FalconResQ: Disaster Management Ground Station

EXHAUSTIVE TECHNICAL REFERENCE MANUAL

Project Classification: Level 1 - Complete Technical Reference
Version: 3.0.0 (Comprehensive Edition)
Date: December 24, 2025
Developer: Asshray Sudhakara (ECE'27, Aviation Coordinator & EvRe Domain Student @ MARVEL, UVCE)
Documentation Scope: Complete system documentation with all algorithms, formulas, and implementation details

---

TABLE OF CONTENTS

---

1. EXECUTIVE SUMMARY

1.1 Project Overview

FalconResQ is an enterprise-grade disaster management ground station application that transforms raw LoRa wireless signals into actionable intelligence for rescue operations. The system enables automatic victim detection within a 10-20 km range, providing real-time visualization, intelligent prioritization, and comprehensive analytics for rescue coordinators.

Core Mission: Reduce victim location time from hours to seconds, enabling faster rescue response and saving lives during disaster scenarios.

1.2 Problem Statement

Traditional disaster response faces critical challenges:

- **Manual Search Operations:** Rescue teams spend hours physically searching for victims
- **Limited Range:** Ground-based search limited to 1-2 km visibility
- **Safety Risks:** Rescue personnel exposed to dangerous conditions during search
- **Coordination Issues:** Multiple teams may search overlapping areas
- **Data Loss:** Paper-based tracking prone to errors and loss
- **No Prioritization:** Rescue teams cannot identify most critical victims
- **Limited Analytics:** Post-operation analysis is manual and incomplete

1.3 Solution Architecture

FalconResQ addresses these challenges through:

1. **Automatic Detection:** Victims broadcast their location via LoRa (10-20 km range)
2. **Real-Time Processing:** Ground station receives and processes signals instantly
3. **Intelligent Prioritization:** AI-like algorithm scores victims based on signal strength and time
4. **Interactive Visualization:** Real-time map shows all victim locations
5. **Status Tracking:** Complete audit trail of rescue operations
6. **Scientific Analytics:** Automated metrics for operation efficiency
7. **Professional Reporting:** Multi-format export for compliance and analysis

1.4 Technical Achievement

Single Python Application Integrating: - Hardware communication (PySerial - COM port interface) - Geospatial visualization (Folium - Leaflet.js wrapper) - Real-time analytics (NumPy/Pandas vectorization) - Responsive web UI (Streamlit framework) - Persistent storage (JSON auto-save with backups) - Asynchronous broadcasting (WebSocket server) - Browser integration (JavaScript geolocation API)

Key Metrics: - **Detection Range:** 10-20 km (LoRa wireless range) - **Processing Speed:** 50-100 packets/second - **Map Rendering:** <1 second for 100 victims - **Memory Footprint:** ~2 MB per 100 victim records - **Supported Victims:** 100-500 concurrent tracking - **Update Latency:** <500ms from signal to UI update

---

2. SYSTEM ARCHITECTURE

2.1 High-Level Architecture

```
                    VICTIM DEVICES
(WiFi LoRa 32 V3 - ESP32S3 + SX1262)
- Broadcast GPS coordinates via LoRa
- Transmit RSSI, battery status, victim ID

                  LoRa Wireless (10-20 km range)


                 DRONE RELAY (Optional)
(Wireless Tracker - ESP32S3 + SX1262 + GPS)
- Extends range for remote victims
- Aerial signal relay

                  LoRa Relay


              GROUND STATION HARDWARE
(WiFi LoRa 32 V3 - ESP32S3 + SX1262)
- Receives LoRa packets
- Outputs to serial port (COM/USB)

                  Serial Communication (115200 baud)


            FALCONRESQ APPLICATION LAYER


         SERIAL READER MODULE
  - Background thread
  - Packet parsing & validation
  - Observer pattern callbacks




         DATA MANAGER MODULE
  - In-memory victim database
  - CRUD operations
  - Auto-save persistence (JSON)
  - RSSI history tracking
```

```
     MAP MANAGER          ANALYTICS MODULE
     - Folium maps        - Statistics
     - Markers            - Clustering
     - Popups             - Trend analysis



            STREAMLIT UI LAYER

       Dashboard       Analytics        Export
         Page            Page            Page


       Settings
         Page



         WEBSOCKET SERVER (Real-time broadcast)
     - Async message broadcasting
     - Multi-client support



                    PERSISTENT STORAGE
  - victims_backup.json (auto-save every 30s)
  - rescue_log.csv (rescue operations log)
  - operation_exports/ (CSV, JSON, PDF reports)
```

2.2 Data Flow Architecture

Phase 1: Signal Reception

`Victim Device → LoRa Transmission → Ground Station Hardware → Serial Port`

Phase 2: Packet Processing

```
Serial Port → SerialReader._read_loop() → JSON parsing → Validation
```

Phase 3: Data Ingestion

```
Valid Packet → on_packet_received callback → DataManager.add_or_update_victim()
```

Phase 4: State Update

```
DataManager update → Session state increment → WebSocket broadcast → force_rerun flag
```

Phase 5: UI Refresh

```
force_rerun check → st.rerun() → Page re-render → Map update → Metrics update
```

2.3 Threading Model

Main Thread (Streamlit): - UI rendering and event handling - User interaction processing - Page navigation - Session state management

Serial Reader Thread (Background): - Continuous serial port monitoring - Packet reading and parsing - Callback invocation - Error handling and reconnection

WebSocket Server Thread (Async): - Asynchronous message broadcasting - Client connection management - Real-time packet distribution

Auto-Save Thread (Periodic): - Periodic JSON backup creation - Data persistence management - Checkpoint creation

2.4 Module Dependencies

```
app.py
  config.py (configuration constants)
  modules/
     serial_reader.py → utils/validators.py
     data_manager.py → config.py
     map_manager.py → config.py, utils/helpers.py
     analytics.py → utils/helpers.py, numpy, pandas
     websocket_server.py → asyncio, websockets
  _pages/
     dashboard.py → modules/*, utils/*
     analytics.py → modules/analytics, plotly
     export.py → modules/data_manager
     settings.py → modules/serial_reader, config
  utils/
     helpers.py → config, datetime, math
     validators.py → config
```

---

3. TECHNOLOGY STACK ANALYSIS

3.1 Programming Languages

Python 3.11 **Usage:** Primary application language (100% of backend logic)

Justification: - **Rapid Development:** Interpreted language enables fast prototyping - **Rich Ecosystem:** 300,000+ packages on PyPI - **Scientific Computing:** NumPy/Pandas for analytics - **Hardware Integration:** PySerial for COM port communication - **Cross-Platform:** Windows/Mac/Linux compatibility

Alternatives Considered: - Java: More verbose, slower development - C++: Overkill for this application, harder maintenance - JavaScript (Node.js): Weaker scientific computing libraries

JavaScript (ES6+) **Usage:** Browser-side geolocation and localStorage integration

Justification: - **Browser APIs:** Direct access to Geolocation API - **Local Storage:** Persistent user preferences - **Streamlit Integration:** Via `components.html()` injection

Code Example:

```
navigator.geolocation.getCurrentPosition(
    function(position) {
        localStorage.setItem('user_lat', position.coords.latitude);
        localStorage.setItem('user_lon', position.coords.longitude);
    }
);
```

JSON **Usage:** Data serialization format for persistence and packet structure

Justification: - **Human-Readable:** Easy debugging - **Universal:** Supported by all modern platforms - **Lightweight:** Minimal overhead

3.2 Frameworks & Libraries

Streamlit 1.28+ **Purpose:** Web UI framework

Technical Details: - **Architecture:** Pure Python web apps without HTML/CSS/JS - **Rendering Model:** Reactive - reruns script on state change - **Session State:** Persistent data across reruns via `st.session_state` - **Component System:** Extensible via custom components

Why Chosen Over Alternatives:

Framework	Pros	Cons	Decision
Flask	Full control, mature	Requires HTML/CSS/JS expertise	Too complex
Django	Full-featured, ORM	Overkill for this project	Too heavy
Streamlit	Pure Python, rapid dev	Limited customization	**SELECTED**
Dash	Plotly integration	More complex than Streamlit	Unnecessary complexity

Key Features Used: - `st.session_state` - Global state management - `st.rerun()` - Forced UI refresh on new packets - `st.columns()` - Responsive layouts - `st.metric()` - Dashboard metrics - `components.html()` - JavaScript injection

Folium 0.14+ **Purpose:** Interactive mapping library

Technical Details: - **Based On:** Leaflet.js (JavaScript mapping library) - **Rendering:** Generates HTML/JS that Streamlit can embed - **Tile Sources:** Google Maps, OpenStreetMap, etc.

Why Chosen: - **Python-Native:** No JavaScript knowledge required - **Leaflet Power:** Full Leaflet.js functionality - **Streamlit Integration:** Via `streamlit-folium` adapter

Map Generation Flow:

```
folium.Map() → Add markers → Add layers → Generate HTML →
streamlit_folium.st_folium() → Render in browser
```

NumPy 1.24+ **Purpose:** Numerical computing and array operations

Usage Examples: - RSSI history arrays - Statistical calculations (mean, std, percentiles) - Geographic coordinate operations - Vectorized distance calculations

Performance Benefit: - Pure Python loop: 10ms for 100 victims - NumPy vectorized: 0.5ms for 100 victims (20x faster)

Pandas 2.0+ **Purpose:** Data manipulation and analysis

Usage Examples: - DataFrame conversion for export - Time series analysis - Groupby operations for sector analysis - CSV generation

Code Example:

```
df = pd.DataFrame([v for v in victims.values()])
df.to_csv('export.csv', index=False)
```

PySerial 3.5+ **Purpose:** Serial port communication

Technical Details: - **Protocol:** RS-232 serial communication - **Baud Rates Supported:** 9600-921600 baud - **Timeout Handling:** Non-blocking read with timeout - **Platform Support:** Windows (COM), Linux (ttyUSB), Mac (cu.*)

Communication Parameters:

```
serial.Serial(
    port='COM23',
    baudrate=115200,
    bytesize=8,         # 8 data bits
    parity='N',         # No parity
    stopbits=1,         # 1 stop bit
    timeout=1           # 1 second read timeout
)
```

Plotly 5.17+ **Purpose:** Interactive charts and visualizations

Chart Types Used: - Pie charts (status distribution) - Bar charts (rescue timeline) - Line charts (signal trends) - Scatter plots (geographic distribution)

WebSockets (websockets library) **Purpose:** Real-time bidirectional communication

Architecture:

```
async def handler(websocket):
    connected_clients.add(websocket)
    try:
        await websocket.wait_closed()
    finally:
        connected_clients.remove(websocket)
```

Python-dotenv **Purpose:** Environment variable management

Usage:

```
load_dotenv()  # Loads .env file
GOOGLE_MAPS_API_KEY = os.getenv('GOOGLE_MAPS_API_KEY')
```

3.3 Data Formats

JSON (JavaScript Object Notation) **Usage:** Packet format, persistent storage, export

Packet Structure:

```json
{
  "ID": 1001,
  "LAT": 13.022456,
  "LON": 77.587234,
  "RSSI": -75,
  "STATUS": "STRANDED",
  "BATTERY": 85,
  "TIMESTAMP": "2025-12-24T10:30:45"
}
```

CSV (Comma-Separated Values) **Usage:** Export format, rescue logs

Export Format:

```
ID,LAT,LON,STATUS,RSSI,FIRST_DETECTED,LAST_UPDATE,RESCUED_BY
1001,13.022456,77.587234,RESCUED,-75,2025-12-24 10:30,2025-12-24 11:15,Operator-01
```

3.4 Hardware Specifications

Ground Station Device **Model:** WiFi LoRa 32 (V3)
Chipset: ESP32-S3 + SX1262
Frequency: 868/915 MHz (regional)
TX Power: 22 dBm (158 mW)
RX Sensitivity: -148 dBm
Range: 10-20 km (line of sight)

Victim Device (Beacon) **Model:** WiFi LoRa 32 (V3)
GPS: Integrated or external module
Battery: Li-Ion rechargeable
Transmission Interval: 5-60 seconds (configurable)

Drone Relay (Optional) **Model:** Wireless Tracker
GPS: High-precision GNSS
Purpose: Extend range to remote areas

---

4. CORE MODULES DOCUMENTATION

4.1 app.py - Application Entry Point

Purpose: Main application controller handling initialization, navigation, and global state management.

File Size: 544 lines
Functions: 4 primary functions

Function: `initialize_session_state()` **Purpose:** Initialize all Streamlit session state variables on first run

Algorithm:

```
IF 'data_manager' NOT IN session_state:
    CREATE new DataManager instance
    STORE in session_state

IF 'serial_reader' NOT IN session_state:
    CREATE new SerialReader instance
    STORE in session_state

FOR each configuration variable:
    IF NOT EXISTS in session_state:
        SET default value from config.py

IF WebSocket server NOT started:
    CALL start_websocket_server()
    MARK ws_server_started = True
```

Session State Variables Initialized:

Variable	Type	Default	Purpose
`data_manager`	DataManager	new instance	Victim database
`serial_reader`	SerialReader	new instance	COM port handler
`serial_connected`	bool	False	Connection status
`operator_name`	str	"Operator"	Current user
`operation_start_time`	datetime	now()	Operation timestamp
`packet_count`	int	0	Total packets received
`error_count`	int	0	Parse errors
`last_packet_time`	datetime	None	Most recent packet
`show_rescued`	bool	True	UI filter
`show_heatmap`	bool	False	Map layer toggle
`show_priority_only`	bool	False	UI filter
`serial_port`	str	None	COM port (user must set)
`baud_rate`	int	115200	Serial speed
`map_center`	list	[13.022, 77.587]	Map coordinates

Variable	Type	Default	Purpose
`rescue_centre_lat`	float	13.022	Base location
`rescue_centre_lon`	float	77.587	Base location
`rssi_strong_threshold`	int	-70	Signal threshold (dBm)
`rssi_weak_threshold`	int	-85	Signal threshold (dBm)
`time_critical_threshold`	int	15	Time threshold (min)
`force_rerun`	bool	False	UI update trigger
`ws_server_started`	bool	False	WebSocket status

Function: **`get_user_location()`** **Purpose:** Generate JavaScript code to request browser geolocation

Returns: HTML string with embedded JavaScript

JavaScript Generated:

```
<script>
function getLocation() {
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(
            function(position) {
                // Success callback
                window.parent.postMessage({
                    type: 'streamlit:setComponentValue',
                    lat: position.coords.latitude,
                    lon: position.coords.longitude
                }, '*');
            },
            function(error) {
                // Error callback
                console.log('Geolocation error:', error);
            }
        );
    }
}
getLocation();
</script>
```

Browser Permission Flow: 1. User clicks "Detect My Location" in Settings 2. JavaScript executes `navigator.geolocation.getCurrentPosition()` 3. Browser prompts user for permission 4. If granted, coordinates returned to callback 5. Stored in `localStorage` for persistence 6. PostMessage sent to Streamlit parent window

Function: **`render_sidebar()`** **Purpose:** Render navigation sidebar with status indicators

Returns: Selected page name (str)

UI Components: 1. **Title:** "FalconResQ Ground Station Control" 2. **Navigation Radio Buttons:** - Dashboard - Analytics - Export Data - Settings 3. **Connection Status:** - Port selection warning (if no port) - Connected indicator (green) with Disconnect button - Disconnected indicator (red) with Connect button 4. **Quick Statistics:** - Total victims - Stranded count - Rescued count - En-route count 5. **Data Stream Info:** - Packet count - Error count - Last packet time (seconds ago) 6. **Operation Duration:** - Time since operation start (hours:minutes)

Serial Connection Logic:

```
IF Connect button clicked:
    DEFINE serial_callback(packet):
        CALL data_manager.add_or_update_victim(packet)
        INCREMENT packet_count
        UPDATE last_packet_time
        CALL broadcast_packet(packet)  # WebSocket
        SET force_rerun = True

    DEFINE error_callback():
        INCREMENT error_count

    success = serial_reader.start_reading(
        port=session_state.serial_port,
        baudrate=session_state.baud_rate,
        on_packet_received=serial_callback,
        on_error=error_callback
    )

    IF success:
        SET serial_connected = True
        CALL st.rerun()
```

Function: `main()` **Purpose:** Main application orchestrator

Algorithm:

```
CALL render_sidebar() → selected_page

SWITCH selected_page:
    CASE "Dashboard":
        IMPORT _pages.dashboard
        CALL dashboard.render_dashboard()

    CASE "Analytics":
        IMPORT _pages.analytics
        CALL analytics.render_analytics()

    CASE "Export Data":
        IMPORT _pages.export
        CALL export.render_export()

    CASE "Settings":
        IMPORT _pages.settings
        CALL settings.render_settings()
```

Page Routing Table:

Route	Module	Primary Function
Dashboard	`_pages.dashboard`	`render_dashboard()`
Analytics	`_pages.analytics`	`render_analytics()`
Export Data	`_pages.export`	`render_export()`
Settings	`_pages.settings`	`render_settings()`

Geolocation Auto-Load Component **Purpose:** Load saved geolocation from browser localStorage

Implementation:

```
auto_geolocation_component = """
<script>
(function() {
    try {
        const userLat = parseFloat(localStorage.getItem('user_lat'));
        const userLon = parseFloat(localStorage.getItem('user_lon'));

        if (userLat && userLon && !isNaN(userLat) && !isNaN(userLon)) {
            sessionStorage.setItem('geo_lat', userLat.toString());
            sessionStorage.setItem('geo_lon', userLon.toString());
        }
    } catch(e) {
        // Geolocation not available
    }
})();
</script>
"""
```

Storage Flow:

```
Settings Page → User clicks "Detect Location" →
Browser Geolocation API →
localStorage.setItem('user_lat', lat) →
localStorage.setItem('user_lon', lon) →
```

```
app.py reads from localStorage →
Updates session_state.rescue_centre_lat/lon
```

4.2 config.py - Configuration Management

Purpose: Central configuration repository with environment variable integration

File Size: 320 lines
Configuration Categories: 10

Configuration Categories 1. API Keys

```
GOOGLE_MAPS_API_KEY = os.getenv('GOOGLE_MAPS_API_KEY', '')
```

- Loaded from `.env` file
- Used for Google Maps tile layers
- Validated on startup

2. Serial Port Settings

```
DEFAULT_SERIAL_PORT = None  # User must select
DEFAULT_BAUD_RATE = 115200
SERIAL_TIMEOUT = 1  # seconds
AVAILABLE_BAUD_RATES = [9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600]
```

Baud Rate Justification: - 115200: Standard for ESP32 (used by default) - Lower rates: Compatibility with older hardware - Higher rates: Faster data transfer for high-frequency operations

3. Map Settings

```
DEFAULT_MAP_CENTER_LAT = 13.022  # Bangalore
DEFAULT_MAP_CENTER_LON = 77.587
DEFAULT_MAP_ZOOM = 14

MAP_TILE_PROVIDERS = {
    'google_roadmap': 'https://mt1.google.com/vt/lyrs=m&x={x}&y={y}&z={z}',
    'google_satellite': 'https://mt1.google.com/vt/lyrs=s&x={x}&y={y}&z={z}',
    'google_hybrid': 'https://mt1.google.com/vt/lyrs=y&x={x}&y={y}&z={z}',
    'openstreetmap': 'https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png'
}
```

Tile Provider Selection: - Google Roadmap: Default for urban areas (clear labels) - Google Satellite: For terrain visualization - Google Hybrid: Combines satellite + labels - OpenStreetMap: Fallback if Google API unavailable

4. Priority Thresholds

```
# RSSI (Signal Strength) in dBm
RSSI_STRONG_THRESHOLD = -70  # Better than -70 dBm = Strong
RSSI_WEAK_THRESHOLD = -85    # Worse than -85 dBm = Weak

# Time thresholds in minutes
TIME_STALE_THRESHOLD = 20    # No update for 20min = Stale
TIME_CRITICAL_THRESHOLD = 15 # No update for 15min = Critical
```

RSSI Reference Scale: | RSSI (dBm) | Signal Quality | Typical Range | Priority Impact | |————|——————|——————|———————| | -30 to -50 | Excellent | <100m | Low priority | | -50 to -70 | Good | 100m-1km | Low priority | | -70 to -85 | Fair | 1km-5km | Medium priority | | -85 to -100 | Weak | 5km-10km | High priority | | <-100 | Very Weak | 10km-20km | Critical priority |

Time Threshold Rationale: - 15 minutes: Victim may be in distress (battery dying, moved to dangerous area) - 20 minutes: Data considered stale (may need re-verification)

5. Status Definitions

```
STATUS_STRANDED = "STRANDED"   # Initial detection
STATUS_EN_ROUTE = "EN_ROUTE"   # Rescue team dispatched
STATUS_RESCUED = "RESCUED"     # Victim recovered

ALL_STATUSES = [STATUS_STRANDED, STATUS_EN_ROUTE, STATUS_RESCUED]
```

State Machine:

```
STRANDED → EN_ROUTE → RESCUED
   ↑           ↓
          (Can revert if mission aborted)
```

6. Color Schemes

```
STATUS_COLORS = {
    STATUS_STRANDED: "#FF4445",  # Red (alert)
    STATUS_EN_ROUTE: "#FFA500",  # Orange (in progress)
    STATUS_RESCUED: "#00CC66"    # Green (success)
}

SIGNAL_COLORS = {
    'strong': "#00CC66",    # Green
    'medium': "#FFCC00",    # Yellow
    'weak': "#FF4444"       # Red
}

MARKER_COLORS = {
    STATUS_STRANDED: 'red',
    STATUS_EN_ROUTE: 'orange',
    STATUS_RESCUED: 'green'
}
```

Color Psychology: - Red: Urgency, danger, immediate action required - Orange: Caution, in progress, monitoring needed - Green: Success, safety, completion

7. Data Persistence

```
BACKUP_INTERVAL_SECONDS = 30
BACKUP_FILE_PATH = 'data/victims_backup.json'
RESCUE_LOG_PATH = 'data/rescue_log.csv'
```

Auto-Save Strategy: - Every 30 seconds: Balance between data safety and disk I/O - JSON format: Human-readable for manual recovery - CSV log: Append-only rescue operations for audit trail

8. Geographic Settings

```
SECTOR_SIZE_LAT = 0.001  # ~111 meters
SECTOR_SIZE_LON = 0.001  # ~111 meters at equator
```

Sector Grid Explanation: - Used for density heatmaps - 0.001° 111 meters at equator - Creates 100m x 100m grid cells for clustering analysis

9. Validation Rules

```
VALID_LAT_RANGE = (-90, 90)
VALID_LON_RANGE = (-180, 180)
VALID_RSSI_RANGE = (-150, -30)
MIN_VICTIM_ID = 1
MAX_VICTIM_ID = 9999
```

Validation Rationale: - Latitude: -90° (South Pole) to +90° (North Pole) - Longitude: -180° (Date Line West) to +180° (Date Line East) - RSSI: -150 dBm (theoretical minimum) to -30 dBm (very close range) - Victim ID: 4-digit range for human readability

10. Export Settings

```
EXPORT_COLUMNS = [
    'ID', 'LAT', 'LON', 'STATUS', 'RSSI',
    'FIRST_DETECTED', 'LAST_UPDATE', 'RESCUED_TIME',
    'RESCUED_BY', 'UPDATE_COUNT'
]

EXPORT_FILENAME_FORMAT = "{type}_victims_{timestamp}.csv"
```

4.3 modules/serial_reader.py - Serial Communication

Purpose: Background thread for continuous serial port monitoring and packet reception

File Size: ~250 lines
Design Pattern: Observer Pattern with callbacks

Class: SerialReader **Attributes:**

```python
self.serial_port = None           # pyserial.Serial instance
self.is_reading = False           # Thread control flag
self.read_thread = None           # Background thread
self.port = None                  # COM port name
self.baudrate = 115200            # Communication speed
self.on_packet_received = None    # Callback for valid packets
self.on_error = None              # Callback for errors
```

Method: `start_reading()` **Signature:**

```python
def start_reading(self, port: str, baudrate: int = 115200,
                  on_packet_received=None, on_error=None) -> bool
```

Parameters: - `port` (str): COM port identifier (e.g., "COM23", "/dev/ttyUSB0") - `baudrate` (int): Communication speed in bits per second - `on_packet_received` (callable): Callback invoked on valid packet - `on_error` (callable): Callback invoked on parse error

Returns: bool - True if successfully started, False otherwise

Algorithm:

```
FUNCTION start_reading(port, baudrate, callbacks):
    IF already reading:
        RETURN False

    TRY:
        CREATE serial.Serial instance:
            port = port
            baudrate = baudrate
            bytesize = 8
            parity = 'N'
            stopbits = 1
            timeout = 1 second

        IF serial port opened successfully:
            STORE callbacks
            SET is_reading = True
            CREATE new Thread(target=_read_loop)
            START thread as daemon
            RETURN True
        ELSE:
            RETURN False

    CATCH SerialException as e:
        PRINT error message
        RETURN False
```

Error Scenarios: - Port not available (already in use) - Invalid port name - Permission denied (Linux/Mac) - Device disconnected - Invalid baud rate

Method: `_read_loop()` (Background Thread) **Purpose:** Continuously read and process serial data

Algorithm:

```
FUNCTION _read_loop():
    WHILE is_reading:
        IF serial_port is None OR not open:
            SLEEP 0.1 seconds
            CONTINUE

        TRY:
            # Non-blocking read with timeout
            IF serial_port.in_waiting > 0:
                line = serial_port.readline()

                # Decode with fallback encodings
                TRY:
                    text = line.decode('utf-8').strip()
                CATCH UnicodeDecodeError:
                    TRY:
                        text = line.decode('latin-1').strip()
                    CATCH:
                        text = line.decode('ascii', errors='ignore').strip()

                IF text is empty:
                    CONTINUE

                # Parse JSON packet
                packet = _parse_packet(text)

                IF packet is None:
                    CONTINUE

                # Validate packet structure
                IF _validate_packet(packet):
                    IF on_packet_received callback exists:
                        CALL on_packet_received(packet)
                ELSE:
                    IF on_error callback exists:
                        CALL on_error()

        CATCH SerialException as e:
            PRINT "Serial error:" + str(e)
            CALL _attempt_reconnect()

        CATCH Exception as e:
            PRINT "Unexpected error:" + str(e)
            IF on_error callback exists:
                CALL on_error()
```

Performance Characteristics: - Read timeout: 1 second (prevents blocking) - Check interval: 0.1 seconds when no data - Processing speed: ~100 packets/second maximum - Memory per packet: ~200 bytes

Method: `_parse_packet()` **Purpose:** Convert raw string to JSON dictionary

Algorithm:

```
FUNCTION _parse_packet(text):
    # Remove whitespace
    text = text.strip()

    # Detect JSON braces
    IF text does not start with '{' OR not end with '}':
        RETURN None

    TRY:
        # Parse JSON
        packet = json.loads(text)
        RETURN packet

    CATCH json.JSONDecodeError:
        # Try to fix common issues
        text = text.replace("'", '"')  # Single quotes → double quotes
        text = text.replace("\n", "")  # Remove newlines

        TRY:
            packet = json.loads(text)
            RETURN packet
        CATCH:
            RETURN None
```

Common Packet Formats:

```
// Standard format
{"ID": 1001, "LAT": 13.022, "LON": 77.587, "RSSI": -75, "STATUS": "STRANDED"}

// With battery info
{"ID": 1001, "LAT": 13.022, "LON": 77.587, "RSSI": -75, "BATTERY": 85}

// Minimal format
{"ID": 1001, "LAT": 13.022, "LON": 77.587, "RSSI": -75}
```

Method: **`_validate_packet()`** **Purpose:** Verify packet structure and data ranges

Algorithm:

```
FUNCTION _validate_packet(packet):
    # Import validators
    FROM utils.validators IMPORT validate_packet

    # Delegate to comprehensive validator
    is_valid, error_message = validate_packet(packet)

    IF not is_valid:
        PRINT "Validation failed:" + error_message
        RETURN False

    RETURN True
```

Validation Checks (from utils/validators.py): 1. Required fields present (ID, LAT, LON, RSSI) 2. Data types correct (ID: int, LAT/LON/RSSI: float/int) 3. Value ranges (LAT: -90 to 90, LON: -180 to 180, RSSI: -150 to -30) 4. ID range (1 to 9999)

Method: **`stop_reading()`** **Purpose:** Gracefully stop serial reading thread

Algorithm:

```
FUNCTION stop_reading():
    # Signal thread to stop
    SET is_reading = False

    # Wait for thread to finish (max 2 seconds)
    IF read_thread is not None AND read_thread.is_alive():
        read_thread.join(timeout=2.0)

    # Close serial port
    IF serial_port is not None AND serial_port.is_open:
        CALL serial_port.close()
        SET serial_port = None
```

Method: `_attempt_reconnect()` **Purpose:** Automatic reconnection on serial port errors

Algorithm:

```
FUNCTION _attempt_reconnect():
    PRINT "Attempting to reconnect..."

    # Close existing connection
    IF serial_port is not None:
        TRY:
            CALL serial_port.close()
        CATCH:
            PASS

    # Wait before retry
    SLEEP 2 seconds

    # Try to reopen
    TRY:
        CREATE new serial.Serial instance
        IF successful:
            PRINT "Reconnected successfully"
            RETURN True
    CATCH:
        PRINT "Reconnection failed"
        RETURN False
```

Reconnection Strategy: - Max retries: Infinite (until manually stopped) - Retry interval: 2 seconds - Exponential backoff: No (fixed interval for predictability)

Thread Safety Considerations: - `is_reading` flag: Atomic boolean, no lock needed - Serial port access: Single thread reader, no concurrent writes - Callbacks: Invoked from serial thread, must be thread-safe - Session state updates: Protected by Streamlit's rerun mechanism

---

4.4 modules/data_manager.py - Data Management

Purpose: In-memory victim database with CRUD operations and persistence

File Size: ~400 lines
Design Pattern: Repository Pattern

Class: DataManager Attributes:

```
self.victims = {}  # Dictionary: {victim_id: victim_record}
self.backup_file = config.BACKUP_FILE_PATH
self.last_backup_time = datetime.now()
self.backup_interval = config.BACKUP_INTERVAL_SECONDS
```

Victim Record Structure Complete Field Specification:

```
victim_record = {
    'ID': int,                # Unique identifier (1-9999)
    'LAT': float,             # Latitude (-90 to 90)
    'LON': float,             # Longitude (-180 to 180)
    'STATUS': str,            # STRANDED / EN_ROUTE / RESCUED
    'RSSI': int,              # Signal strength in dBm (-150 to -30)
    'RSSI_HISTORY': list,     # Last 20 RSSI values [int, ...]
    'FIRST_DETECTED': str,    # ISO datetime: "2025-12-24T10:30:45"
    'LAST_UPDATE': str,       # ISO datetime: "2025-12-24T10:35:20"
    'UPDATE_COUNT': int,      # Number of packets received
    'RESCUED_TIME': str,      # ISO datetime or None
    'RESCUED_BY': str,        # Operator name or None
    'NOTES': str,             # Operator notes
    'BATTERY': int,           # Battery percentage (0-100) or None
}
```

Field Details:

Field	Type	Required	Default	Purpose
ID	int	Yes	-	Unique victim identifier
LAT	float	Yes	-	GPS latitude coordinate
LON	float	Yes	-	GPS longitude coordinate
STATUS	str	Yes	"STRANDED"	Current rescue status
RSSI	int	Yes	-	Most recent signal strength
RSSI_HISTORY	list	No	[]	Signal trend analysis
FIRST_DETECTED	str	Auto	now()	Initial detection timestamp
LAST_UPDATE	str	Auto	now()	Most recent packet timestamp
UPDATE_COUNT	int	Auto	1	Packet counter
RESCUED_TIME	str	Auto	None	Rescue completion timestamp
RESCUED_BY	str	Manual	None	Operator who rescued
NOTES	str	Manual	""	Free-form operator notes
BATTERY	int	Optional	None	Victim device battery level

Method: `add_or_update_victim()` Signature:

```
def add_or_update_victim(self, packet: dict) -> bool
```

Purpose: Add new victim or update existing victim from incoming packet

Algorithm:

```
FUNCTION add_or_update_victim(packet):
    victim_id = packet['ID']
    current_time = datetime.now().isoformat()

    IF victim_id EXISTS in self.victims:
        # UPDATE EXISTING VICTIM
        victim = self.victims[victim_id]

        # Update coordinates (may have moved)
        victim['LAT'] = packet['LAT']
        victim['LON'] = packet['LON']

        # Update signal strength
        old_rssi = victim['RSSI']
        new_rssi = packet['RSSI']
        victim['RSSI'] = new_rssi

        # Maintain RSSI history (last 20 values)
        IF 'RSSI_HISTORY' not in victim:
            victim['RSSI_HISTORY'] = []

        victim['RSSI_HISTORY'].append(new_rssi)

        IF length(victim['RSSI_HISTORY']) > 20:
            victim['RSSI_HISTORY'] = victim['RSSI_HISTORY'][-20:]

        # Update timestamp and counter
        victim['LAST_UPDATE'] = current_time
        victim['UPDATE_COUNT'] += 1

        # Update battery if present
        IF 'BATTERY' in packet:
            victim['BATTERY'] = packet['BATTERY']

    ELSE:
        # CREATE NEW VICTIM
        victim = {
            'ID': victim_id,
            'LAT': packet['LAT'],
            'LON': packet['LON'],
            'STATUS': packet.get('STATUS', 'STRANDED'),
            'RSSI': packet['RSSI'],
            'RSSI_HISTORY': [packet['RSSI']],
            'FIRST_DETECTED': current_time,
            'LAST_UPDATE': current_time,
            'UPDATE_COUNT': 1,
            'RESCUED_TIME': None,
            'RESCUED_BY': None,
            'NOTES': '',
            'BATTERY': packet.get('BATTERY', None)
        }

        self.victims[victim_id] = victim

    # Check if auto-save needed
    time_since_backup = (datetime.now() - self.last_backup_time).total_seconds()

    IF time_since_backup >= self.backup_interval:
        CALL _auto_save()

    RETURN True
```

Update Logic: - **Coordinates:** Always update (victim may have moved) - **RSSI:** Always update (signal strength changes) - **Status:** Only update if explicitly provided in packet - **Battery:** Update if present in packet - **Timestamps:** LAST_UPDATE always updated, FIRST_DETECTED preserved

RSSI History Management: - Max length: 20 values - Purpose: Signal trend detection - Storage: Ring buffer (oldest dropped when full) - Usage: `analyze_signal_trends()` in analytics

Method: `mark_enroute()` **Signature:**

```
def mark_enroute(self, victim_id: int, operator_name: str) -> bool
```

Purpose: Mark victim as having rescue team en route

Algorithm:

```
FUNCTION mark_enroute(victim_id, operator_name):
    IF victim_id NOT in self.victims:
        RETURN False

    victim = self.victims[victim_id]

    # Update status
    victim['STATUS'] = config.STATUS_EN_ROUTE
    victim['LAST_UPDATE'] = datetime.now().isoformat()
    # Add audit trail note
    timestamp = datetime.now().strftime('%H:%M:%S')
    note = f"[{timestamp}] {operator_name}: Rescue team dispatched"

    IF victim['NOTES']:
        victim['NOTES'] += "\n" + note
    ELSE:
        victim['NOTES'] = note

    # Trigger auto-save
    CALL _auto_save()

    RETURN True
```

State Transition:

```
STRANDED → EN_ROUTE
```

Audit Trail Format:

```
[10:30:45] Operator-01: Rescue team dispatched
[10:45:12] Operator-01: Team reports delay - flooding
[11:15:30] Operator-02: Victim rescued successfully
```

Method: `mark_rescued()` **Signature:**

```python
def mark_rescued(self, victim_id: int, operator_name: str) -> bool
```

Purpose: Mark victim as successfully rescued

Algorithm:

```
FUNCTION mark_rescued(victim_id, operator_name):
    IF victim_id NOT in self.victims:
        RETURN False

    victim = self.victims[victim_id]
    current_time = datetime.now()

    # Update status
    victim['STATUS'] = config.STATUS_RESCUED
    victim['RESCUED_TIME'] = current_time.isoformat()
    victim['RESCUED_BY'] = operator_name
    victim['LAST_UPDATE'] = current_time.isoformat()

    # Add audit trail note
    timestamp = current_time.strftime('%H:%M:%S')
    note = f"[{timestamp}] {operator_name}: Victim rescued"
    IF victim['NOTES']:
        victim['NOTES'] += "\n" + note
    ELSE:
        victim['NOTES'] = note

    # Calculate rescue duration
    first_detected = datetime.fromisoformat(victim['FIRST_DETECTED'])
    duration = (current_time - first_detected).total_seconds() / 60

    duration_note = f"Rescue duration: {duration:.1f} minutes"
    victim['NOTES'] += "\n" + duration_note

    # Log to CSV file
    CALL _log_rescue_operation(victim)

    # Trigger auto-save
    CALL _auto_save()

    RETURN True
```

State Transition:

```
EN_ROUTE → RESCUED
```

Rescue Log CSV Format:

```
ID,LAT,LON,FIRST_DETECTED,RESCUED_TIME,DURATION_MIN,RESCUED_BY,RSSI,BATTERY
1001,13.022,77.587,2025-12-24 10:30:45,2025-12-24 11:15:30,44.75,Operator-01,-75,65
```

Method: `get_statistics()` **Signature:**

```
def get_statistics(self) -> dict
```

Purpose: Calculate real-time statistics for dashboard

Returns:

```
{
    'total': int,       # Total victims tracked
    'stranded': int,    # Currently stranded
    'enroute': int,     # Rescue in progress
    'rescued': int,     # Successfully rescued
    'active': int,      # Stranded + En-route
    'success_rate': float  # Rescued / Total * 100
}
```

Algorithm:

```
FUNCTION get_statistics():
    total = length(self.victims)
    stranded = 0
    enroute = 0
    rescued = 0

    FOR each victim in self.victims.values():
        SWITCH victim['STATUS']:
            CASE 'STRANDED':
                stranded += 1
            CASE 'EN_ROUTE':
                enroute += 1
            CASE 'RESCUED':
                rescued += 1

    active = stranded + enroute

    IF total > 0:
        success_rate = (rescued / total) * 100
    ELSE:
        success_rate = 0.0

    RETURN {
        'total': total,
        'stranded': stranded,
        'enroute': enroute,
        'rescued': rescued,
        'active': active,
        'success_rate': success_rate
    }
```

Method: `_auto_save()` **Purpose:** Periodic backup to JSON file

Algorithm:

```
FUNCTION _auto_save():
    TRY:
        # Create backup directory if needed
        os.makedirs(os.path.dirname(self.backup_file), exist_ok=True)

        # Write to temporary file first (atomic write)
        temp_file = self.backup_file + '.tmp'

        WITH open(temp_file, 'w') as f:
            json.dump(self.victims, f, indent=2, default=str)

        # Rename temp file to actual backup (atomic operation)
        os.replace(temp_file, self.backup_file)
        # Update backup timestamp
        self.last_backup_time = datetime.now()

        RETURN True

    CATCH Exception as e:
        PRINT "Auto-save failed:" + str(e)
        RETURN False
```

Atomic Write Strategy: 1. Write to temp file (.tmp extension) 2. If write successful, rename temp → actual file 3. Rename is atomic operation (prevents corruption) 4. If crash during write, original file intact

Backup Frequency: - Interval: 30 seconds (configurable) - Trigger: After any update that changes data - Manual: Can be triggered from Settings page

Method: `export_to_csv()` Signature:

```python
def export_to_csv(self, filename: str, include_rescued: bool = True) -> bool
```

Purpose: Export victim data to CSV file

Algorithm:

```
FUNCTION export_to_csv(filename, include_rescued):
    TRY:
        # Filter victims based on include_rescued flag
        victims_to_export = []

        FOR each victim in self.victims.values():
            IF include_rescued OR victim['STATUS'] != 'RESCUED':
                victims_to_export.append(victim)

        # Convert to pandas DataFrame
        df = pd.DataFrame(victims_to_export)

        # Select and order columns
        columns = config.EXPORT_COLUMNS
        df = df[columns]

        # Write to CSV
        df.to_csv(filename, index=False)

        RETURN True

    CATCH Exception as e:
        PRINT "CSV export failed:" + str(e)
        RETURN False
```

4.5 modules/map_manager.py - Map Visualization

Purpose: Generate interactive Folium maps with victim markers

File Size: ~350 lines
Design Pattern: Builder Pattern

Class: MapManager **Attributes:**

```python
self.config = config  # Configuration reference
```

Method: `create_victim_map()` **Signature:**

```python
def create_victim_map(self, victims: dict, center: list, zoom: int,
                      show_rescued: bool = True,
                      show_priority_only: bool = False,
                      rssi_strong_threshold: int = -70,
                      rssi_weak_threshold: int = -85,
                      time_critical_threshold: int = 15) -> folium.Map
```

Purpose: Create interactive map with victim markers

Parameters: - `victims` (dict): Victim records from DataManager - `center` (list): [lat, lon] map center coordinates - `zoom` (int): Initial zoom level (1-18) - `show_rescued` (bool): Include rescued victims on map - `show_priority_only` (bool): Show only high-priority victims - `rssi_strong_threshold` (int): Strong signal cutoff (dBm) - `rssi_weak_threshold` (int): Weak signal cutoff (dBm) - `time_critical_threshold` (int): Critical time threshold (minutes)

Returns: folium.Map object

Algorithm:

```
FUNCTION create_victim_map(victims, center, zoom, filters, thresholds):
    # Create base map
    map_object = folium.Map(
        location=center,
        zoom_start=zoom,
        tiles=None  # Add custom tiles later
    )

    # Add Google Maps tile layer
    folium.TileLayer(
        tiles=config.MAP_TILE_PROVIDERS['google_roadmap'],
        attr='Google Maps',
        name='Roadmap',
        overlay=False,
        control=True
    ).add_to(map_object)

    # Add satellite layer
    folium.TileLayer(
        tiles=config.MAP_TILE_PROVIDERS['google_satellite'],
        attr='Google Satellite',
        name='Satellite',
        overlay=False,
        control=True
    ).add_to(map_object)

    # Add layer control
    folium.LayerControl().add_to(map_object)

    # Process each victim
    FOR each victim in victims.values():
        # Apply filters
        IF not show_rescued AND victim['STATUS'] == 'RESCUED':
            CONTINUE

        # Calculate priority
        priority = calculate_priority(
            victim,
            rssi_weak_threshold,
            time_critical_threshold
        )

        IF show_priority_only AND priority != 'HIGH':
            CONTINUE

        # Determine marker properties
        marker_color, icon_name, opacity = _get_marker_properties(
            victim,
            priority
        )

        # Create popup HTML
        popup_html = _create_popup_html(victim, priority)

        # Create marker
        marker = folium.Marker(
            location=[victim['LAT'], victim['LON']],
            popup=folium.Popup(popup_html, max_width=300),
            tooltip=f"Victim {victim['ID']} - {victim['STATUS']}",
            icon=folium.Icon(
                color=marker_color,
                icon=icon_name,
                prefix='fa',
                opacity=opacity
            )
        )

        marker.add_to(map_object)

    # Add legend
    legend_html = _create_legend_html(
        rssi_strong_threshold,
        rssi_weak_threshold,
        time_critical_threshold
    )

    map_object.get_root().html.add_child(folium.Element(legend_html))

    RETURN map_object
```

Method: `_get_marker_properties()` **Purpose:** Determine marker color, icon, and opacity based on status and priority

Algorithm:

```
FUNCTION _get_marker_properties(victim, priority):
    status = victim['STATUS']

    # Base color from status
    SWITCH status:
        CASE 'STRANDED':
            base_color = 'red'
            icon = 'exclamation-circle'
        CASE 'EN_ROUTE':
            base_color = 'orange'
            icon = 'ambulance'
        CASE 'RESCUED':
            base_color = 'green'
            icon = 'check-circle'

    # Modify for high priority
    IF priority == 'HIGH':
        opacity = 1.0  # Full opacity
        # Add pulsing animation class (via CSS)
    ELSE IF priority == 'MEDIUM':
        opacity = 0.8
    ELSE:
        opacity = 0.6

    RETURN base_color, icon, opacity
```

Font Awesome Icons Used: - `exclamation-circle`: Stranded victims (alert) - `ambulance`: En-route (rescue in progress) - `check-circle`: Rescued (success)

Method: `_create_popup_html()` **Purpose:** Generate HTML content for marker popups

Algorithm:

```
FUNCTION _create_popup_html(victim, priority):
    # Extract data
    victim_id = victim['ID']
    lat = victim['LAT']
    lon = victim['LON']
    status = victim['STATUS']
    rssi = victim['RSSI']
    first_detected = victim['FIRST_DETECTED']
    last_update = victim['LAST_UPDATE']
    battery = victim.get('BATTERY', 'N/A')

    # Calculate time since last update
    last_update_dt = datetime.fromisoformat(last_update)
    time_ago = format_time_ago(last_update_dt)

    # Get signal quality indicator
    signal_quality, signal_color = get_signal_indicator(rssi)

    # Build HTML
    html = f"""
    <div style="font-family: Arial; min-width: 250px;">
        <h4 style="margin: 0 0 10px 0; color: #333;">
            Victim #{victim_id}
        </h4>

        <table style="width: 100%; font-size: 13px;">
            <tr>
                <td><strong>Status:</strong></td>
                <td style="color: {config.STATUS_COLORS[status]}">
                    {status}
                </td>
            </tr>
            <tr>
```

```
                <td><strong>Priority:</strong></td>
                <td style="color: {'red' if priority=='HIGH' else 'orange' if priority=='MEI
                    {priority}
                </td>
            </tr>
            <tr>
                <td><strong>Location:</strong></td>
                <td>{lat:.6f}, {lon:.6f}</td>
            </tr>
            <tr>
                <td><strong>Signal:</strong></td>
                <td style="color: {signal_color}">
                    {rssi} dBm ({signal_quality})
                </td>
            </tr>
            <tr>
                <td><strong>Battery:</strong></td>
                <td>{battery}%</td>
            </tr>
            <tr>
                <td><strong>First Seen:</strong></td>
                <td>{first_detected}</td>
            </tr>
            <tr>
                <td><strong>Last Update:</strong></td>
                <td>{time_ago}</td>
            </tr>
        </table>

        {_get_action_buttons_html(victim_id, status)}
    </div>
    """

    RETURN html
```

Popup Features: - Color-coded status and priority - Formatted coordinates (6 decimal places) - Signal strength with quality indicator - Battery percentage (if available) - Timestamps in human-readable format - Action buttons (context-specific)

Method: `_create_legend_html()` **Purpose:** Create map legend showing priority thresholds

Algorithm:

```
FUNCTION _create_legend_html(rssi_strong, rssi_weak, time_critical):
    html = f"""
```

```
<div style="
    position: fixed;
    bottom: 50px;
    left: 50px;
    width: 200px;
    background: white;
    border: 2px solid #ccc;
    border-radius: 5px;
    padding: 10px;
    z-index: 9999;
    font-family: Arial;
    font-size: 12px;
">
    <h4 style="margin: 0 0 10px 0;">Priority Legend</h4>

    <div style="margin-bottom: 5px;">
        <span style="color: red;"> </span>
        <strong>HIGH:</strong> RSSI < {rssi_weak} dBm
        OR no update > {time_critical} min
    </div>

    <div style="margin-bottom: 5px;">
        <span style="color: orange;"> </span>
        <strong>MEDIUM:</strong> RSSI {rssi_weak} to {rssi_strong} dBm
    </div>

    <div>
        <span style="color: green;"> </span>
        <strong>LOW:</strong> RSSI > {rssi_strong} dBm
    </div>

    <hr style="margin: 10px 0;">

    <div style="font-size: 11px; color: #666;">
        <div><strong>Status Colors:</strong></div>
        <div style="color: red;">  Stranded</div>
        <div style="color: orange;">  En-Route</div>
        <div style="color: green;">  Rescued</div>
    </div>
</div>
"""

RETURN html
```

5. MATHEMATICAL FOUNDATIONS

5.1 Geographic Distance Calculation (Haversine Formula)

Purpose: Calculate great-circle distance between two GPS coordinates on Earth's surface

Mathematical Formula:

```
Given two points:
  Point 1: (lat , lon )
  Point 2: (lat , lon )

Haversine Formula:
  a = sin²(Δ /2) + cos( ) × cos( ) × sin²(Δ /2)
  c = 2 × atan2(√a, √(1-a))
  d = R × c

Where:
    ,   = latitude of points 1 and 2 (in radians)
  Δ  =   -
  Δ  =   -   (difference in longitude)
  R = Earth's radius = 6371 km
  d = distance between points (in kilometers)
```

Implementation in Python:

```python
import math

def calculate_distance(lat1, lon1, lat2, lon2):
    """
    Calculate distance between two GPS coordinates using Haversine formula

    Parameters:
        lat1, lon1: Coordinates of point 1 (decimal degrees)
        lat2, lon2: Coordinates of point 2 (decimal degrees)

    Returns:
        Distance in kilometers (float)
    """
    # Earth's radius in kilometers
    R = 6371.0

    # Convert degrees to radians
     1 = math.radians(lat1)
     2 = math.radians(lat2)
    Δ  = math.radians(lat2 - lat1)
    Δ  = math.radians(lon2 - lon1)
    # Haversine formula
    a = math.sin(Δ /2)**2 + math.cos( 1) * math.cos( 2) * math.sin(Δ /2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))

    # Distance in kilometers
    distance = R * c

    return distance
```

Usage in Application: - Coverage area calculation (distance from rescue station to victims) - Proximity sorting (nearest victims first) - Sector-based clustering validation

Accuracy: - Precision: ±0.5% for distances < 1000 km - Earth approximation: Spherical (ignores ellipsoid, adequate for rescue operations) - Valid range: Any two points on Earth

Example Calculation:

```
Rescue Station: (13.022, 77.587) - Bangalore
Victim Location: (13.045, 77.620)

Δ  = 0.023° = 0.000401 rad
Δ  = 0.033° = 0.000576 rad

a = sin²(0.0002005) + cos(13.022°) × cos(13.045°) × sin²(0.000288)
a   0.0000001602 + 0.9487 × 0.9486 × 0.0000000829
a   0.0000001602 + 0.0000000746
a   0.0000002348

c = 2 × atan2(√0.0000002348, √0.9999997652)
c   2 × atan2(0.000485, 0.999999)
c   2 × 0.000485
c   0.00097 radians

d = 6371 × 0.00097   6.18 km
```

5.2 Priority Scoring Algorithm

Purpose: Assign priority levels (HIGH/MEDIUM/LOW) based on signal strength and time since last update

Mathematical Model:

```
Priority Score = f(RSSI, Δt)

Where:
  RSSI = Received Signal Strength Indicator (dBm)
  Δt = Time since last update (minutes)

Decision Tree:
  IF RSSI < RSSI_weak_threshold OR Δt > Time_critical_threshold:
      Priority = HIGH
  ELSE IF RSSI_weak_threshold   RSSI   RSSI_strong_threshold:
      Priority = MEDIUM
  ELSE:
      Priority = LOW

Default Thresholds:
  RSSI_strong_threshold = -70 dBm
  RSSI_weak_threshold = -85 dBm
  Time_critical_threshold = 15 minutes
```

Implementation:

```python
def calculate_priority(victim, rssi_weak_threshold=-85,
                      time_critical_threshold=15):
    """
    Calculate victim priority based on signal strength and staleness

    Parameters:
        victim (dict): Victim record with RSSI and LAST_UPDATE
        rssi_weak_threshold (int): Weak signal cutoff (dBm)
        time_critical_threshold (int): Critical time cutoff (minutes)

    Returns:
        str: 'HIGH', 'MEDIUM', or 'LOW'
    """
    # Extract RSSI
    rssi = victim['RSSI']

    # Calculate time since last update
    last_update = datetime.fromisoformat(victim['LAST_UPDATE'])
    now = datetime.now()
    time_delta = (now - last_update).total_seconds() / 60  # Convert to minutes

    # Priority logic
    if rssi < rssi_weak_threshold or time_delta > time_critical_threshold:
        return 'HIGH'
    elif rssi < -70:  # Between weak and strong thresholds
        return 'MEDIUM'
    else:
        return 'LOW'
```

Rationale:

RSSI-Based Priority: - **Strong Signal (RSSI > -70 dBm):** Victim close to ground station, good communication, LOW priority - **Medium Signal (-85 to -70 dBm):** Victim at moderate distance, MEDIUM priority - **Weak Signal (RSSI < -85 dBm):** Victim far away or signal obstructed, HIGH priority (may lose contact)

Time-Based Priority: - **Recent Update (< 15 min):** Device actively transmitting, LOW/MEDIUM priority - **Stale Update (> 15 min):** Device may have failed, victim in danger, HIGH priority

Combined Logic:

```
Truth Table:

 RSSI          Time Δt       Priority

 Strong        Recent        LOW
 Medium        Recent        MEDIUM
 Weak          Recent        HIGH
 Strong        Stale         HIGH
 Medium        Stale         HIGH
 Weak          Stale         HIGH
```

5.3 Signal Strength Interpretation

RSSI to Distance Approximation:

```
Path Loss Model (Free Space):
  RSSI = P_tx - PL(d)

Where:
  P_tx = Transmission power (dBm)
  PL(d) = Path loss at distance d

Path Loss Formula:
  PL(d) = 20 × log (d) + 20 × log (f) + 32.45

Where:
  d = distance in kilometers
  f = frequency in MHz

For LoRa at 868 MHz with 22 dBm TX power:
  RSSI  22 - (20×log (d) + 20×log (868) + 32.45)
  RSSI  22 - (20×log (d) + 59.01 + 32.45)
  RSSI  -69.46 - 20×log (d)
```

Approximate Distance Table:

RSSI (dBm)	Estimated Distance	Signal Quality	Priority
-30 to -50	0.01 - 0.1 km	Excellent	LOW
-50 to -70	0.1 - 1 km	Good	LOW
-70 to -85	1 - 5 km	Fair	MEDIUM
-85 to -100	5 - 10 km	Weak	HIGH
-100 to -120	10 - 20 km	Very Weak	HIGH
$<$ -120	$>$ 20 km	Critical	HIGH

Note: Actual distances vary due to terrain, obstacles, weather

5.4 Rescue Efficiency Scoring

Purpose: Quantify operation effectiveness for analytics

Formula:

```
Efficiency Score = (W  × RescueRate + W  × SpeedScore) / (W  + W )

Where:
  RescueRate = (Rescued / Total) × 100

  SpeedScore = 100 × (1 - AvgRescueTime / MaxAcceptableTime)

  W  = Weight for rescue rate = 0.6
  W  = Weight for speed = 0.4
  MaxAcceptableTime = 60 minutes (threshold)

Grade Assignment:
  Score   90: Excellent
  Score   80: Good
  Score   70: Satisfactory
  Score   60: Needs Improvement
  Score < 60: Poor
```

Implementation:

```python
def calculate_rescue_efficiency(rescued_count, total_count, avg_rescue_time_minutes):
    """
    Calculate operation efficiency score

    Parameters:
        rescued_count (int): Number of rescued victims
        total_count (int): Total number of victims
        avg_rescue_time_minutes (float): Average time to rescue (minutes)

    Returns:
        tuple: (efficiency_score, grade)
    """
    if total_count == 0:
        return 0.0, "N/A"

    # Calculate rescue rate
    rescue_rate = (rescued_count / total_count) * 100

    # Calculate speed score (normalized against 60-minute threshold)
    max_acceptable_time = 60
    speed_score = 100 * (1 - min(avg_rescue_time_minutes, max_acceptable_time) / max_accepta
    speed_score = max(0, speed_score)  # Clamp to 0 minimum

    # Weighted average
    w1, w2 = 0.6, 0.4
    efficiency_score = (w1 * rescue_rate + w2 * speed_score) / (w1 + w2)

    # Assign grade
    if efficiency_score >= 90:
        grade = "Excellent"
    elif efficiency_score >= 80:
        grade = "Good"
    elif efficiency_score >= 70:
        grade = "Satisfactory"
    elif efficiency_score >= 60:
        grade = "Needs Improvement"
    else:
        grade = "Poor"

    return efficiency_score, grade
```

Example Calculation:

```
Scenario:
  Total victims: 20
  Rescued: 18
  Average rescue time: 35 minutes

Calculations:
  RescueRate = (18 / 20) × 100 = 90%

  SpeedScore = 100 × (1 - 35/60) = 100 × 0.417 = 41.7

  EfficiencyScore = (0.6 × 90 + 0.4 × 41.7) / 1.0
                  = (54 + 16.68) / 1.0
                  = 70.68
  Grade = "Satisfactory"
```

5.5 Geographic Clustering (Grid-Based)

Purpose: Identify victim density hotspots for resource allocation

Algorithm:

```
Grid-Based Spatial Clustering:

  1. Define grid cell size:
     Cell_lat = 0.001°   111 meters
     Cell_lon = 0.001°   111 meters (at equator)

  2. Calculate grid cell for each victim:
     Grid_x = floor(victim_lon / Cell_lon)
     Grid_y = floor(victim_lat / Cell_lat)
     Cell_ID = (Grid_x, Grid_y)

  3. Count victims per cell:
     Density[Cell_ID] = number of victims in cell

  4. Identify hotspots:
     Hotspot = Cell where Density   Threshold
```

Implementation:

```python
def analyze_geographic_density(victims, cell_size=0.001):
    """
    Cluster victims into geographic grid cells

    Parameters:
        victims (dict): Victim records
        cell_size (float): Grid cell size in degrees

    Returns:
        dict: {(grid_x, grid_y): victim_count}
    """
    density_map = {}

    for victim in victims.values():
        if victim['STATUS'] == 'RESCUED':
            continue  # Skip rescued victims

        lat = victim['LAT']
        lon = victim['LON']

        # Calculate grid cell
        grid_x = int(lon / cell_size)
        grid_y = int(lat / cell_size)
        cell_id = (grid_x, grid_y)

        # Increment density counter
        if cell_id not in density_map:
            density_map[cell_id] = []

        density_map[cell_id].append(victim['ID'])

    # Convert to counts
    density_counts = {cell: len(victims) for cell, victims in density_map.items()}

    return density_counts
```

Heatmap Generation:

```
def create_density_heatmap(density_map):
    """
    Convert density map to heatmap coordinates

    Parameters:
        density_map (dict): {(grid_x, grid_y): count}

    Returns:
        list: [[lat, lon, weight], ...]
    """
    heatmap_data = []

    for (grid_x, grid_y), count in density_map.items():
        # Convert grid cell back to coordinates (cell center)
        lon = (grid_x + 0.5) * 0.001
        lat = (grid_y + 0.5) * 0.001

        # Weight = victim count
        heatmap_data.append([lat, lon, count])

    return heatmap_data
```

5.6 Signal Trend Analysis

Purpose: Detect signal deterioration indicating victim movement or device failure

Statistical Method:

```
Linear Regression on RSSI History:
  Given RSSI history: [r , r , r , ..., r ]
  Time points: [t , t , t , ..., t ]

  Fit linear model: RSSI = m × t + b

  Where:
    m = slope (signal change rate)
    b = intercept (initial signal)

  Slope calculation (Least Squares):
    m = (n (t r ) -  t r ) / (n t ² - ( t )²)

  Trend interpretation:
    m < -2 dBm/update: Deteriorating (victim moving away)
    -2   m   2: Stable
    m > 2 dBm/update: Improving (victim approaching)
```

Implementation:

```python
import numpy as np

def analyze_signal_trend(rssi_history):
    """
    Analyze RSSI trend using linear regression

    Parameters:
        rssi_history (list): Historical RSSI values

    Returns:
        tuple: (trend_direction, slope, confidence)
    """
    if len(rssi_history) < 3:
        return "INSUFFICIENT_DATA", 0.0, 0.0

    # Time points (equally spaced)
    n = len(rssi_history)
    time_points = np.arange(n)
    rssi_array = np.array(rssi_history)

    # Calculate slope using numpy polyfit (degree 1 = linear)
    coefficients = np.polyfit(time_points, rssi_array, 1)
    slope = coefficients[0]

    # Calculate R² (coefficient of determination)
    p = np.poly1d(coefficients)
    y_predicted = p(time_points)
    ss_res = np.sum((rssi_array - y_predicted) ** 2)
    ss_tot = np.sum((rssi_array - np.mean(rssi_array)) ** 2)

    if ss_tot == 0:
        r_squared = 0.0
    else:
        r_squared = 1 - (ss_res / ss_tot)

    # Determine trend direction
    if slope < -2:
        trend = "DETERIORATING"
    elif slope > 2:
        trend = "IMPROVING"
    else:
        trend = "STABLE"

    confidence = r_squared * 100  # Convert to percentage

    return trend, slope, confidence
```

Example Analysis:

```
RSSI History: [-70, -72, -75, -78, -82, -85]

Linear regression:
  Slope (m) = -3.0 dBm/update
  R² = 0.98 (98% confidence)

Interpretation:
  Trend: DETERIORATING
  Rate: -3 dBm per update
  Confidence: 98%

Action: Flag as HIGH priority - signal weakening rapidly
```

---

6. ALGORITHM SPECIFICATIONS

6.1 Real-Time Update Mechanism

Purpose: Ensure UI updates immediately when new packets arrive

Challenge: Streamlit reruns entire script on state changes, but serial reading is in background thread

Solution: Force Rerun Pattern

Algorithm:

```
BACKGROUND THREAD (Serial Reader):
  WHILE is_reading:
      packet = read_serial_port()

      IF packet is valid:
          # Update data
          data_manager.add_or_update_victim(packet)

          # Set rerun flag (atomic operation)
          st.session_state.force_rerun = True

          # Update counters
          st.session_state.packet_count += 1
          st.session_state.last_packet_time = now()

MAIN THREAD (Streamlit):
  # Check at start of every page render
  IF st.session_state.force_rerun == True:
      # Clear flag
      st.session_state.force_rerun = False

      # Force immediate rerun
      st.rerun()

  # Continue normal page rendering...
```

Timing Diagram:

```
Time    Serial Thread          Main Thread (UI)

t=0     Read packet           Rendering page
t=1     Validate packet       ...
t=2     Update data           ...
t=3     Set force_rerun=True  ...
t=4     ...                   Check force_rerun
t=5     ...                   Detect True
t=6     ...                   Call st.rerun()
t=7     ...                   Start new render
t=8     ...                   Clear force_rerun
t=9     ...                   Render updated UI
```

Latency Analysis:

- Packet arrival to flag set: 5-10ms
- Flag detection to rerun trigger: 50-100ms (depends on current render)
- Rerun to updated UI: 200-500ms (full page render)
- **Total latency: ~300-600ms** (acceptable for disaster response)

6.2 Duplicate Prevention Algorithm

Purpose: Prevent duplicate victim records when same ID transmits multiple times

Strategy: ID-based deduplication with update logic

Algorithm:

```
FUNCTION add_or_update_victim(packet):
    victim_id = packet['ID']

    # Check if victim already exists
    IF victim_id IN victims_database:
        # UPDATE EXISTING
        existing_victim = victims_database[victim_id]

        # Update mutable fields
        existing_victim['LAT'] = packet['LAT']
        existing_victim['LON'] = packet['LON']
        existing_victim['RSSI'] = packet['RSSI']
        existing_victim['LAST_UPDATE'] = now()
        existing_victim['UPDATE_COUNT'] += 1

        # Append to RSSI history
        existing_victim['RSSI_HISTORY'].append(packet['RSSI'])

        # Keep only last 20 values
        IF length(existing_victim['RSSI_HISTORY']) > 20:
            existing_victim['RSSI_HISTORY'] = existing_victim['RSSI_HISTORY'][-20:]

        # Preserve immutable fields
        # (FIRST_DETECTED, initial status, etc. remain unchanged)

    ELSE:
        # CREATE NEW
        new_victim = {
            'ID': victim_id,
            'LAT': packet['LAT'],
            'LON': packet['LON'],
            'RSSI': packet['RSSI'],
            'RSSI_HISTORY': [packet['RSSI']],
            'STATUS': 'STRANDED',
            'FIRST_DETECTED': now(),
            'LAST_UPDATE': now(),
            'UPDATE_COUNT': 1,
            'RESCUED_TIME': None,
            'RESCUED_BY': None,
            'NOTES': '',
            'BATTERY': packet.get('BATTERY', None)
        }

        victims_database[victim_id] = new_victim
```

Key Design Decisions:

1. **ID as Primary Key:** Unique identifier prevents duplicates
2. **Update Strategy:** Overwrite mutable fields, preserve immutable
3. **RSSI History:** Ring buffer of last 20 values for trend analysis
4. **Timestamps:** FIRST_DETECTED preserved, LAST_UPDATE refreshed
5. **Counter:** UPDATE_COUNT tracks packet frequency

6.3 Auto-Save with Atomic Writes

Purpose: Prevent data corruption during save operations

Problem: If application crashes during write, file may be corrupted

Solution: Atomic write pattern using temporary file

Algorithm:

```
FUNCTION _auto_save():
    backup_file = 'data/victims_backup.json'
    temp_file = backup_file + '.tmp'

    TRY:
        # Step 1: Write to temporary file
        WITH open(temp_file, 'w') as f:
            json.dump(victims_database, f, indent=2)

        # Step 2: Verify write successful
        IF file_exists(temp_file) AND file_size(temp_file) > 0:

            # Step 3: Atomic rename (replaces old file)
            os.replace(temp_file, backup_file)

            # Success
            last_backup_time = now()
            RETURN True

    CATCH Exception as e:
        # Clean up temp file if it exists
        IF file_exists(temp_file):
            os.remove(temp_file)
        PRINT "Auto-save failed:" + str(e)
        RETURN False
```

Why Atomic:

1. **os.replace()** is atomic operation on most systems
2. If crash during Step 1, original file untouched
3. If crash during Step 2, original file untouched
4. Only during Step 3 (atomic rename) does file change
5. Rename is instant (metadata operation, not copy)

Recovery Scenarios:

Scenario	Original File	Temp File	Result
Normal	Intact	Created → Renamed	Updated
Crash in Step 1	Intact	Partial/None	Original preserved
Crash in Step 2	Intact	Complete	Original preserved
Crash in Step 3	Replaced	Deleted	New version saved

6.4 WebSocket Broadcasting Algorithm

Purpose: Push real-time updates to multiple connected clients

Architecture:

```
Client 1


                    WebSocket
               Server
Client 2        (Async IO)




Client 3



                         broadcast_packet()



                    Serial
                    Reader
```

Implementation:

```python
import asyncio
import websockets
import json

# Global set of connected clients
connected_clients = set()

async def websocket_handler(websocket, path):
    """
    Handle individual WebSocket connection
    """
    # Register client
    connected_clients.add(websocket)

    try:
        # Keep connection alive
        await websocket.wait_closed()
    finally:
        # Unregister client
        connected_clients.remove(websocket)

async def broadcast_packet(packet):
    """
    Broadcast packet to all connected clients
    """
    if not connected_clients:
        return

    # Convert packet to JSON
    message = json.dumps(packet)

    # Send to all clients concurrently
    await asyncio.gather(
        *[client.send(message) for client in connected_clients],
        return_exceptions=True  # Don't fail if one client errors
    )

def start_websocket_server():
    """
    Start WebSocket server in background thread
    """
    import threading

    def run_server():
        # Create new event loop for this thread
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)

        # Start WebSocket server
        start_server = websockets.serve(websocket_handler, "localhost", 8765)

        loop.run_until_complete(start_server)
        loop.run_forever()

    # Start in daemon thread
    thread = threading.Thread(target=run_server, daemon=True)
    thread.start()
```

Message Protocol:

```
{
  "type": "victim_update",
  "data": {
    "ID": 1001,
    "LAT": 13.022,
    "LON": 77.587,
    "RSSI": -75,
    "STATUS": "STRANDED"
  },
  "timestamp": "2025-12-24T10:30:45"
}
```

6.5 Packet Validation Pipeline

Purpose: Ensure data integrity before processing

Multi-Stage Validation:

```
Stage 1: Format Validation
   Check JSON structure
   Check required fields
   Check data types

Stage 2: Range Validation
   Latitude: -90 to 90
   Longitude: -180 to 180
   RSSI: -150 to -30
   ID: 1 to 9999
   Battery: 0 to 100 (if present)

Stage 3: Logical Validation
   Coordinates not (0, 0)
   RSSI not exactly 0
   Timestamp not future
```

```
  Status in valid set

Stage 4: Sanitization
  Trim whitespace
  Normalize status (uppercase)
  Round coordinates (6 decimals)
  Clamp values to ranges
```

Implementation:

```python
def validate_packet(packet):
    """
    Comprehensive packet validation

    Returns:
        tuple: (is_valid, error_message)
    """
    # Stage 1: Format Validation
    if not isinstance(packet, dict):
        return False, "Packet must be a dictionary"

    required_fields = ['ID', 'LAT', 'LON', 'RSSI']
    for field in required_fields:
        if field not in packet:
            return False, f"Missing required field: {field}"

    # Check data types
    try:
        victim_id = int(packet['ID'])
        lat = float(packet['LAT'])
        lon = float(packet['LON'])
        rssi = int(packet['RSSI'])
    except (ValueError, TypeError):
        return False, "Invalid data types"

    # Stage 2: Range Validation
    if not (-90 <= lat <= 90):
        return False, f"Invalid latitude: {lat}"

    if not (-180 <= lon <= 180):
        return False, f"Invalid longitude: {lon}"

    if not (-150 <= rssi <= -30):
        return False, f"Invalid RSSI: {rssi}"

    if not (1 <= victim_id <= 9999):
        return False, f"Invalid ID: {victim_id}"
    # Battery validation (if present)
    if 'BATTERY' in packet:
        try:
            battery = int(packet['BATTERY'])
            if not (0 <= battery <= 100):
                return False, f"Invalid battery: {battery}"
        except (ValueError, TypeError):
            return False, "Invalid battery type"

    # Stage 3: Logical Validation
    if lat == 0.0 and lon == 0.0:
        return False, "Invalid coordinates: (0, 0)"

    if rssi == 0:
        return False, "Invalid RSSI: 0"

    # Status validation (if present)
    if 'STATUS' in packet:
        valid_statuses = ['STRANDED', 'EN_ROUTE', 'RESCUED']
        status = packet['STATUS'].upper()
        if status not in valid_statuses:
            return False, f"Invalid status: {status}"

    # Stage 4: Sanitization (modify packet in place)
    packet['ID'] = victim_id
    packet['LAT'] = round(lat, 6)
    packet['LON'] = round(lon, 6)
    packet['RSSI'] = rssi

    if 'STATUS' in packet:
        packet['STATUS'] = packet['STATUS'].upper()

    return True, "Valid"
```

5. MATHEMATICAL FOUNDATIONS

5.1 Geographic Distance Calculation (Haversine Formula)

Purpose: Calculate great-circle distance between two GPS coordinates on Earth's surface

Mathematical Formula:

```
Given two points:
  Point 1: (lat , lon )
  Point 2: (lat , lon )
```

```
Haversine Formula:
  a = sin²(Δ /2) + cos( ) × cos( ) × sin²(Δ /2)
  c = 2 × atan2(√a, √(1-a))
  d = R × c

Where:
   ,   = latitude of points 1 and 2 (in radians)
  Δ  =   -
  Δ  =   -   (difference in longitude)
  R = Earth's radius = 6371 km
  d = distance between points (in kilometers)
```

Implementation in Python:

```python
import math

def calculate_distance(lat1, lon1, lat2, lon2):
    """
    Calculate distance between two GPS coordinates using Haversine formula

    Parameters:
        lat1, lon1: Coordinates of point 1 (decimal degrees)
        lat2, lon2: Coordinates of point 2 (decimal degrees)

    Returns:
        Distance in kilometers (float)
    """
    # Earth's radius in kilometers
    R = 6371.0

    # Convert degrees to radians
     1 = math.radians(lat1)
     2 = math.radians(lat2)
    Δ  = math.radians(lat2 - lat1)
    Δ  = math.radians(lon2 - lon1)

    # Haversine formula
    a = math.sin(Δ /2)**2 + math.cos( 1) * math.cos( 2) * math.sin(Δ /2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))

    # Distance in kilometers
    distance = R * c

    return distance
```

Usage in Application: - Coverage area calculation (distance from rescue station to victims) - Proximity sorting (nearest victims first) - Sector-based clustering validation

Accuracy: - Precision: ±0.5% for distances < 1000 km - Earth approximation: Spherical (ignores ellipsoid, adequate for rescue operations) - Valid range: Any two points on Earth

Example Calculation:

```
Rescue Station: (13.022, 77.587) - Bangalore
Victim Location: (13.045, 77.620)

Δ  = 0.023° = 0.000401 rad
Δ  = 0.033° = 0.000576 rad

a = sin²(0.0002005) + cos(13.022°) × cos(13.045°) × sin²(0.000288)
a   0.0000001602 + 0.9487 × 0.9486 × 0.0000000829
a   0.0000001602 + 0.0000000746
a   0.0000002348

c = 2 × atan2(√0.0000002348, √0.9999997652)
c   2 × atan2(0.000485, 0.999999)
c   2 × 0.000485
c   0.00097 radians

d = 6371 × 0.00097   6.18 km
```

5.2 Priority Scoring Algorithm

Purpose: Assign priority levels (HIGH/MEDIUM/LOW) based on signal strength and time since last update

Mathematical Model:

```
Priority Score = f(RSSI, Δt)

Where:
  RSSI = Received Signal Strength Indicator (dBm)
  Δt = Time since last update (minutes)

Decision Tree:
  IF RSSI < RSSI_weak_threshold OR Δt > Time_critical_threshold:
      Priority = HIGH
  ELSE IF RSSI_weak_threshold  RSSI  RSSI_strong_threshold:
      Priority = MEDIUM
  ELSE:
      Priority = LOW
```

```
Default Thresholds:
  RSSI_strong_threshold = -70 dBm
  RSSI_weak_threshold = -85 dBm
  Time_critical_threshold = 15 minutes
```

Implementation:

```python
def calculate_priority(victim, rssi_weak_threshold=-85,
                      time_critical_threshold=15):
    """
    Calculate victim priority based on signal strength and staleness

    Parameters:
        victim (dict): Victim record with RSSI and LAST_UPDATE
        rssi_weak_threshold (int): Weak signal cutoff (dBm)
        time_critical_threshold (int): Critical time cutoff (minutes)

    Returns:
        str: 'HIGH', 'MEDIUM', or 'LOW'
    """
    # Extract RSSI
    rssi = victim['RSSI']

    # Calculate time since last update
    last_update = datetime.fromisoformat(victim['LAST_UPDATE'])
    now = datetime.now()
    time_delta = (now - last_update).total_seconds() / 60  # Convert to minutes

    # Priority logic
    if rssi < rssi_weak_threshold or time_delta > time_critical_threshold:
        return 'HIGH'
    elif rssi < -70:  # Between weak and strong thresholds
        return 'MEDIUM'
    else:
        return 'LOW'
```

Rationale:

RSSI-Based Priority: - **Strong Signal (RSSI > -70 dBm):** Victim close to ground station, good communication, LOW priority - **Medium Signal (-85 to -70 dBm):** Victim at moderate distance, MEDIUM priority - **Weak Signal (RSSI < -85 dBm):** Victim far away or signal obstructed, HIGH priority (may lose contact)

Time-Based Priority: - **Recent Update (< 15 min):** Device actively transmitting, LOW/MEDIUM priority - **Stale Update (> 15 min):** Device may have failed, victim in danger, HIGH priority

Combined Logic:

Truth Table:

RSSI	Time Δt	Priority
Strong	Recent	LOW
Medium	Recent	MEDIUM
Weak	Recent	HIGH
Strong	Stale	HIGH
Medium	Stale	HIGH
Weak	Stale	HIGH

5.3 Signal Strength Interpretation

RSSI to Distance Approximation:

```
Path Loss Model (Free Space):
  RSSI = P_tx - PL(d)

Where:
  P_tx = Transmission power (dBm)
  PL(d) = Path loss at distance d

Path Loss Formula:
  PL(d) = 20 × log (d) + 20 × log (f) + 32.45

Where:
  d = distance in kilometers
  f = frequency in MHz

For LoRa at 868 MHz with 22 dBm TX power:
  RSSI  22 - (20×log (d) + 20×log (868) + 32.45)
  RSSI  22 - (20×log (d) + 59.01 + 32.45)
  RSSI  -69.46 - 20×log (d)
```

Approximate Distance Table:

RSSI (dBm)	Estimated Distance	Signal Quality	Priority
-30 to -50	0.01 - 0.1 km	Excellent	LOW
-50 to -70	0.1 - 1 km	Good	LOW
-70 to -85	1 - 5 km	Fair	MEDIUM
-85 to -100	5 - 10 km	Weak	HIGH
-100 to -120	10 - 20 km	Very Weak	HIGH
< -120	> 20 km	Critical	HIGH

Note: Actual distances vary due to terrain, obstacles, weather

5.4 Rescue Efficiency Scoring

Purpose: Quantify operation effectiveness for analytics

Formula:

```
Efficiency Score = (W  × RescueRate + W  × SpeedScore) / (W  + W )

Where:
  RescueRate = (Rescued / Total) × 100

  SpeedScore = 100 × (1 - AvgRescueTime / MaxAcceptableTime)

  W  = Weight for rescue rate = 0.6
  W  = Weight for speed = 0.4
  MaxAcceptableTime = 60 minutes (threshold)

Grade Assignment:
  Score   90: Excellent
  Score   80: Good
  Score   70: Satisfactory
  Score   60: Needs Improvement
  Score < 60: Poor
```

Implementation:

```python
def calculate_rescue_efficiency(rescued_count, total_count, avg_rescue_time_minutes):
    """
    Calculate operation efficiency score

    Parameters:
        rescued_count (int): Number of rescued victims
        total_count (int): Total number of victims
        avg_rescue_time_minutes (float): Average time to rescue (minutes)

    Returns:
        tuple: (efficiency_score, grade)
    """
    if total_count == 0:
        return 0.0, "N/A"

    # Calculate rescue rate
    rescue_rate = (rescued_count / total_count) * 100

    # Calculate speed score (normalized against 60-minute threshold)
    max_acceptable_time = 60
    speed_score = 100 * (1 - min(avg_rescue_time_minutes, max_acceptable_time) / max_accepta
    speed_score = max(0, speed_score)  # Clamp to 0 minimum
    # Weighted average
    w1, w2 = 0.6, 0.4
    efficiency_score = (w1 * rescue_rate + w2 * speed_score) / (w1 + w2)

    # Assign grade
    if efficiency_score >= 90:
        grade = "Excellent"
    elif efficiency_score >= 80:
        grade = "Good"
    elif efficiency_score >= 70:
        grade = "Satisfactory"
    elif efficiency_score >= 60:
        grade = "Needs Improvement"
    else:
        grade = "Poor"

    return efficiency_score, grade
```

Example Calculation:

```
Scenario:
  Total victims: 20
  Rescued: 18
  Average rescue time: 35 minutes

Calculations:
  RescueRate = (18 / 20) × 100 = 90%

  SpeedScore = 100 × (1 - 35/60) = 100 × 0.417 = 41.7

  EfficiencyScore = (0.6 × 90 + 0.4 × 41.7) / 1.0
                  = (54 + 16.68) / 1.0
                  = 70.68

  Grade = "Satisfactory"
```

5.5 Geographic Clustering (Grid-Based)

Purpose: Identify victim density hotspots for resource allocation

Algorithm:

```
Grid-Based Spatial Clustering:

  1. Define grid cell size:
     Cell_lat = 0.001°   111 meters
     Cell_lon = 0.001°   111 meters (at equator)
```

```
 2. Calculate grid cell for each victim:
    Grid_x = floor(victim_lon / Cell_lon)
    Grid_y = floor(victim_lat / Cell_lat)
    Cell_ID = (Grid_x, Grid_y)

 3. Count victims per cell:
    Density[Cell_ID] = number of victims in cell

 4. Identify hotspots:
    Hotspot = Cell where Density   Threshold
```

Implementation:

```python
def analyze_geographic_density(victims, cell_size=0.001):
    """
    Cluster victims into geographic grid cells

    Parameters:
        victims (dict): Victim records
        cell_size (float): Grid cell size in degrees

    Returns:
        dict: {(grid_x, grid_y): victim_count}
    """
    density_map = {}

    for victim in victims.values():
        if victim['STATUS'] == 'RESCUED':
            continue  # Skip rescued victims

        lat = victim['LAT']
        lon = victim['LON']

        # Calculate grid cell
        grid_x = int(lon / cell_size)
        grid_y = int(lat / cell_size)
        cell_id = (grid_x, grid_y)

        # Increment density counter
        if cell_id not in density_map:
            density_map[cell_id] = []

        density_map[cell_id].append(victim['ID'])

    # Convert to counts
    density_counts = {cell: len(victims) for cell, victims in density_map.items()}
    return density_counts
```

Heatmap Generation:

```python
def create_density_heatmap(density_map):
    """
    Convert density map to heatmap coordinates

    Parameters:
        density_map (dict): {(grid_x, grid_y): count}

    Returns:
        list: [[lat, lon, weight], ...]
    """
    heatmap_data = []

    for (grid_x, grid_y), count in density_map.items():
        # Convert grid cell back to coordinates (cell center)
        lon = (grid_x + 0.5) * 0.001
        lat = (grid_y + 0.5) * 0.001

        # Weight = victim count
        heatmap_data.append([lat, lon, count])

    return heatmap_data
```

5.6 Signal Trend Analysis

Purpose: Detect signal deterioration indicating victim movement or device failure

Statistical Method:

```
Linear Regression on RSSI History:

  Given RSSI history: [r , r , r , ..., r ]
  Time points: [t , t , t , ..., t ]

  Fit linear model: RSSI = m × t + b

  Where:
    m = slope (signal change rate)
    b = intercept (initial signal)

  Slope calculation (Least Squares):
    m = (n (t r ) -  t r ) / (n t ² - ( t )²)
  Trend interpretation:
    m < -2 dBm/update: Deteriorating (victim moving away)
    -2   m   2: Stable
    m > 2 dBm/update: Improving (victim approaching)
```

Implementation:

```python
import numpy as np

def analyze_signal_trend(rssi_history):
    """
    Analyze RSSI trend using linear regression

    Parameters:
        rssi_history (list): Historical RSSI values

    Returns:
        tuple: (trend_direction, slope, confidence)
    """
    if len(rssi_history) < 3:
        return "INSUFFICIENT_DATA", 0.0, 0.0

    # Time points (equally spaced)
    n = len(rssi_history)
    time_points = np.arange(n)
    rssi_array = np.array(rssi_history)

    # Calculate slope using numpy polyfit (degree 1 = linear)
    coefficients = np.polyfit(time_points, rssi_array, 1)
    slope = coefficients[0]

    # Calculate R² (coefficient of determination)
    p = np.poly1d(coefficients)
    y_predicted = p(time_points)
    ss_res = np.sum((rssi_array - y_predicted) ** 2)
    ss_tot = np.sum((rssi_array - np.mean(rssi_array)) ** 2)

    if ss_tot == 0:
        r_squared = 0.0
    else:
        r_squared = 1 - (ss_res / ss_tot)

    # Determine trend direction
    if slope < -2:
        trend = "DETERIORATING"
    elif slope > 2:
        trend = "IMPROVING"
    else:
        trend = "STABLE"

    confidence = r_squared * 100  # Convert to percentage

    return trend, slope, confidence
```

Example Analysis:

```
RSSI History: [-70, -72, -75, -78, -82, -85]

Linear regression:
  Slope (m) = -3.0 dBm/update
  R² = 0.98 (98% confidence)

Interpretation:
  Trend: DETERIORATING
  Rate: -3 dBm per update
  Confidence: 98%

Action: Flag as HIGH priority - signal weakening rapidly
```

---

6. ALGORITHM SPECIFICATIONS

6.1 Real-Time Update Mechanism

Purpose: Ensure UI updates immediately when new packets arrive

Challenge: Streamlit reruns entire script on state changes, but serial reading is in background thread

Solution: Force Rerun Pattern

Algorithm:

```
BACKGROUND THREAD (Serial Reader):
  WHILE is_reading:
      packet = read_serial_port()

      IF packet is valid:
          # Update data
          data_manager.add_or_update_victim(packet)

          # Set rerun flag (atomic operation)
          st.session_state.force_rerun = True

          # Update counters
          st.session_state.packet_count += 1
          st.session_state.last_packet_time = now()

MAIN THREAD (Streamlit):
  # Check at start of every page render
  IF st.session_state.force_rerun == True:
      # Clear flag
      st.session_state.force_rerun = False

      # Force immediate rerun
      st.rerun()

  # Continue normal page rendering...
```

Timing Diagram:

```
Time    Serial Thread          Main Thread (UI)

t=0     Read packet           Rendering page
t=1     Validate packet       ...
t=2     Update data           ...
t=3     Set force_rerun=True  ...
t=4     ...                   Check force_rerun
t=5     ...                   Detect True
t=6     ...                   Call st.rerun()
t=7     ...                   Start new render
t=8     ...                   Clear force_rerun
t=9     ...                   Render updated UI
```

Latency Analysis:

- Packet arrival to flag set: 5-10ms
- Flag detection to rerun trigger: 50-100ms (depends on current render)
- Rerun to updated UI: 200-500ms (full page render)
- **Total latency:** ~**300-600ms** (acceptable for disaster response)

6.2 Duplicate Prevention Algorithm

Purpose: Prevent duplicate victim records when same ID transmits multiple times

Strategy: ID-based deduplication with update logic

Algorithm:

```
FUNCTION add_or_update_victim(packet):
    victim_id = packet['ID']

    # Check if victim already exists
    IF victim_id IN victims_database:
        # UPDATE EXISTING
        existing_victim = victims_database[victim_id]

        # Update mutable fields
        existing_victim['LAT'] = packet['LAT']
        existing_victim['LON'] = packet['LON']
        existing_victim['RSSI'] = packet['RSSI']
        existing_victim['LAST_UPDATE'] = now()
        existing_victim['UPDATE_COUNT'] += 1

        # Append to RSSI history
        existing_victim['RSSI_HISTORY'].append(packet['RSSI'])

        # Keep only last 20 values
        IF length(existing_victim['RSSI_HISTORY']) > 20:
            existing_victim['RSSI_HISTORY'] = existing_victim['RSSI_HISTORY'][-20:]

        # Preserve immutable fields
        # (FIRST_DETECTED, initial status, etc. remain unchanged)

    ELSE:
        # CREATE NEW
        new_victim = {
            'ID': victim_id,
            'LAT': packet['LAT'],
            'LON': packet['LON'],
            'RSSI': packet['RSSI'],
            'RSSI_HISTORY': [packet['RSSI']],
            'STATUS': 'STRANDED',
            'FIRST_DETECTED': now(),
            'LAST_UPDATE': now(),
            'UPDATE_COUNT': 1,
            'RESCUED_TIME': None,
            'RESCUED_BY': None,
            'NOTES': '',
            'BATTERY': packet.get('BATTERY', None)
        }

        victims_database[victim_id] = new_victim
```

Key Design Decisions:

1. **ID as Primary Key:** Unique identifier prevents duplicates
2. **Update Strategy:** Overwrite mutable fields, preserve immutable
3. **RSSI History:** Ring buffer of last 20 values for trend analysis
4. **Timestamps:** FIRST_DETECTED preserved, LAST_UPDATE refreshed

5. **Counter:** UPDATE_COUNT tracks packet frequency

6.3 Auto-Save with Atomic Writes

Purpose: Prevent data corruption during save operations

Problem: If application crashes during write, file may be corrupted

Solution: Atomic write pattern using temporary file

Algorithm:

```
FUNCTION _auto_save():
    backup_file = 'data/victims_backup.json'
    temp_file = backup_file + '.tmp'

    TRY:
        # Step 1: Write to temporary file
        WITH open(temp_file, 'w') as f:
            json.dump(victims_database, f, indent=2)

        # Step 2: Verify write successful
        IF file_exists(temp_file) AND file_size(temp_file) > 0:

            # Step 3: Atomic rename (replaces old file)
            os.replace(temp_file, backup_file)

            # Success
            last_backup_time = now()
            RETURN True

    CATCH Exception as e:
        # Clean up temp file if it exists
        IF file_exists(temp_file):
            os.remove(temp_file)

        PRINT "Auto-save failed:" + str(e)
        RETURN False
```

Why Atomic:

1. **os.replace()** is atomic operation on most systems
2. If crash during Step 1, original file untouched
3. If crash during Step 2, original file untouched
4. Only during Step 3 (atomic rename) does file change
5. Rename is instant (metadata operation, not copy)

Recovery Scenarios:

Scenario	Original File	Temp File	Result
Normal	Intact	Created → Renamed	Updated
Crash in Step 1	Intact	Partial/None	Original preserved
Crash in Step 2	Intact	Complete	Original preserved
Crash in Step 3	Replaced	Deleted	New version saved

6.4 WebSocket Broadcasting Algorithm

Purpose: Push real-time updates to multiple connected clients

Architecture:

```
   Client 1

                           WebSocket
                     Server
   Client 2           (Async IO)



   Client 3


                                 broadcast_packet()


                           Serial
                           Reader
```

Implementation:

```python
import asyncio
import websockets
import json

# Global set of connected clients
connected_clients = set()

async def websocket_handler(websocket, path):
    """
    Handle individual WebSocket connection
    """
    # Register client
    connected_clients.add(websocket)
    try:
        # Keep connection alive
        await websocket.wait_closed()
    finally:
        # Unregister client
        connected_clients.remove(websocket)

async def broadcast_packet(packet):
    """
    Broadcast packet to all connected clients
    """
    if not connected_clients:
        return

    # Convert packet to JSON
    message = json.dumps(packet)

    # Send to all clients concurrently
    await asyncio.gather(
        *[client.send(message) for client in connected_clients],
        return_exceptions=True  # Don't fail if one client errors
    )

def start_websocket_server():
    """
    Start WebSocket server in background thread
    """
    import threading

    def run_server():
        # Create new event loop for this thread
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)

        # Start WebSocket server
        start_server = websockets.serve(websocket_handler, "localhost", 8765)

        loop.run_until_complete(start_server)
        loop.run_forever()

    # Start in daemon thread
    thread = threading.Thread(target=run_server, daemon=True)
    thread.start()
```

Message Protocol:

```
{
```

```json
  "type": "victim_update",
  "data": {
    "ID": 1001,
    "LAT": 13.022,
    "LON": 77.587,
    "RSSI": -75,
    "STATUS": "STRANDED"
  },
  "timestamp": "2025-12-24T10:30:45"
}
```

6.5 Packet Validation Pipeline

Purpose: Ensure data integrity before processing

Multi-Stage Validation:

```
Stage 1: Format Validation
   Check JSON structure
   Check required fields
   Check data types

Stage 2: Range Validation
   Latitude: -90 to 90
   Longitude: -180 to 180
   RSSI: -150 to -30
   ID: 1 to 9999
   Battery: 0 to 100 (if present)

Stage 3: Logical Validation
   Coordinates not (0, 0)
   RSSI not exactly 0
   Timestamp not future
   Status in valid set

Stage 4: Sanitization
   Trim whitespace
   Normalize status (uppercase)
   Round coordinates (6 decimals)
   Clamp values to ranges
```

Implementation:

```python
def validate_packet(packet):
    """
    Comprehensive packet validation

    Returns:
```

```
    tuple: (is_valid, error_message)
"""
# Stage 1: Format Validation
if not isinstance(packet, dict):
    return False, "Packet must be a dictionary"

required_fields = ['ID', 'LAT', 'LON', 'RSSI']
for field in required_fields:
    if field not in packet:
        return False, f"Missing required field: {field}"

# Check data types
try:
    victim_id = int(packet['ID'])
    lat = float(packet['LAT'])
    lon = float(packet['LON'])
    rssi = int(packet['RSSI'])
except (ValueError, TypeError):
    return False, "Invalid data types"

# Stage 2: Range Validation
if not (-90 <= lat <= 90):
    return False, f"Invalid latitude: {lat}"

if not (-180 <= lon <= 180):
    return False, f"Invalid longitude: {lon}"

if not (-150 <= rssi <= -30):
    return False, f"Invalid RSSI: {rssi}"

if not (1 <= victim_id <= 9999):
    return False, f"Invalid ID: {victim_id}"

# Battery validation (if present)
if 'BATTERY' in packet:
    try:
        battery = int(packet['BATTERY'])
        if not (0 <= battery <= 100):
            return False, f"Invalid battery: {battery}"
    except (ValueError, TypeError):
        return False, "Invalid battery type"

# Stage 3: Logical Validation
if lat == 0.0 and lon == 0.0:
    return False, "Invalid coordinates: (0, 0)"
if rssi == 0:
    return False, "Invalid RSSI: 0"

# Status validation (if present)
if 'STATUS' in packet:
    valid_statuses = ['STRANDED', 'EN_ROUTE', 'RESCUED']
    status = packet['STATUS'].upper()
    if status not in valid_statuses:
        return False, f"Invalid status: {status}"

# Stage 4: Sanitization (modify packet in place)
packet['ID'] = victim_id
packet['LAT'] = round(lat, 6)
packet['LON'] = round(lon, 6)
packet['RSSI'] = rssi

if 'STATUS' in packet:
    packet['STATUS'] = packet['STATUS'].upper()

return True, "Valid"
```

---

7. DATA STRUCTURES

7.1 Victim Record Schema

Complete Field Specification:

```
VictimRecord = {
    # Identity
    'ID': int,                    # Unique identifier (1-9999)

    # Location
    'LAT': float,                 # Latitude in decimal degrees (-90 to 90)
    'LON': float,                 # Longitude in decimal degrees (-180 to 180)

    # Status
    'STATUS': str,                # Current state: STRANDED | EN_ROUTE | RESCUED

    # Signal Data
    'RSSI': int,                  # Current signal strength (dBm, -150 to -30)
    'RSSI_HISTORY': List[int],    # Last 20 RSSI values for trend analysis

    # Timestamps
    'FIRST_DETECTED': str,        # ISO 8601: "2025-12-24T10:30:45.123456"
    'LAST_UPDATE': str,            # ISO 8601: "2025-12-24T10:35:20.654321"
    'RESCUED_TIME': Optional[str],# ISO 8601 or None

    # Metadata
    'UPDATE_COUNT': int,           # Number of packets received from this victim
    'RESCUED_BY': Optional[str],   # Operator name who marked as rescued
    'NOTES': str,                  # Free-form operator notes
    'BATTERY': Optional[int],      # Battery percentage (0-100) if available
}
```

Field Constraints:

Field	Type	Nullable	Default	Validation
ID	int	No	-	1 ID 9999
LAT	float	No	-	-90 LAT 90
LON	float	No	-	-180 LON 180
STATUS	str	No	"STRANDED"	Must be in [STRANDED, EN_ROUTE, RESCUED]
RSSI	int	No	-	-150 RSSI -30
RSSI_HISTORY	list	No	[]	Max length 20, all values must be valid RSSI
FIRST_DETECTED	str	No	now()	Valid ISO 8601 timestamp
LAST_UPDATE	str	No	now()	Valid ISO 8601 timestamp
UPDATE_COUNT	int	No	1	COUNT 1
RESCUED_TIME	str	Yes	None	Valid ISO 8601 timestamp or None
RESCUED_BY	str	Yes	None	Any string or None
NOTES	str	No	""	Any string (unlimited length)
BATTERY	int	Yes	None	0 BATTERY 100 or None

Example Record:

```
{
  "ID": 1001,
  "LAT": 13.022456,
  "LON": 77.587234,
  "STATUS": "EN_ROUTE",
  "RSSI": -75,
  "RSSI_HISTORY": [-70, -72, -73, -75, -75],
  "FIRST_DETECTED": "2025-12-24T10:30:45.123456",
  "LAST_UPDATE": "2025-12-24T10:45:20.654321",
  "UPDATE_COUNT": 5,
  "RESCUED_TIME": null,
  "RESCUED_BY": null,
  "NOTES": "[10:35:12] Operator-01: Rescue team dispatched\n[10:40:30] Operator-01: Team rep
  "BATTERY": 65
}
```

7.2 Session State Structure

Streamlit Session State Variables:

```
st.session_state = {
    # Core Objects
    'data_manager': DataManager(),            # Victim database
    'serial_reader': SerialReader(),          # Serial port handler

    # Connection State
    'serial_connected': bool,                 # Is serial port connected?
    'serial_port': Optional[str],             # COM port name (e.g., "COM23")
    'baud_rate': int,                         # Baud rate (115200)

    # Operation Metadata
    'operator_name': str,                     # Current operator name
    'operation_start_time': datetime,         # When operation began

    # Statistics Counters
    'packet_count': int,                      # Total packets received
    'error_count': int,                       # Parse errors encountered
    'last_packet_time': Optional[datetime],   # Most recent packet timestamp

    # UI State
    'show_rescued': bool,                     # Show rescued victims on map?
    'show_heatmap': bool,                     # Show density heatmap?
    'show_priority_only': bool,               # Filter to high-priority only?

    # Map Configuration
    'map_center': List[float],                # [lat, lon] map center
    'map_zoom': int,                          # Zoom level (1-18)
    'rescue_centre_lat': float,               # Rescue base latitude
    'rescue_centre_lon': float,               # Rescue base longitude
    'location_detected': bool,                # Has user shared location?
    # Dynamic Thresholds
    'rssi_strong_threshold': int,           # Strong signal cutoff (dBm)
    'rssi_weak_threshold': int,             # Weak signal cutoff (dBm)
    'time_critical_threshold': int,         # Critical time cutoff (minutes)

    # Real-Time Control
    'force_rerun': bool,                    # Trigger UI refresh?
    'ws_server_started': bool,              # WebSocket server running?
}
```

7.3 Configuration Structure

config.py Constants:

```
# API Keys
GOOGLE_MAPS_API_KEY: str

# Serial Settings
DEFAULT_SERIAL_PORT: Optional[str]
DEFAULT_BAUD_RATE: int
SERIAL_TIMEOUT: int
AVAILABLE_BAUD_RATES: List[int]

# Map Settings
DEFAULT_MAP_CENTER_LAT: float
DEFAULT_MAP_CENTER_LON: float
DEFAULT_MAP_ZOOM: int
MAP_TILE_PROVIDERS: Dict[str, str]

# Priority Thresholds
RSSI_STRONG_THRESHOLD: int
RSSI_WEAK_THRESHOLD: int
TIME_STALE_THRESHOLD: int
TIME_CRITICAL_THRESHOLD: int

# Status Definitions
STATUS_STRANDED: str
STATUS_EN_ROUTE: str
STATUS_RESCUED: str
ALL_STATUSES: List[str]

# Color Schemes
STATUS_COLORS: Dict[str, str]
SIGNAL_COLORS: Dict[str, str]
MARKER_COLORS: Dict[str, str]
```

```
# Data Persistence
BACKUP_INTERVAL_SECONDS: int
BACKUP_FILE_PATH: str
RESCUE_LOG_PATH: str

# Geographic Settings
SECTOR_SIZE_LAT: float
SECTOR_SIZE_LON: float

# Validation Rules
VALID_LAT_RANGE: Tuple[float, float]
VALID_LON_RANGE: Tuple[float, float]
VALID_RSSI_RANGE: Tuple[int, int]
MIN_VICTIM_ID: int
MAX_VICTIM_ID: int

# Export Settings
EXPORT_COLUMNS: List[str]
EXPORT_FILENAME_FORMAT: str

# Debugging
DEBUG_MODE: bool
LOG_LEVEL: str
```

7.4 Packet Format

Incoming Serial Packet (JSON):

```
{
  "ID": 1001,             // Required: Victim identifier
  "LAT": 13.022456,       // Required: Latitude
  "LON": 77.587234,       // Required: Longitude
  "RSSI": -75,            // Required: Signal strength
  "STATUS": "STRANDED",  // Optional: Override status
  "BATTERY": 65,          // Optional: Battery percentage
  "TIMESTAMP": "..."      // Optional: Transmission timestamp
}
```

Minimal Valid Packet:

```
{"ID": 1001, "LAT": 13.022, "LON": 77.587, "RSSI": -75}
```

Extended Packet with All Fields:

```
{
  "ID": 1001,
  "LAT": 13.022456,
  "LON": 77.587234,
  "RSSI": -75,
  "STATUS": "STRANDED",
  "BATTERY": 65,
  "TIMESTAMP": "2025-12-24T10:30:45",
  "DEVICE_INFO": {
    "firmware": "v1.2.3",
    "hardware": "ESP32-S3"
  }
}
```

7.5 Export Data Structures

CSV Export Format:

```
ID,LAT,LON,STATUS,RSSI,FIRST_DETECTED,LAST_UPDATE,RESCUED_TIME,RESCUED_BY,UPDATE_COUNT
1001,13.022456,77.587234,RESCUED,-75,2025-12-24 10:30:45,2025-12-24 11:15:30,2025-12-24 11:1
1002,13.025123,77.590456,EN_ROUTE,-82,2025-12-24 10:35:12,2025-12-24 11:10:20,,,8
1003,13.019876,77.583567,STRANDED,-95,2025-12-24 10:40:30,2025-12-24 11:12:45,,,5
```

JSON Export Format:

```
{
  "export_metadata": {
    "timestamp": "2025-12-24T11:15:30",
    "operator": "Operator-01",
    "total_victims": 3,
    "rescued_count": 1
  },
  "victims": [
    {
      "ID": 1001,
      "LAT": 13.022456,
      "LON": 77.587234,
      "STATUS": "RESCUED",
      "RSSI": -75,
      "RSSI_HISTORY": [-70, -72, -75],
      "FIRST_DETECTED": "2025-12-24T10:30:45",
      "LAST_UPDATE": "2025-12-24T11:15:30",
      "UPDATE_COUNT": 15,
      "RESCUED_TIME": "2025-12-24T11:15:30",
      "RESCUED_BY": "Operator-01",
      "NOTES": "Rescued successfully",
      "BATTERY": 45
    }
  ]
}
```

7.6 Analytics Data Structures

Statistics Dictionary:

```
{
    'total': int,           # Total victims detected
    'stranded': int,        # Currently stranded
    'enroute': int,         # Rescue in progress
    'rescued': int,         # Successfully rescued
    'active': int,          # stranded + enroute
    'success_rate': float   # (rescued / total) * 100
}
```

Density Map:

```
{
    (grid_x, grid_y): {
        'count': int,                 # Number of victims in cell
        'victims': List[int],         # List of victim IDs
        'center': (lat, lon),         # Cell center coordinates
        'priority_count': {           # Breakdown by priority
            'HIGH': int,
            'MEDIUM': int,
            'LOW': int
        }
    }
}
```

Signal Trend Analysis:

```
{
    'victim_id': int,
    'trend': str,               # IMPROVING | STABLE | DETERIORATING
    'slope': float,             # dBm per update
    'confidence': float,        # R² score (0-100%)
    'history': List[int],       # RSSI values
    'prediction': int           # Predicted next RSSI
}
```

---

8. REAL-TIME PROCESSING

8.1 Serial Port Communication Flow

Complete Data Path:

```
  VICTIM DEVICE (ESP32 + LoRa)
```

```
   GPS Module
     Read coordinates
     Validate fix
     Generate packet

   LoRa Radio (SX1262)
     Encode packet
     Modulate signal
     Transmit at 868/915 MHz


                   LoRa Wireless (10-20 km range)


GROUND STATION HARDWARE (ESP32 + LoRa)

   LoRa Radio (SX1262)
     Receive signal
     Demodulate
     Measure RSSI
     Decode packet

   Serial UART
     Format as JSON string
     Add newline terminator
     Send to USB/COM port


                   Serial: 115200 baud, 8N1


COMPUTER OS (Windows/Linux/Mac)

   Serial Driver
     Buffer incoming bytes
     Handle flow control
     Provide COM/tty interface


                   PySerial Library


FALCONRESQ APPLICATION

   SerialReader (Background Thread)
     serial.readline()
```

```
  Decode bytes to UTF-8
  Parse JSON
  Validate packet
  Invoke callback

Callback Function
  DataManager.add_or_update_victim()
  WebSocket.broadcast_packet()
  session_state.packet_count++
  session_state.force_rerun = True

Main Thread (Streamlit)
  Check force_rerun flag
  Call st.rerun() if True
  Render updated UI
  Display new victim on map
```

8.2 Timing Analysis

End-to-End Latency Breakdown:

Stage	Duration	Notes
1. Victim device GPS acquisition	1-5 seconds	Depends on GPS fix quality
2. LoRa packet transmission	100-1000 ms	Depends on spreading factor
3. LoRa reception & demodulation	50-100 ms	Hardware processing
4. Serial transmission (JSON)	10-50 ms	Depends on baud rate & packet size
5. PySerial read	1-10 ms	OS buffer read
6. JSON parsing	1-5 ms	Python json.loads()
7. Validation	1-3 ms	Field checks
8. Database update	1-5 ms	Dictionary operation
9. UI rerun trigger	50-100 ms	Streamlit detect & trigger
10. Page re-render	200-500 ms	Full UI regeneration

Total Latency: 1.4 - 6.8 seconds (victim transmission to UI display)

Critical Path: GPS acquisition (1-5s) + LoRa transmission (0.1-1s) + UI render (0.2-0.5s)

Optimization Opportunities: - **GPS:** Use assisted GPS (A-GPS) to reduce fix time - **LoRa:** Reduce spreading factor for faster transmission (trades range) - **UI:** Selective rerender (only update changed elements) - future enhancement

8.3 Concurrency Model

Thread Overview:

```
Main Thread (Streamlit)
  UI rendering
  User interaction handling
  Page navigation
  Session state management

Serial Reader Thread (Python threading.Thread)
  Continuous serial port monitoring
  Packet parsing
  Callback invocation
  Error handling

WebSocket Server Thread (asyncio)
  Client connection management
  Message broadcasting
  Asynchronous I/O
  Connection cleanup

Auto-Save Thread (Timer)
  Periodic backup trigger
  JSON file writing
  Atomic file operations
```

Thread Safety Considerations:

1. **Session State Access:**
   - Streamlit manages thread-safe access to `st.session_state`
   - Atomic operations (integer increment, boolean set) are safe
   - Complex updates should use locks (not currently needed)
2. **DataManager:**
   - Single writer (serial thread via callback)
   - Multiple readers (UI thread, export thread)
   - Python GIL (Global Interpreter Lock) provides basic protection
   - Dictionary operations atomic at Python level
3. **WebSocket Clients:**
   - asyncio manages concurrent client connections
   - broadcast_packet() uses asyncio.gather() for concurrent sends
   - Exception handling prevents one client error from affecting others

Deadlock Prevention: - No nested locks - Callbacks execute quickly (no blocking I/O) - Serial thread can be stopped cleanly

8.4 Memory Management

Memory Footprint Analysis:

Base Application: - Streamlit framework: ~50 MB - Python interpreter: ~30 MB - Libraries (Folium, NumPy, Pandas): ~100 MB - **Total Base:** ~180 MB

Per-Victim Memory:

```
victim_record = {
    'ID': 8 bytes (int64)
    'LAT': 8 bytes (float64)
    'LON': 8 bytes (float64)
    'STATUS': ~20 bytes (string)
    'RSSI': 8 bytes (int64)
    'RSSI_HISTORY': 20 * 8 = 160 bytes (list of ints)
    'FIRST_DETECTED': ~50 bytes (ISO string)
    'LAST_UPDATE': ~50 bytes (ISO string)
    'UPDATE_COUNT': 8 bytes (int64)
    'RESCUED_TIME': ~50 bytes (string or None)
    'RESCUED_BY': ~30 bytes (string or None)
    'NOTES': variable (~100-1000 bytes)
    'BATTERY': 8 bytes (int64 or None)
}

Estimated per-victim: ~500-1500 bytes (average ~1 KB)
```

Scaling: - 100 victims: ~100 KB - 500 victims: ~500 KB
- 1000 victims: ~1 MB

Total Application Memory: - 100 victims: 180 MB + 0.1 MB **180 MB** - 500 victims: 180 MB + 0.5 MB **181 MB** - 1000 victims: 180 MB + 1 MB **181 MB**

Conclusion: Memory usage dominated by framework overhead, victim data negligible

Map Rendering Memory: - Folium generates HTML/JS for each marker - 100 markers: ~50 KB HTML - 500 markers: ~250 KB HTML - Rendered in browser (not Python process)

8.5 Performance Optimization Techniques

1. RSSI History Ring Buffer:

```
# Naive approach (creates new list every time)
victim['RSSI_HISTORY'] = victim['RSSI_HISTORY'] + [new_rssi]
if len(victim['RSSI_HISTORY']) > 20:
    victim['RSSI_HISTORY'] = victim['RSSI_HISTORY'][-20:]

# Optimized approach (in-place modification)
victim['RSSI_HISTORY'].append(new_rssi)
if len(victim['RSSI_HISTORY']) > 20:
    victim['RSSI_HISTORY'].pop(0)  # Remove oldest
```

2. NumPy Vectorization:

```
# Slow: Python loop
distances = []
for victim in victims:
    d = calculate_distance(base_lat, base_lon, victim['LAT'], victim['LON'])
    distances.append(d)

# Fast: NumPy vectorization
lats = np.array([v['LAT'] for v in victims])
lons = np.array([v['LON'] for v in victims])
distances = vectorized_haversine(base_lat, base_lon, lats, lons)
```

3. Selective UI Updates:

```
# Instead of full st.rerun(), use st.empty() placeholders
metric_placeholder = st.empty()

def update_metrics():
    with metric_placeholder.container():
        st.metric("Total", count)
```

4. Lazy Map Rendering:

```
# Only render map when tab is visible
if selected_tab == "Map":
    map_obj = create_victim_map(victims)
    st_folium(map_obj)
```

---

9. USER INTERFACE COMPONENTS

9.1 Dashboard Page (_pages/dashboard.py)

Purpose: Real-time operational interface for monitoring and managing victims

Layout Structure:

```
  METRICS BAR (5 columns)

   Total     Strand    Route     Rescue    Success
    145        23        8        114       78.6%
```

```
MAP CONTROLS (collapsible)
[x] Show Rescued   [ ] Priority Only   [ ] Show Heatmap
Base Layer: ( ) Roadmap  (•) Satellite


INTERACTIVE MAP (Folium)


             Victim Markers

    = Red (Stranded)
    = Orange (En-Route)
    = Green (Rescued)




VICTIM MANAGEMENT TABLE
Search: [________]  Status Filter: [All ]

 ID Stat  Prio   RSSI    Time  Batt     Actions

 01 STRA HIGH    -95     3min   45%  [En-Route] [ ]
 02 EN_R MED     -75    12min   67%  [Rescued] [ ]
 03 RESC LOW     -65    45min   22%  [View]
```

Key Functions:

render_dashboard()

```python
def render_dashboard():
    st.title(" Live Operations Dashboard")

    # Render metrics bar
    render_metrics_bar()

    # Render map controls
    render_map_controls()

    # Render interactive map
    render_victim_map()
    # Render victim management table
    render_victim_table()
```

render_metrics_bar()

```
def render_metrics_bar():
    stats = st.session_state.data_manager.get_statistics()

    col1, col2, col3, col4, col5 = st.columns(5)

    with col1:
        st.metric(
            label="Total Detected",
            value=stats['total'],
            delta=f"+{new_today}" if new_today > 0 else None
        )

    with col2:
        st.metric(
            label="Stranded",
            value=stats['stranded'],
            delta=None,
            delta_color="inverse"  # Red for high numbers
        )

    with col3:
        st.metric(
            label="En-Route",
            value=stats['enroute'],
            delta=None
        )

    with col4:
        st.metric(
            label="Rescued",
            value=stats['rescued'],
            delta=f"+{rescued_today}" if rescued_today > 0 else None,
            delta_color="normal"  # Green for rescued
        )

    with col5:
        st.metric(
            label="Success Rate",
            value=f"{stats['success_rate']:.1f}%",
            delta=None
        )

render_victim_table()

def render_victim_table():
    st.subheader("Victim Management")

    # Get victims
    victims = st.session_state.data_manager.get_all_victims()

    # Apply filters
    search_term = st.text_input("Search by ID or location", "")
    status_filter = st.selectbox("Status", ["All"] + config.ALL_STATUSES)

    if search_term:
        victims = [v for v in victims if search_term in str(v['ID'])]

    if status_filter != "All":
        victims = [v for v in victims if v['STATUS'] == status_filter]

    # Render table
    for victim in victims:
        col1, col2, col3, col4, col5, col6, col7 = st.columns([1, 2, 2, 2, 2, 2, 3])

        with col1:
            st.write(f"**{victim['ID']}**")

        with col2:
            status_color = config.STATUS_COLORS[victim['STATUS']]
            st.markdown(f"<span style='color:{status_color}'>{victim['STATUS']}</span>",
                        unsafe_allow_html=True)

        with col3:
            priority = calculate_priority(victim)
            priority_color = {'HIGH': 'red', 'MEDIUM': 'orange', 'LOW': 'green'}[priority]
            st.markdown(f"<span style='color:{priority_color}'>{priority}</span>",
                        unsafe_allow_html=True)

        with col4:
            signal_quality, color = get_signal_indicator(victim['RSSI'])
            st.write(f"{victim['RSSI']} dBm")

        with col5:
            time_ago = format_time_ago(datetime.fromisoformat(victim['LAST_UPDATE']))
            st.write(time_ago)
```

```python
        with col6:
            battery = victim.get('BATTERY', 'N/A')
            st.write(f"{battery}%") if isinstance(battery, int) else st.write(battery)

        with col7:
            if victim['STATUS'] == 'STRANDED':
                if st.button(f"Mark En-Route", key=f"enroute_{victim['ID']}"):
                    st.session_state.data_manager.mark_enroute(
                        victim['ID'],
                        st.session_state.operator_name
                    )
                    st.rerun()

            elif victim['STATUS'] == 'EN_ROUTE':
                if st.button(f"Mark Rescued", key=f"rescued_{victim['ID']}"):
                    st.session_state.data_manager.mark_rescued(
                        victim['ID'],
                        st.session_state.operator_name
                    )
                    st.rerun()

            if st.button(" ", key=f"notes_{victim['ID']}"):
                show_notes_dialog(victim)
```

9.2 Analytics Page (_pages/analytics.py)

Purpose: Operation statistics, trends, and performance metrics

Key Visualizations:

1. **Status Distribution Pie Chart**
2. **Signal Strength Distribution**
3. **Rescue Timeline**
4. **Geographic Density Heatmap**
5. **Signal Trend Analysis**

Status Distribution Chart

```python
def render_status_distribution():
    stats = st.session_state.data_manager.get_statistics()

    fig = go.Figure(data=[go.Pie(
        labels=['Stranded', 'En-Route', 'Rescued'],
        values=[stats['stranded'], stats['enroute'], stats['rescued']],
        marker=dict(colors=['#FF4445', '#FFA500', '#00CC66'])
    )])
    fig.update_layout(title="Victim Status Distribution")
    st.plotly_chart(fig)
```

9.3 Settings Page (_pages/settings.py)

Configuration Sections:

1. **Serial Port Configuration**
2. **Geolocation Detection**
3. **Priority Thresholds**
4. **System Management**

Geolocation Detection Implementation

```
def render_geolocation_section():
    st.subheader("  Rescue Station Location")

    col1, col2 = st.columns([2, 1])

    with col1:
        if st.button("  Detect My Location", type="primary"):
            # JavaScript to request geolocation
            geolocation_js = """
            <script>
            navigator.geolocation.getCurrentPosition(
                function(position) {
                    localStorage.setItem('user_lat', position.coords.latitude);
                    localStorage.setItem('user_lon', position.coords.longitude);
                    window.location.reload();
                },
                function(error) {
                    alert('Geolocation failed: ' + error.message);
                }
            );
            </script>
            """
            components.html(geolocation_js, height=0)

    # Display current location
    st.write(f"**Current Location:**")
    st.write(f"Latitude: {st.session_state.rescue_centre_lat:.6f}")
    st.write(f"Longitude: {st.session_state.rescue_centre_lon:.6f}")
```

10. CONFIGURATION MANAGEMENT

10.1 Environment Variables (.env file)

```
# API Keys
GOOGLE_MAPS_API_KEY=AIzaSyDuyQM98Dr8A-hTHqGAQh1-u7x7Anpd-vA

# Serial Port Settings
SERIAL_PORT=COM23
BAUD_RATE=115200

# Map Settings
MAP_CENTER_LAT=13.022
MAP_CENTER_LON=77.587
MAP_ZOOM=14

# Debugging
DEBUG_MODE=False
LOG_LEVEL=INFO
```

10.2 Configuration Hierarchy

```
Priority (High to Low):
1. User Input (Settings Page) → st.session_state
2. Environment Variables (.env) → config.py
3. Default Constants → config.py
```

10.3 Dynamic Threshold Configuration

Purpose: Allow operators to adjust priority thresholds without code changes

Storage: `st.session_state.rssi_weak_threshold`, etc.

Propagation:

```
# Settings Page: User adjusts slider
st.session_state.rssi_weak_threshold = st.slider("Weak Signal Threshold", -120, -60, -85)

# Map Page: Use dynamic threshold
map_obj = create_victim_map(
    victims,
    rssi_weak_threshold=st.session_state.rssi_weak_threshold
)

# Analytics Page: Use dynamic threshold
priority = calculate_priority(
    victim,
    rssi_weak_threshold=st.session_state.rssi_weak_threshold
)
```

11. SECURITY AND VALIDATION

11.1 Input Validation Pipeline

Multi-Layer Validation:

```
Layer 1: Serial Reader
  JSON format validation
  Required field check
  Basic type validation

Layer 2: Validators Module
  Coordinate range validation
  RSSI range validation
  ID range validation
  Logical consistency checks

Layer 3: Data Manager
  Duplicate prevention
  State transition validation
  Data sanitization
```

11.2 Data Integrity Safeguards

1. Atomic File Operations: - Prevents corruption during save - Uses temporary file + rename

2. Input Sanitization: - Strip whitespace - Normalize status (uppercase) - Round coordinates to 6 decimals - Clamp values to valid ranges

3. State Machine Enforcement:

```python
def mark_rescued(victim_id):
    victim = victims[victim_id]

    # Can only rescue STRANDED or EN_ROUTE
    if victim['STATUS'] not in ['STRANDED', 'EN_ROUTE']:
        raise ValueError("Cannot rescue victim in status: " + victim['STATUS'])
```

11.3 Error Handling Strategy

Serial Communication:

```python
try:
    serial_port.readline()
except serial.SerialException as e:
    log_error(f"Serial error: {e}")
    attempt_reconnect()
except UnicodeDecodeError as e:
    log_error(f"Decode error: {e}")
    error_count += 1
```

File Operations:

```
try:
    with open(backup_file, 'w') as f:
        json.dump(data, f)
except IOError as e:
    log_error(f"Write failed: {e}")
    alert_operator("Backup failed")
```

---

12. PERFORMANCE ANALYSIS

12.1 Benchmarks

Operation	Time	Notes
Packet parsing	1-5 ms	JSON decode + validation
Database update	1-3 ms	Dictionary operation
Priority calculation	0.1 ms	Simple arithmetic
Distance calculation	0.5 ms	Haversine formula
Map generation (100 victims)	500-800 ms	Folium HTML generation
UI full rerun	200-500 ms	Streamlit script execution
CSV export (100 victims)	50-100 ms	Pandas DataFrame
Auto-save (JSON)	20-50 ms	File I/O

12.2 Bottleneck Analysis

Primary Bottleneck: Map rendering with Folium

Mitigation Strategies: 1. Lazy loading (only render when map tab active) 2. Caching (Streamlit `@st.cache_data`) 3. Marker clustering for large datasets 4. Progressive loading (render in chunks)

12.3 Scalability Limits

Current Architecture:

Metric	Limit	Reason
Concurrent victims	500-1000	Map render time
Packet rate	100/sec	Serial parsing speed
RSSI history per victim	20 values	Memory/performance balance

Metric	Limit	Reason
Map markers	~500	Browser rendering limit
File size (backup JSON)	~5 MB	Acceptable for auto-save

Future Enhancements for Scale: - Database backend (SQLite/PostgreSQL) - Marker clustering library - WebSocket for map updates (no full rerun) - Server-side rendering

---

13. DEPLOYMENT GUIDE

13.1 Installation Steps

1. Prerequisites:

```
# Python 3.11+
python --version

# pip package manager
pip --version
```

2. Clone Repository:

```
git clone https://github.com/yourorg/falconresq.git
cd falconresq
```

3. Install Dependencies:

```
pip install -r requirements.txt
```

4. Configure Environment:

```
# Copy example env file
cp .env.example .env

# Edit .env file with your API keys
nano .env
```

5. Run Application:

```
streamlit run app.py
```

13.2 System Requirements

Minimum: - CPU: 2 cores, 2.0 GHz - RAM: 4 GB - Storage: 1 GB free space - OS: Windows 10, macOS 10.15, or Linux (Ubuntu 20.04+) - Python: 3.11 or higher - Network: Internet for map tiles

Recommended: - CPU: 4 cores, 3.0 GHz - RAM: 8 GB - Storage: 5 GB free space (for operation logs) - Display: 1920x1080 or higher

13.3 Configuration Checklist

- ☐ Google Maps API key configured in `.env`
- ☐ Serial port identified (check Device Manager)
- ☐ Baud rate matches hardware (default 115200)
- ☐ Rescue station coordinates set
- ☐ Priority thresholds adjusted for region
- ☐ Auto-save interval configured
- ☐ Operator name set

---

14. API REFERENCE

14.1 DataManager Class

Methods:

```
class DataManager:
    def __init__(self, backup_file: str = None)

    def add_or_update_victim(self, packet: dict) -> bool

    def mark_enroute(self, victim_id: int, operator: str) -> bool

    def mark_rescued(self, victim_id: int, operator: str) -> bool

    def get_all_victims(self) -> List[dict]

    def get_victims_by_status(self, status: str) -> List[dict]

    def get_statistics(self) -> dict

    def export_to_csv(self, filename: str, include_rescued: bool) -> bool

    def export_to_json(self, filename: str) -> bool

    def load_from_backup(self, filename: str) -> bool
```

14.2 SerialReader Class

```
class SerialReader:
    def __init__(self)

    def start_reading(self, port: str, baudrate: int,
                     on_packet_received: callable,
                     on_error: callable) -> bool
    def stop_reading(self) -> None

    def is_connected(self) -> bool

    def get_available_ports(self) -> List[str]
```

14.3 Helper Functions

```
def calculate_distance(lat1: float, lon1: float,
                       lat2: float, lon2: float) -> float

def calculate_priority(victim: dict, rssi_weak: int,
                       time_critical: int) -> str

def format_time_ago(timestamp: datetime) -> str

def get_signal_indicator(rssi: int) -> Tuple[str, str]

def validate_packet(packet: dict) -> Tuple[bool, str]

def validate_coordinates(lat: float, lon: float) -> bool

def validate_rssi(rssi: int) -> bool
```

---

15. TESTING AND QUALITY ASSURANCE

15.1 Unit Testing Strategy

Test Coverage Areas:

1. **Data Manager:**
    - Add/update victim
    - Mark status transitions
    - Export functionality
    - Statistics calculation
2. **Validators:**
    - Coordinate validation
    - RSSI validation
    - Packet structure validation
3. **Helpers:**
    - Distance calculation accuracy
    - Priority calculation logic
    - Time formatting

Example Test:

```python
def test_calculate_priority_high_rssi():
    victim = {
        'RSSI': -95,  # Weak signal
        'LAST_UPDATE': datetime.now().isoformat()
    }

    priority = calculate_priority(victim, rssi_weak=-85, time_critical=15)

    assert priority == 'HIGH'
```

15.2 Integration Testing

Test Scenarios:

1. **Serial Communication:**
    - Send mock packets via virtual COM port
    - Verify parsing and database update
2. **UI Workflow:**
    - Simulate victim detection
    - Mark en-route
    - Mark rescued
    - Verify audit trail
3. **Export Functionality:**
    - Generate CSV/JSON exports
    - Verify data completeness

15.3 Performance Testing

Load Tests: - 100 victims: <1s map render - 500 victims: <3s map render - 100 packets/sec: No dropped packets

---

16. FUTURE ENHANCEMENTS

16.1 Planned Features

Phase 1: Core Improvements - [ ] Database backend (SQLite/PostgreSQL) - [ ] User authentication system - [ ] Role-based access control - [ ] Advanced filtering and search

Phase 2: Enhanced Analytics - [ ] Predictive analytics (ML-based) - [ ] Historical trend analysis - [ ] Performance benchmarking - [ ] Custom report generation

Phase 3: Advanced Features - [ ] Multi-operator coordination - [ ] Voice alerts for high-priority victims - [ ] SMS/Email notifications - [ ] Mobile app integration - [ ] Drone flight path optimization

16.2 Architecture Evolution

Current: Monolithic Streamlit application

Future: Microservices architecture

```
  Web UI
 (React)



  API Gateway (FastAPI)


           Data Service (PostgreSQL)
           Analytics Service (ML)
           Serial Service (Hardware)
           WebSocket Service (Real-time)
```

APPENDICES

Appendix A: Hardware Specifications

Ground Station: - Model: WiFi LoRa 32 (V3) - MCU: ESP32-S3 (240 MHz dual-core) - Radio: Semtech SX1262 - Frequency: 863-928 MHz - TX Power: Up to 22 dBm - RX Sensitivity: -148 dBm - Range: 10-20 km (line of sight) - Interface: USB-C (Serial)

Appendix B: LoRa Configuration

Recommended Settings: - Spreading Factor: SF7-SF10 - Bandwidth: 125 kHz - Coding Rate: 4/5 - Preamble Length: 8 symbols - Sync Word: 0x12 (private network)

Appendix C: Glossary

- **RSSI:** Received Signal Strength Indicator (dBm)
- **LoRa:** Long Range wireless protocol
- **Haversine:** Great-circle distance formula
- **Streamlit:** Python web framework
- **Folium:** Python mapping library
- **PySerial:** Python serial communication library

Appendix D: References

1. LoRa Alliance Specification v1.0.4

2. GPS NMEA Protocol Specification
3. Streamlit Documentation
4. Folium Documentation
5. Haversine Formula (Mathematical Derivation)

DOCUMENT METADATA

Document Information: - **Version:** 3.0.0 - **Date:** December 24, 2025 - **Author:** Asshray Sudhakara - **Pages:** 50+ equivalent - **Total Lines:** 2500+ - **Estimated Read Time:** 3-4 hours

Revision History: - v1.0.0: Initial documentation - v2.0.0: Added expanded sections - v3.0.0: Complete exhaustive documentation with all formulas and algorithms

Contact: - Developer: Asshray Sudhakara - Institution: MARVEL, UVCE - Role: Aviation Coordinator & EvRe Domain Student

---

CONCLUSION

FalconResQ represents a complete, production-ready disaster management system that integrates hardware communication, real-time data processing, geospatial visualization, and professional analytics. This comprehensive documentation provides all necessary information for development, deployment, maintenance, and future enhancement of the system.

The system has been designed with disaster response best practices in mind, prioritizing reliability, real-time performance, and ease of use under high-stress operational conditions. All mathematical formulas, algorithms, and implementation details have been thoroughly documented to enable full understanding and future modification of the system.

END OF TECHNICAL DOCUMENTATION# FalconResQ: Disaster Management Ground Station ## EXHAUSTIVE TECHNICAL REFERENCE MANUAL
