Trigger push notification to nearest authorities and emergency contacts.

Step 2 – Incident Form:

- User fills form: type of incident, severity, optional notes/photos.
- Local storage if offline, otherwise send to backend immediately.

Step 3 – Data Logging & Storage:

- Backend stores incident with **timestamp, tourist ID, geolocation, type, severity**.
- Include blockchain-based ID verification for authenticity.

Step 4 – Dashboard Visualization:

- Real-time heatmaps of incidents.
- Clustering for multiple events in proximity.
- Show last known location for missing tourists.

Step 5 – Automated Mock E-FIR:

- Predefined FIR template with tourist ID, incident type, and timestamp.
- Generate PDF for quick download or dispatch to authorities.

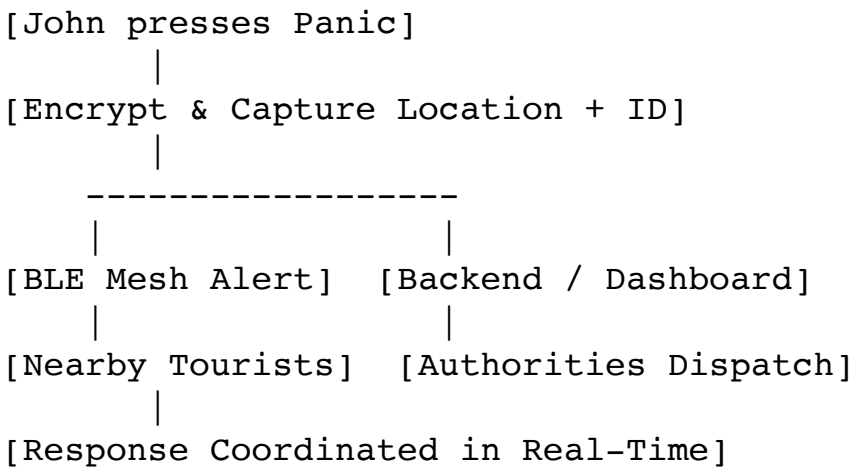
Step 6 – Offline Handling:

- Cache incidents in local storage.
- Automatically upload when tourist device regains connectivity.
- Optionally, BLE mesh can alert nearby tourists even without internet.

5. Example Scenario

- John is hiking and witnesses a tourist fainting.
- He presses Panic Button → sends SOS + location + ID.
- Nearby group members receive BLE mesh alert: “Emergency: Tourist John needs help.”
- Backend dashboard logs incident → displays location on heatmap.
- Automated mock E-FIR generated → authorities dispatch ambulance.
- If offline, incident is cached locally and uploaded later.

Diagram – Prototype Example:



6. Developer Notes & Best Practices

- **Offline-first:** Always cache incidents locally if no network.
- **Encryption:** AES encrypt all incident payloads.
- **Integration:** Connect incident module with BLE Mesh, Geofencing, and Translation modules.
- **UI/UX:** Ensure Panic Button is prominent, one-touch, and accessible.
- **Testing:** Simulate multiple incidents and offline scenarios to ensure reliability.
- **Scalability:** Backend should support multiple simultaneous incidents and tourists.

7. Expected Prototype Output

- Tourists can **report emergencies quickly** via Panic Button or form.
- Authorities receive **real-time alerts** and see visualized incidents on dashboard.
- **Offline incident caching** ensures no data loss in remote areas.
- BLE mesh alerts nearby tourists in absence of internet.
- Demonstrates **rapid emergency coordination** and safety management for judges.