**Rescue Decision Systems (RDS) - Development Guide (V0.2)**

**1. Purpose & Scope**

This document provides guidelines, rules, and best practices for the development, testing, and deployment of all components within the Rescue Decision Systems (RDS) platform. This includes SARSAT message processing, weather data retrieval, GIS integration, and machine learning-based distress classification.

This document ensures **consistency across modules**, **adherence to project-wide data handling rules**, and **maintains alignment with RDS\_Data\_Schema\_Rules and RDS\_Folder & Module Structure Guide**.

**2. Coding Standards & Best Practices**

**2.1 Code Structure**

* Use modular functions and avoid monolithic scripts.
* Each script should serve a **single responsibility** (e.g., parsing, visualization, ML analysis).
* Follow **PEP 8** for Python coding style.
* Use **meaningful variable and function names**.

**2.2 Package Management**

* Use setup.py for dependency management.
* Define all required packages in setup.py and install with:

pip install -e .

* Keep individual script imports minimal but ensure consistency across scripts.

All packages required for Rescue Decision Systems are listed in: ➡️ /docs/RDS\_Installation\_Requirements.md The authoritative package list for installation is maintained in: ➡️ setup.py (project root) Developers must install packages using: pip install -e . Any new package must: 1. Be added to setup.py with a pinned version (if known compatibility constraints exist). 2. Be added to RDS\_Installation\_Requirements.md with a note explaining its purpose. 3. Be reviewed for compatibility with existing packages (using pip check). setup\_imports.py remains the single source for imports across project scripts, ensuring consistent and centralized handling of: - Environment variable loading - Logging configuration - Core library imports setup.py defines the installation rules. setup\_imports.py controls how the code accesses them.

**2.3 Error Handling & Logging**

* Implement try-except blocks for error handling.
* Use Python’s logging module instead of print statements.
* Log errors to a central file for debugging:

python

CopyEdit

import logging

logging.basicConfig(filename='app.log', level=logging.INFO)

**3. Workflow & Environment**

**3.1 Local Development Setup**

* Use Anaconda + Jupyter Lab for interactive development.
* Test all changes locally before deployment.
* Maintain separate environments for development and production.

**3.2 GitHub Workflow**

* Maintain a **main branch for stable releases**.
* Create **feature branches for new functionality**.
* Use meaningful commit messages:

vbnet

CopyEdit

[Module] Short description of change

* Before merging, ensure:
  + Code is tested.
  + Code is reviewed.

**4. Schema Verification Process**

**4.1 Purpose**

The Schema Verification Script ensures the database schema remains aligned with:

* **sql\_models.py**
* **RDS\_Data\_Schema\_Rules**

It also verifies:

* New fields are properly stored.
* Missing data rules (NaN to NULL) are applied correctly.

**4.2 When to Run**

* After updates to sql\_models.py.
* After modifying fields in RDS\_Data\_Schema\_Rules.
* During regular development checkpoints.
* Before and after applying database migrations.

**4.3 How to Run**

The script is located in:

bash

CopyEdit

flask\_app/app/

Run it with:

bash

CopyEdit

python app/schema\_verification\_script.py

**5. Data Handling & Processing**

**5.1 Data Flow**

1. Input (SARSAT alerts, weather data, vessel tracking)
2. Temporary Processing (DataFrames in Pandas)
3. Storage (SQL database for permanent storage)
4. Analysis (ML models, GIS visualization)
5. User Display (Web interface, reports)

**5.2 Temporary Data Storage**

* Store temporary computations in Pandas DataFrames for quick analysis.
* Persist only finalized data in SQL after case closure.
* Support two locations per alert (A & B).
* Use position resolution to generate range rings.

**5.3 Data Validation**

* Ensure consistent units across datasets (e.g., degrees, UTC timestamps).
* Validate all incoming data.
* Flag malformed data rather than rejecting outright.
* Use robust parsing (e.g., regex) to handle incomplete SARSAT and weather messages.
* User-facing displays must show:
  + Distances in **meters and feet**.
  + Temperatures in **Celsius and Fahrenheit**.

**6. Deployment Strategy**

**6.1 DigitalOcean Deployment**

* Flask backend hosted on **DigitalOcean Droplet**.
* SSL encryption.
* Gunicorn + Nginx for production hosting.

**6.2 Deployment Steps**

1. Test locally using Flask + SQL.
2. Push changes to GitHub.
3. Deploy to DigitalOcean:

bash

CopyEdit

ssh user@server

cd /path/to/app

git pull origin main

systemctl restart gunicorn

1. Verify via web interface.

**7. Security & Access Control**

**7.1 User Authentication**

* Use Flask-Login.
* Restrict write access to authorized users.

**7.2 Secure Data Transmission**

* Use **HTTPS** for all web traffic.
* Encrypt stored user credentials with **bcrypt**.

**7.3 International Collaboration**

* Allow SAR teams to securely share case links.
* Real-time messaging for active cases.

**8. Processing Rules & Core Logic**

**8.1 Core Processing Rules**

**8.1.1 DataFrame-First Approach**

* All data starts in **Pandas DataFrames**.
* Data is only stored in SQL after processing completes.
* This ensures:
  + Consistent transformation logic.
  + Unified handling of missing data.
  + Compatibility with ML models.

**8.1.2 NaN to NULL Conversion Rule**

* Any empty or unparseable field is set to:
  + NaN in Pandas.
  + NULL in PostgreSQL.

**8.1.3 Coordinate Parsing Standardization**

* All coordinate parsing must use:

python

CopyEdit

from utils import parse\_coordinates

* Direct manual parsing (e.g., degrees/minutes splitting) is **prohibited**.

**8.2 Message Processing Rules**

**8.2.1 SARSAT Alert Parsing**

* All SARSAT messages are processed using:

python

CopyEdit

from sarsat\_parser import parse\_sarsat\_message

* Parser workflow:
  + Extract all fields to a **DataFrame**.
  + Normalize fields per RDS\_Data\_Schema\_Rules.
  + Call parse\_coordinates() for all coordinates.
  + Fill missing fields with NaN.

**8.2.2 Automatic Message Type Detection**

* The parser auto-detects message type (Unlocated First Alert, etc.).
* This type is saved in alert\_type.

**Supported Types (V0.2)**

* Unlocated First Alert
* Initial Located Alert
* Position Update
* Detection Update
* Notification of Country of Registration

**Adding New Message Types**

1. Add detection logic to sarsat\_parser.py.
2. Update:
   * **RDS\_Data\_Schema\_Rules (alert\_type).**
   * **RDS\_Development\_Guide (this section).**

**8.3 Weather Processing Rules**

**8.3.1 Weather Data Sources**

* NOAA Shore Stations
* NOAA Buoy Stations (NDBC)

**8.3.2 Weather Data Handling**

* All weather data is fetched using:

python

CopyEdit

from noaa\_weather\_fetch import fetch\_nearest\_weather\_stations, fetch\_weather\_data

* Missing fields are stored as NaN (converted to NULL in SQL).

**8.3.3 Dual Position Handling**

* Weather data is fetched for:
  + **Position A (Primary).**
  + **Position B (Alternate/Doppler).**

**8.4 Logging & Error Handling**

**8.4.1 Structured Logging**

* All scripts must use:

python

CopyEdit

import logging

* Errors must log at ERROR level.
* Recoverable parsing issues log at WARNING level.

**8.4.2 Centralized Error Logger**

* All scripts can call:

python

CopyEdit

from utils import log\_error\_and\_continue

* This ensures consistent log formatting.

**8.5 Schema Verification & Testing**

**8.5.1 Verification Process**

* Run schema verification after:
  + Changes to sql\_models.py.
  + Changes to RDS\_Data\_Schema\_Rules.

**8.5.2 Test Message Library**

* The development guide maintains a **test message library**, including:
  + All supported types.
  + Edge cases (missing coordinates, malformed fields).
  + Multiple coordinate formats.