FalconResQ: Disaster Management Ground Station - COMPREHENSIVE TECHNICAL REPORT (MEDIUM LEVEL)

Project Classification: Level 2 - Comprehensive Overview
Version: 1.0.0
Date: December 24, 2025
Developer: Asshray Sudhakara (ECE'27, MARVEL UVCE)
Scope: Detailed architecture, key algorithms, and operational workflows

---

1. SYSTEM PURPOSE & ARCHITECTURE

FalconResQ is a Python-based Streamlit web application for real-time disaster victim detection and rescue coordination. The system receives GPS data from wireless LoRa transmitters, displays victims on interactive maps, manages rescue status (STRANDED → EN_ROUTE → RESCUED), and provides comprehensive analytics for post-operation review.

Key Capabilities: - Real-time victim detection via serial LoRa receiver - Interactive geographic visualization (Folium maps) - Intelligent victim prioritization (signal strength + temporal analysis) - Multi-format data export (CSV, JSON, PDF reports) - Dynamic configuration without code changes

---

2. COMPLETE PROJECT STRUCTURE

```
Gnd_Stat_Web/
   app.py                          # Main entry point, session initialization
   config.py                        # Centralized configuration & constants
   requirements.txt                 # Python dependencies
   _pages/                          # Streamlit multi-page modules
      dashboard.py                 # Real-time rescue operations dashboard
      analytics.py                 # Advanced analytics & insights
      export.py                    # Multi-format data export
      settings.py                  # System configuration
   modules/                         # Core business logic
      serial_reader.py            # Serial port communication
      data_manager.py             # Victim data CRUD operations
      map_manager.py              # Folium map builder
      analytics.py                # Statistical analysis engine
      websocket_server.py         # Real-time broadcasting (optional)
   utils/                           # Utility functions
      helpers.py                  # Formatting, priority calculation
      validators.py               # Input validation
```

```
  data/                              # Data persistence
     victims_backup.json           # Auto-saved victim database
     rescue_log.csv                # Audit trail
  reports/                           # Documentation
```

---

3. TECHNOLOGY STACK & LANGUAGES USED

Component	Technology	Rationale
Backend	Python 3.11	Rapid development, NumPy/Pandas, PySerial support
Web Framework	Streamlit	No HTML/CSS needed, session state management
Mapping	Folium	Leaflet wrapper, multiple tile providers
Analytics	NumPy/Pandas	Efficient statistical calculations
Charts	Plotly	Interactive responsive visualizations
Serial Comms	PySerial	Native Python serial API
Data Storage	JSON/CSV	Human-readable, universal compatibility
Geolocation	JavaScript	Browser API for automatic location detection

---

4. CORE MODULES DOCUMENTATION

4.1 app.py - Application Orchestration

Responsibilities: - Streamlit page configuration and navigation - Session state initialization (persistent objects) - WebSocket server startup for real-time broadcasting - Browser geolocation integration - Sidebar navigation to pages

Key Functions:

```
initialize_session_state()
    Creates/preserves DataManager, SerialReader
    Initializes counters, operator info, UI toggles
```

```
get_user_location()
    Generates JavaScript for geolocation API
```

Design Pattern: Session State Persistence - Problem: Streamlit reruns entire script on state changes - Solution: Store objects in `st.session_state` so they persist - Benefit: Serial reader thread continues; data not lost

---

4.2 config.py - Centralized Configuration

Key Configuration:

```
# Serial Port Settings
DEFAULT_BAUD_RATE = 115200                    # LoRa standard
AVAILABLE_BAUD_RATES = [9600, 19200, ..., 921600]

# Map Configuration
DEFAULT_MAP_CENTER_LAT = 13.022               # Bangalore default
DEFAULT_MAP_ZOOM = 14                         # Street-level detail
MAP_TILE_PROVIDERS = {Google, OpenStreetMap}

# Victim Status Definitions
STATUS_STRANDED = "STRANDED"
STATUS_EN_ROUTE = "EN_ROUTE"
STATUS_RESCUED = "RESCUED"

# Priority Thresholds
RSSI_STRONG_THRESHOLD = -70 dBm               # Good signal
RSSI_WEAK_THRESHOLD = -85 dBm                 # Critical signal
TIME_CRITICAL_THRESHOLD = 15 minutes
TIME_STALE_THRESHOLD = 20 minutes

# Data Persistence
BACKUP_INTERVAL_SECONDS = 30
BACKUP_FILE_PATH = 'data/victims_backup.json'

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

Why These Settings? - 115200 baud: Standard for LoRa; reduces latency - Thresholds: Based on LoRa signal standards and typical operation timescales - Zoom 14: Shows building-level detail for victim location precision

---

4.3 modules/serial_reader.py - Hardware Communication

Architecture: Background Thread + Observer Pattern

```
Victim Device (LoRa)
       ↓ [LoRa RF Signal]
Ground Station Hardware
       ↓ [USB UART @ 115200 baud]
SerialReader._read_loop() [background thread]
       ↓ [JSON parse & validate]
on_packet_received callback
       ↓
DataManager.add_or_update_victim()
       ↓
st.session_state.force_rerun = True
       ↓
Streamlit page rerun → Map/metrics update
```

Key Class: SerialReader

```
class SerialReader:
    def start_reading(port, baudrate, on_packet_received, on_error)
        → Opens serial port
        → Starts background _read_loop() in daemon thread
        → Returns True on success

    def _read_loop() [background]
        while is_reading:
            line = serial_port.readline()
            packet = json.loads(line)
            if validate_packet(packet):
                on_packet_received(packet)
            else:
                on_error()

    def stop_reading()
        → Sets is_reading = False
        → Closes port & waits for thread
```

Expected Packet Format:

```
{"ID": 1, "LAT": 13.0227, "LON": 77.5873, "TIME": "2025-12-24 14:30:45", "RSSI": -72}
```

Why Background Thread? - Prevents UI freezing during serial reads - Continuous listening independent of Streamlit reruns - Non-blocking I/O

4.4 modules/data_manager.py - Data Persistence & CRUD

Victim Record Structure:

```
victim = {
    'ID': 1,
    'LAT': 13.0227, 'LON': 77.5873,
    'TIME': '2025-12-24 14:30:45',
    'RSSI': -72,
    'STATUS': 'STRANDED',
    'FIRST_DETECTED': '2025-12-24 14:30:45',
    'LAST_UPDATE': '2025-12-24 14:30:55',
    'RESCUED_TIME': None,
    'RESCUED_BY': None,
    'UPDATE_COUNT': 5,
    'RSSI_HISTORY': [-72, -71, -73, ...],
    'NOTES': 'Conscious, mobile'
}
```

Key Methods:

```
add_or_update_victim(packet)
    if victim_id exists:
        UPDATE: location, RSSI, timestamp
        APPEND: RSSI to history (keep 20)
    else:
        CREATE: new record, STATUS = STRANDED
    _auto_save()

mark_enroute(victim_id)
    → STATUS = EN_ROUTE, LAST_UPDATE = now

mark_rescued(victim_id, operator_name)
    → STATUS = RESCUED, RESCUED_TIME = now, RESCUED_BY = operator_name

get_statistics()
    → Returns: {total, stranded, enroute, rescued, percentages}

get_geographic_clusters()
    → Groups victims into 0.001° sectors (~100m grid)
    → Used for density heatmap

_auto_save()  [private]
    → Saves to victims_backup.json every 30 seconds

export_to_csv()
    → Generates CSV with all victim fields
```

Design Philosophy: - No duplicate victims (ID-based upsert) - Automatic persistence prevents data loss - RSSI history enables signal trend analysis - Temporal fields enable efficiency calculation

---

4.5 modules/map_manager.py - Geographic Visualization

Technology: Folium (Leaflet.js wrapper)

```
def create_victim_map(victims, center, zoom, show_heatmap, show_sectors):
    1. Calculate map center from victims
    2. Create Folium base map
    3. Add tile layers (Google, OSM)
    4. Add victim markers:
       for each victim:
           color = determine_marker_color(victim, thresholds)
           popup = build_popup_html(victim)
           add marker to map
    5. If show_heatmap: add HeatLayer
    6. If show_sectors: draw sector grid (0.001° cells)
    7. Add layer control
    return folium.Map object
```

Marker Color Logic:

```
if STATUS == RESCUED: green
elif STATUS == EN_ROUTE: orange
elif STATUS == STRANDED:
    if RSSI > RSSI_STRONG: green (good signal)
    elif RSSI > RSSI_WEAK: yellow (medium)
    else: red (critical)
```

Popup Information:

```
Victim ID: 1
Location: 13.0227°N, 77.5873°E
Signal: -72 dBm (Strong)
Status: STRANDED
Last Update: 5 minutes ago
```

Why Folium? - Python wrapper around Leaflet.js (no JavaScript coding) - Easy Streamlit integration via streamlit-folium - Multiple tile providers for redundancy - Built-in plugins (HeatLayer, MarkerCluster)

4.6 modules/analytics.py - Statistical Analysis

Key Calculations:

```
def calculate_rescue_rate():
    rescued = get_victims_by_status(RESCUED)
    operation_duration = now - earliest_detection

    rescue_times = [RESCUED_TIME - FIRST_DETECTED for each rescued victim]

    return {
        'total_rescued': len(rescued),
        'rescues_per_hour': len(rescued) / operation_duration,
        'average_rescue_time_minutes': mean(rescue_times),
        'fastest_rescue_minutes': min(rescue_times),
        'slowest_rescue_minutes': max(rescue_times)
    }

def analyze_geographic_density():
    clusters = get_geographic_clusters()
    for each sector:
        stranded = count(status == STRANDED)
        rescued = count(status == RESCUED)
        rescue_percentage = rescued / total
    return {total_sectors, sectors_dict, highest_density}

def analyze_signal_trends():
    weak_signals = count(RSSI < RSSI_WEAK_THRESHOLD)
    medium_signals = count(weak   RSSI < strong)
    strong_signals = count(RSSI   RSSI_STRONG)

    deteriorating = [victims with declining RSSI trend]

    return {counts, deteriorating_list}

def calculate_efficiency_score():
    rescue_rate = rescued / total
    speed_score = 100 - (avg_rescue_time / max_acceptable * 100)
    efficiency = (rescue_rate * 0.6) + (speed_score * 0.4)
    return efficiency (0-100)
```

4.7 _pages/dashboard.py - Real-Time Operations Dashboard

UI Components:

```
FalconResQ Dashboard - Active Rescue Operations
[Total: 42] [Stranded: 15] [En-Route: 8] [Rescued: 19] [Success: 45%]
```

```
 Interactive Map        Victim Management
 - Markers (colored)
 - Heatmap toggle         Show Rescued
 - Satellite view         Priority Only
 - Click for details
                        Victim #1: RSSI -72
                        [En-Route][Rescue]
                        Notes: ______


Victim Status Table

 ID   Status   RSSI   Last Update

 1    EN_ROUTE   -72   5 min ago
 2    STRANDED   -89   2 min ago
 3    RESCUED    -75   10 min ago
```

Real-Time Update Mechanism:

```python
# At TOP of every page (critical)
if st.session_state.get('force_rerun', False):
    st.session_state.force_rerun = False
    st.rerun()

# SerialReader callback
def on_packet_received(packet):
    st.session_state.data_manager.add_or_update_victim(packet)
    st.session_state.packet_count += 1
    st.session_state.force_rerun = True  # ← Triggers rerun
```

4.8 _pages/analytics.py - Operation Analytics

Charts Generated:

1. **Pie Chart:** Status distribution (STRANDED%, EN_ROUTE%, RESCUED%)
2. **Bar Chart:** Victims by geographic sector
3. **Line Chart:** Detection timeline (victims over time)
4. **Scatter Plot:** RSSI vs time (signal strength trends)
5. **Heatmap:** Geographic density visualization

Computed Metrics: - Rescue rate: rescued / total - Average rescue time - Efficiency score (0-100) - Geographic coverage: largest sector by victim count

---

4.9 __pages/export.py - Multi-Format Data Export

Formats:

1. **CSV Export:**

   ```
   ID, LAT, LON, STATUS, RSSI, FIRST_DETECTED, LAST_UPDATE, RESCUED_TIME, OPERATOR, UPDATE
   ```

2. **JSON Export** (Full victim records)
3. **Operation Report** (Markdown with stats & charts)
4. **Rescue Log** (Audit trail: timestamp, victim_id, status_change, operator)

4.10 __pages/settings.py - System Configuration

Configuration Sections:

1. **Serial Port Setup**
   - Select COM port (auto-detected)
   - Select baud rate (default 115200)
   - Connect/Disconnect buttons
2. **Map Configuration**
   - Tile provider selection
   - Zoom level adjustment
   - Heatmap/sector grid toggles
3. **Operator Information**
   - Operator name input
   - Shift start/end (optional)
4. **Priority Thresholds**
   - RSSI strong/weak threshold sliders
   - Time critical/stale threshold inputs
5. **System Management**
   - Clear victim data
   - Manual backup/restore
   - System information display

Geolocation Integration:

```javascript
// Browser geolocation in settings.py
navigator.geolocation.getCurrentPosition(
    function(position) {
        localStorage.setItem('user_lat', position.coords.latitude);
        localStorage.setItem('user_lon', position.coords.longitude);
    }
);
```

4.11 utils/helpers.py - Utility Functions

```
format_time_ago(timestamp)
    → "5 minutes ago", "2 hours ago", etc.

calculate_priority(victim, thresholds)
    → Priority level 1-5 (combines RSSI + temporal staleness)

get_signal_color(rssi, thresholds)
    → Color code for map markers

validate_coordinates(lat, lon)
    → GPS bounds validation

calculate_distance(lat1, lon1, lat2, lon2)
    → Haversine formula for distance

export_to_dataframe(victims)
    → Pandas DataFrame for analytics

format_victim_record(victim)
    → Formatted string for display
```

4.12 utils/validators.py - Data Validation

```
validate_packet(packet)
    → Check required fields, types, ranges

validate_status(status)
    → Check  {STRANDED, EN_ROUTE, RESCUED}

validate_operator_name(name)
    → Non-empty, max 50 chars

validate_notes(notes)
    → Max 500 characters

validate_rssi(rssi)
    → Range (-150, -30) dBm
```

---

5. CRITICAL ALGORITHMS

5.1 Victim Priority Calculation

```
def calculate_priority(victim, rssi_strong, rssi_weak, time_critical_min):
    last_update = victim['LAST_UPDATE']
    minutes_since_update = (now - last_update).minutes
    rssi = victim['RSSI']

    if rssi < rssi_weak or minutes_since_update > time_critical_min:
        return PRIORITY_CRITICAL  # Rescue immediately
    elif (rssi < rssi_strong) and (minutes_since_update > time_critical_min/2):
        return PRIORITY_HIGH      # Prioritize
    else:
        return PRIORITY_MEDIUM    # Stable

    # Reasoning: Weak signals + stale data = danger
```

5.2 Geographic Clustering (Sector-Based)

```
def get_geographic_clusters(victims):
    SECTOR_SIZE = 0.001 degrees  # ~100m grid

    clusters = {}
    for victim_id, victim in victims.items():
        sector_id = (int(LAT/SECTOR_SIZE), int(LON/SECTOR_SIZE))
        if sector_id not in clusters:
            clusters[sector_id] = []
        clusters[sector_id].append(victim_id)

    return clusters

    # Output:
    # {"(13022, 77587)": [1, 2, 3],
    #  "(13023, 77588)": [4, 5],
    #  "(13024, 77589)": [6, 7, 8, 9]}  ← Highest density
```

5.3 Signal Trend Detection

```
def detect_signal_deterioration(victim):
    rssi_history = victim['RSSI_HISTORY']  # Last 20 readings

    if len(rssi_history) < 5:
        return False

    recent_avg = mean(rssi_history[-5:])   # Last 5
    older_avg = mean(rssi_history[:5])     # First 5
```

```
is_deteriorating = recent_avg < (older_avg - 5 dBm)

# Deterioration = victim moving away or antenna failure
```

---

6. DATA FLOW ARCHITECTURE

```
Victim Device (LoRa TX)
  ↓ [LoRa Signal]
Ground Station (LoRa RX + MCU)
  ↓ [USB UART @ 115200]
SerialReader (background thread)
  ↓ [JSON parse & validate]
DataManager (add/update)
  ↓ [Auto-save to JSON]
force_rerun flag
  ↓ [Trigger Streamlit rerun]
Dashboard Page
  ↓ [MapManager renders map]
Folium map with markers
  ↓ [Browser visualization]
Operator sees victims on map
```

---

7. SECURITY & DATA INTEGRITY

7.1 Input Validation Pipeline

```
Serial read → JSON parse → Validate fields → Check ranges
→ Check coordinates → Check ID → Validated packet
→ on_packet_received() callback
```

7.2 Data Safeguards

- **Duplicate Prevention:** ID-based upsert
- **Range Validation:** GPS, RSSI, timestamps
- **Auto-Save:** JSON backups every 30 seconds
- **Audit Trail:** rescue_log.csv records all changes
- **Error Handling:** Failures don't block serial reading

---

8. PERFORMANCE & SCALABILITY

Metric	Capacity	Notes
Victims	100-500	Map render <1s for 100
Packet Rate	50-100/sec	Sustainable
Memory	~2 MB per 100 victims	Raw data + session state
Auto-save	~100ms	JSON serialization
Analytics Calc	<2s	NumPy vectorization

Scaling Path: - **SQLite:** 500+ victims - **PostgreSQL:** Enterprise multi-site - **Distributed:** Multiple servers with shared DB

---

9. DEPLOYMENT & OPERATION

9.1 Installation

```
python -m venv venv
venv\Scripts\activate
pip install -r requirements.txt
streamlit run app.py
```

9.2 Access

Browser: http://localhost:8501

9.3 Operational Workflow

```
1. [Connect Hardware] → Serial port selected
2. [Monitor Dashboard] → Watch for victim markers
3. [Victim Appears] → Click marker for details
4. [Dispatch Team] → Click "Mark En-Route"
5. [Team Arrives] → Click "Mark Rescued"
6. [Continue Monitoring] → Watch other victims
7. [Export Data] → CSV for audit
```

Total Code: 3,500+ lines
Version: 1.0.0
Status: Production Ready

Author: Asshray Sudhakara, MARVEL UVCE

FalconResQ offers a Streamlit-based dashboard for live victim detection and rescue coordination using data from ground station hardware. The application emphasizes real-time updates, dynamic thresholds for priority calculation, and geolocation-driven operations.

Main Components

- `app.py`: initializes session state and UI navigation.
- `modules/serial_reader.py`: background serial port reader.
- `modules/data_manager.py`: in-memory store and persistence helpers.
- `modules/map_manager.py`: Folium map creation and legend handling.
- `modules/analytics.py`: statistics and coverage calculation.
- `_pages/` (dashboard, analytics, export, settings): UI pages.
- `utils/helpers.py`, `utils/validators.py`: reusable logic and input validation.

Key Features and Why

- Background serial reader: keeps UI responsive while ingesting live sensor data.
- Session state objects: persist DataManager and SerialReader across reruns.
- Dynamic thresholds (RSSI/time): operator-tunable priorities without app restart.
- Geolocation: rescue station coordinates sourced from the browser for ease of use and privacy.
- Map legend and filtering: ensure visual and functional consistency with threshold values.

Core Algorithms

- Priority: uses RSSI thresholds and last-update staleness to classify HIGH/MEDIUM/LOW.
- Coverage: Haversine distance from rescue station to first detected beacon.
- Sector clustering: grid-based density for heatmaps.

Data Flow (short)

SerialReader -> validate packet -> DataManager.add_or_update_victim -> broadcast_packet -> UI rerun

File-by-file (short descriptions)

- `config.py`: constants and defaults.
- `app.py`: startup, autoload geolocation, sidebar.
- `modules/*.py`: core logic modules.
- `_pages/*.py`: UI pages using Streamlit components.
- `utils/*`: helpers/validators.

Security & Deployment Notes

- Keep API keys private and restrict them.

- Add operator authentication for multi-user deployments.
- Use a small DB for persistence for production deployments.

Recommendations

- Add unit tests for validators and business logic.
- Introduce CI scripts for linting and testing.
- Consider containerization (Docker) for consistent deployments.

(End of Medium Report)