DRONE MANAGEMENT SYSTEM FOR DISASTERS

Project done by:
Arnav Lunia (PES1UG23CS104)
Anuva Kishore (PES1UG23CS090)

1. Problem Statement: Enhancing Real-time Situational Awareness and Resource Deployment in Disaster Zones through Autonomous Drone Analytics.

This project addresses the critical need for rapid, data-driven decision-making during disaster events. Traditional disaster response efforts are often hampered by delayed information, fragmented communication, and inefficient resource allocation. The **Drone Disaster Response Dashboard** aims to solve this by creating a centralized, real-time analytics platform that integrates data from a fleet of autonomous drones. The system will provide immediate insights into ongoing disaster severity, required maintenance for drone hardware, and the current operational status of personnel, thereby significantly reducing response times and improving the effectiveness of humanitarian and recovery missions.

2. User Requirement Specification (URS)

Purpose of the Project

The purpose of this project is to build a simple, reliable system that manages drone-based disaster response operations. It allows users to track disasters, deploy drones, store mission details, and monitor alerts in one place. The project demonstrates how databases,

backend logic, and a visual dashboard can work together to support fast decision-making during emergency situations.

Scope of the Project

The scope includes a full database system with tables for disasters, drones, operators, alerts, and missions; a backend server to process requests; and a frontend dashboard for users to add, view, and resolve operations. It also covers triggers, procedures, analytical queries, and basic role-based access. The system focuses on database-driven disaster management but does not include actual drone hardware or real-time GPS integration.

2.1. Functional Requirements (What the System Must Do):

- **Real-time Dashboard Overview:** Active Disaster Count, Resource Tally (Drones/Personnel), Critical Alert Count (Maintenance/Availability).
- **Disaster Tracking:** Sortable Log (Name, ID, Status, Time). Filter Mission Status by Type (Success Rate, Completion Time).
- **Personnel Management:** Roster Display (Name, Certification, Experience). Query Personnel by Certification.
- **Drone Fleet Health:** Maintenance Alert List (High/Critical). Payload Tier Summary.

2.2. Non-Functional Requirements (How Well the System Performs):

- **UI Responsiveness:** The user interface must load quickly and be highly responsive to client-side actions (e.g., scrolling, sorting, filtering).

Reliability:

- **Data Consistency:** The system must consistently display mock or simulated data feeds without errors.

Security:

- **User Authentication:** The system must implement basic **user authentication** (login/logout) to protect access to the dashboard.

Usability:

- **Access Control:** The system must provide distinct access control points (different URLs) for Client and Editor Staff to segregate views.
- **Cross-Platform Compatibility:** The dashboard must be fully functional and render correctly on both desktop and mobile/tablet browsers.

3. List of Softwares / Tools / Programming Languages Used:

Programming Languages:

- **SQL** – For creating the `drone` database, tables, constraints, and queries.
- **JavaScript** – For frontend interactivity and backend server logic.
- **HTML** – For structuring the web pages (dashboard UI).
- **CSS** – For styling the frontend (custom `style.css`).

Database:

- **MySQL** – Relational database to store all drone, disaster, operator, and mission data.

- **MySQL Command-Line Client** – Used to run `CREATE TABLE`, `INSERT`, and query commands.

Backend:

- **Node.js** – Runtime used to execute the backend code (`server.cjs`).

Frontend

- **React.js** – Used in `DisasterDashboard.jsx` to build a dynamic disaster response dashboard UI.

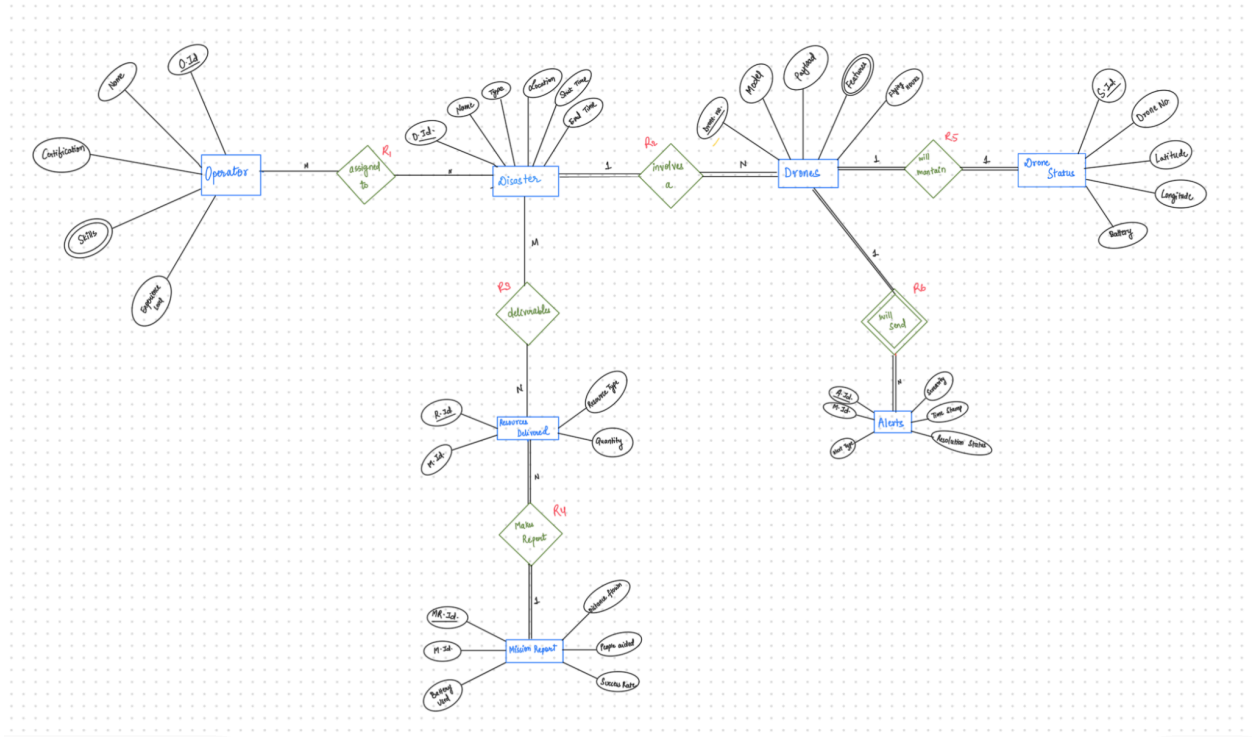
- **Web Browser (e.g., Chrome)** – To run and test the frontend dashboard.

Development Tools

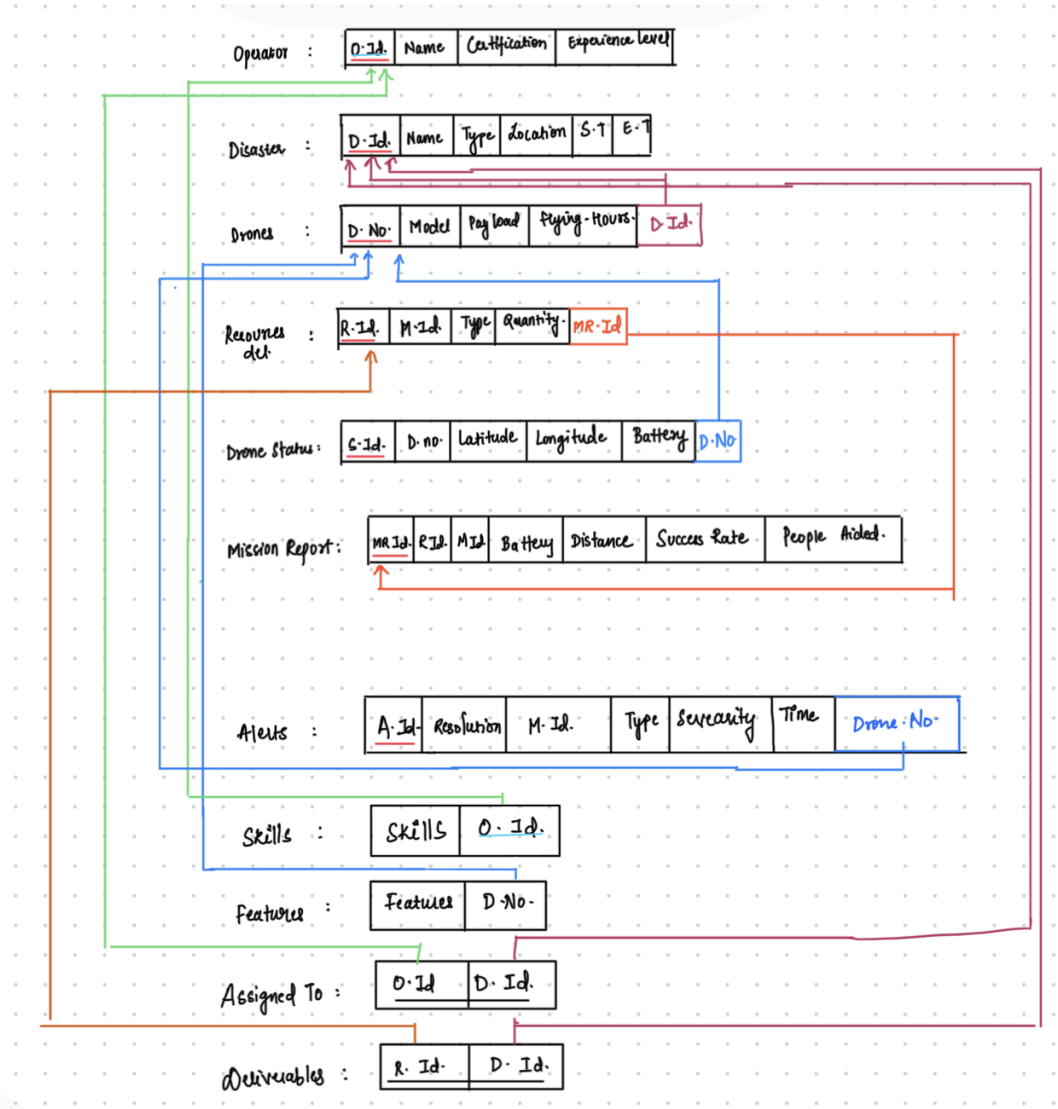
- **Visual Studio Code (VS Code)** – Main code editor for SQL, frontend, and backend files.

- **macOS Terminal** – For running MySQL server, Node.js backend, and other commands.

4. ER Diagram:



5. Relational Schema:



6. DDL, DML, DCL Commands:

Key DDL commands used include:

- **CREATE DATABASE** – To create the main `drone` database.
- **CREATE TABLE** – To define tables, their attributes, and data types.
- **ALTER TABLE** – Used to modify existing tables, such as adding foreign keys or constraints.

Key DML commands include:

- **INSERT** – To add new records into the database tables.
- **UPDATE** – To modify existing records, such as updating battery levels or mission outcomes.
- **DELETE** – To remove unnecessary or incorrect records from a table.
- **SELECT** – To retrieve data for backend queries, dashboards, and analytical reports.

Key DCL commands include:

- **GRANT** – To give specific privileges (such as SELECT only or full access) to users.
- **REVOKE** – To remove permissions when required.

7. CRUD operation Screenshots:

Below are some examples of crud operations like select, insert, delete, update:

```
mysql> INSERT INTO disaster (D_ID, Name, Type, Location, Start_Time)
    -> VALUES ('DS010', 'Test Disaster', 'Test', 'Test Location', NOW());
Query OK, 1 row affected (0.01 sec)

mysql> USE drone;
Database changed
mysql>
mysql> INSERT INTO DRONE (D_NO, Model, Payload, Flying_Hours, D_ID)
    -> VALUES ('DRN13', 'Phantom XZ', 3.50, 2.00, 'DS010');
Query OK, 1 row affected (0.00 sec)

mysql> select * from drone;
+-------+----------------+---------+--------------+-------+
| D_NO  | Model          | Payload | Flying_Hours | D_ID  |
+-------+----------------+---------+--------------+-------+
| DRN01 | Phantom 4      |    1.50 |       120.50 | DS001 |
| DRN02 | Matrice M300   |    5.00 |        45.30 | DS002 |
| DRN03 | Mavic Pro      |    0.50 |       200.00 | DS001 |
| DRN04 | Heavy Lifter X |   10.00 |        10.10 | DS003 |
| DRN05 | Swarm Scout    |    0.20 |       500.00 | DS004 |
| DRN06 | Hazard Mapper  |    3.00 |        15.00 | DS005 |
| DRN07 | Guardian Pro   |    2.50 |        80.00 | DS004 |
| DRN08 | Mini-Copter    |    0.10 |         5.00 | DS006 |
| DRN09 | DJI Inspire    |    2.00 |        30.00 | DS007 |
| DRN10 | Sensor Rover   |    0.10 |       550.00 | DS008 |
| DRN11 | Thermal Eagle  |    4.00 |        12.00 | DS009 |
| DRN12 | Cargo Master   |   15.00 |         5.00 | DS009 |
| DRN13 | Phantom XZ     |    3.50 |         2.00 | DS010 |
+-------+----------------+---------+--------------+-------+
13 rows in set (0.00 sec)

mysql>
```

```
mysql> UPDATE DRONE
    -> SET Flying_Hours = 10.00
    -> WHERE D_NO = 'DRN13';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT D_NO, Model, Flying_Hours
    -> FROM DRONE
    -> WHERE D_NO = 'DRN13';
+-------+-----------+--------------+
| D_NO  | Model     | Flying_Hours |
+-------+-----------+--------------+
| DRN13 | Phantom XZ |        10.00 |
+-------+-----------+--------------+
1 row in set (0.00 sec)

mysql> DELETE FROM DRONE
    -> WHERE D_NO = 'DRN13';
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM DRONE WHERE D_NO = 'DRN13';
Empty set (0.00 sec)
```

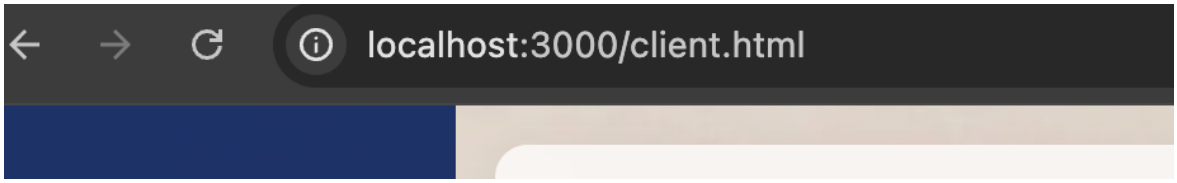
8. List of Functionalities / Features (Front End)

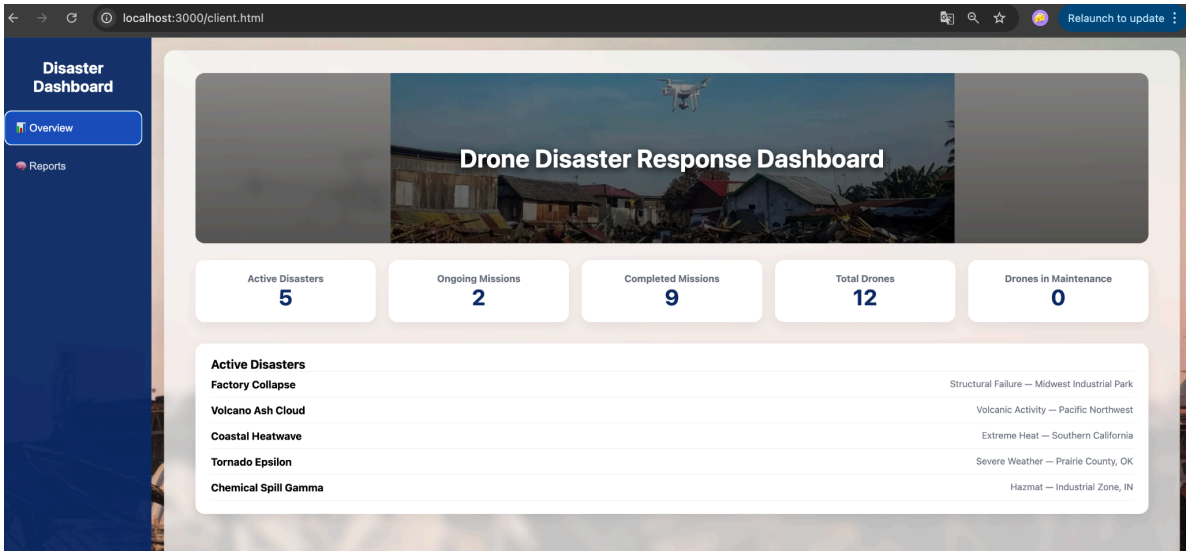
1. Overview Dashboard

- **Description:**
  The home screen shows a high-level **Drone Disaster Response Dashboard**. It displays summary cards such as *Active Disasters, Ongoing Missions, Completed Missions, Total Drones,* and *Drones in Maintenance*.
  Below that, it lists **Active & Recent Disasters** with their status

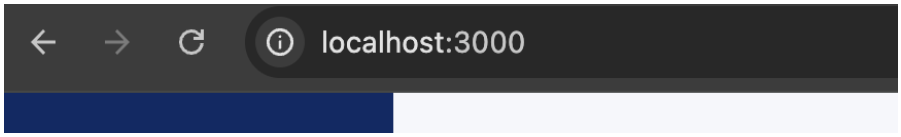
and start time, and a **Personnel Roster** showing operators and their roles.

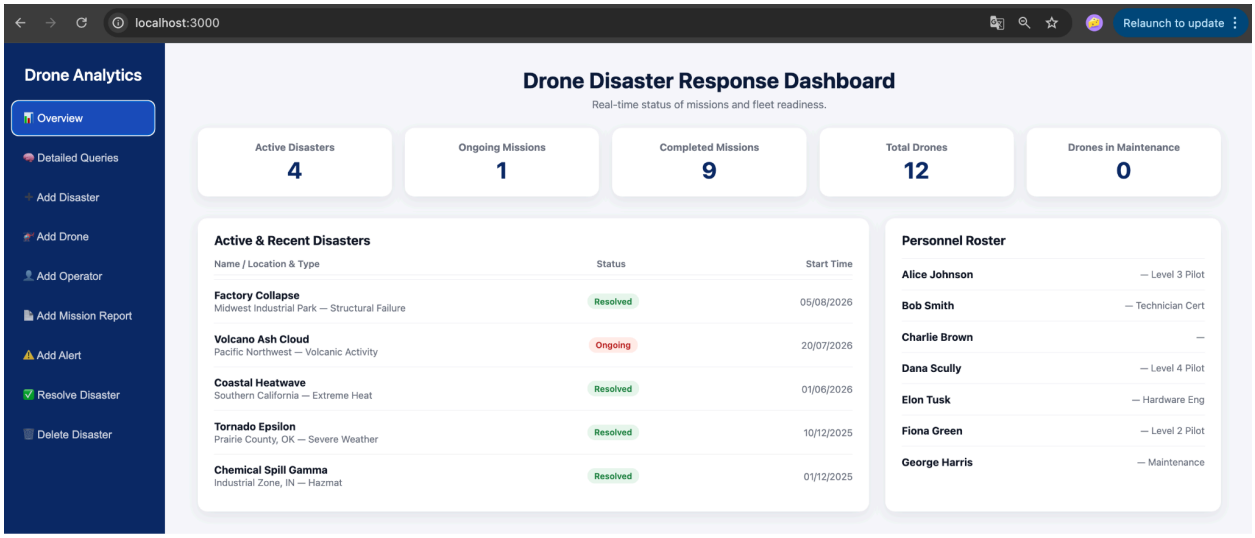
User Side:





Editor Side:





2. Detailed Query Reports

- **Description:**
  The **Detailed Queries** section visualizes analytical SQL results from the database. It includes:

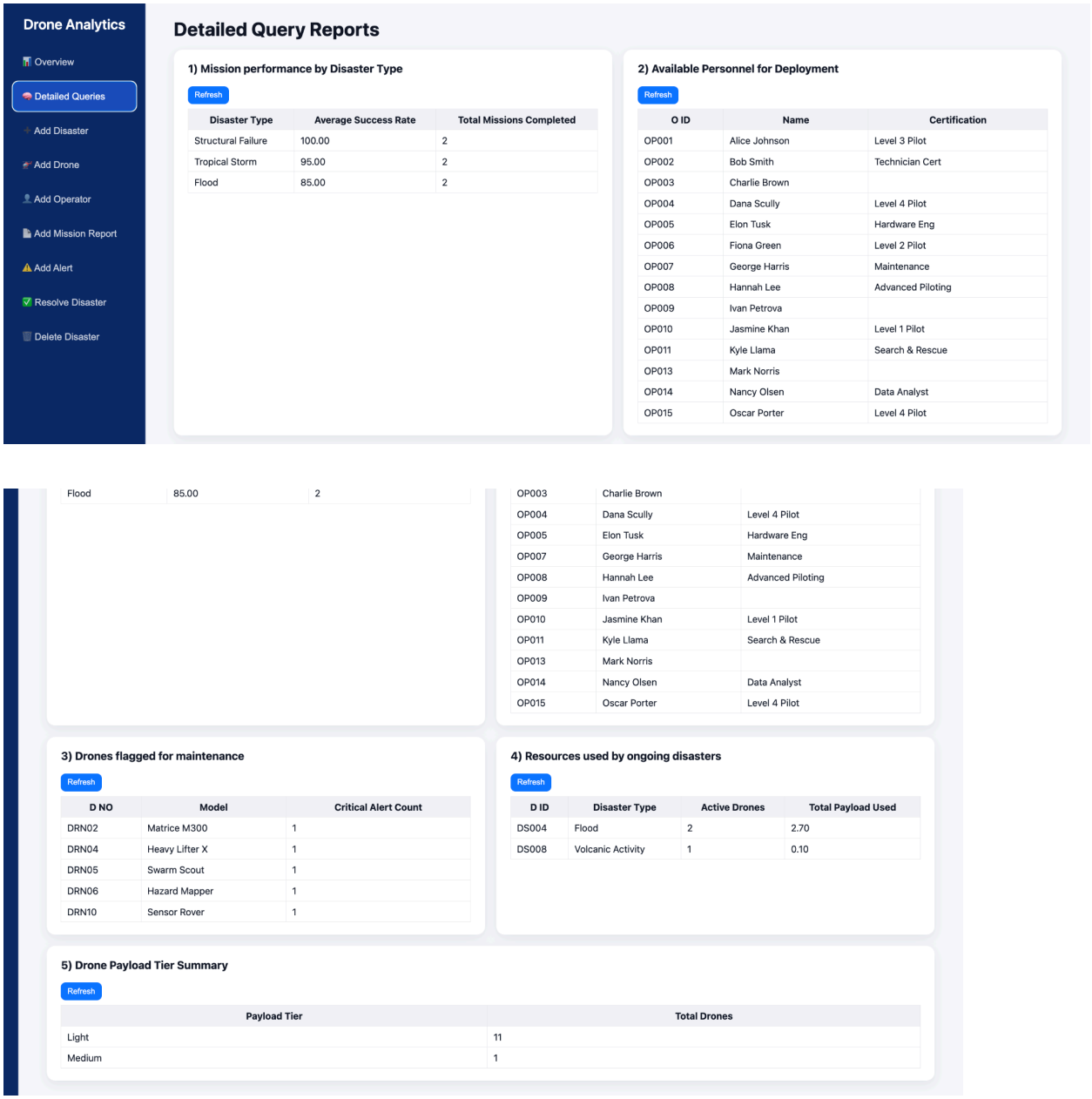
    1. **Mission performance by Disaster Type** – average success rate and total missions per disaster type.

    2. **Available Personnel for Deployment** – list of operators who are free to be assigned, along with their certifications.

    3. **Drones flagged for maintenance** – drones with critical alerts, showing model and alert count.

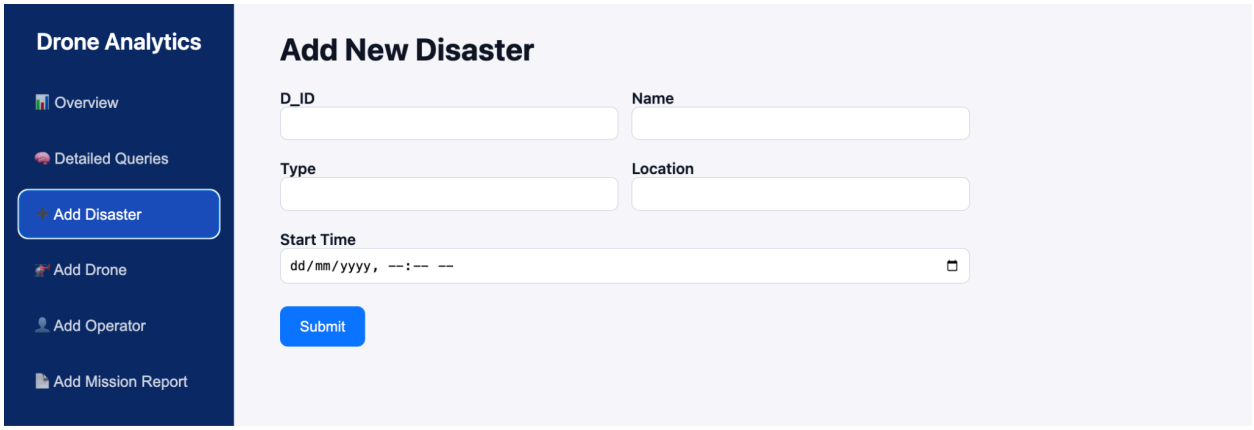
    4. **Resources used by ongoing disasters** – disasters with active missions and total payload used.

    5. **Drone Payload Tier Summary** – number of drones in each payload tier (light/medium/heavy).



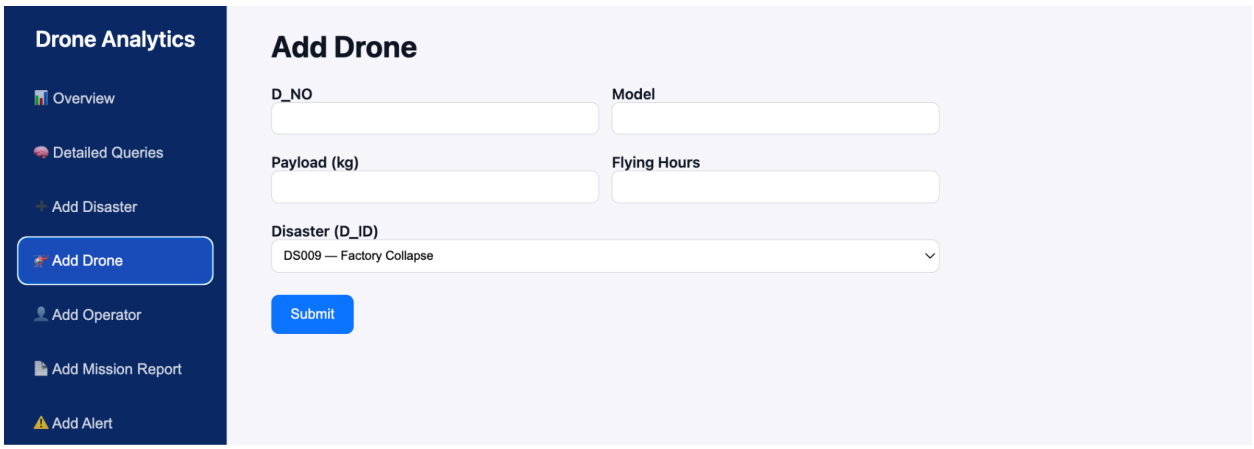
3. Add Disaster

- **Description:**
  The **Add Disaster** page provides a form to create a new disaster entry. The user can enter disaster ID, name, type, location, start time, and other details. On submission, the data is sent to the backend and stored in the `DISASTER` table.



4. Add Drone

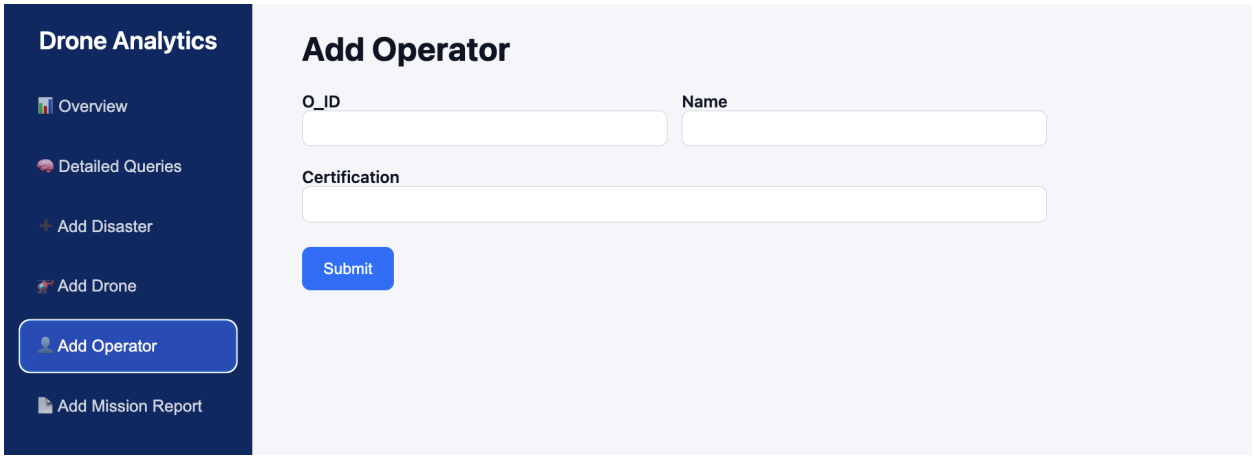
- **Description:**
  The **Add Drone** section allows the user to register a new drone into the system by entering its drone number, model, payload capacity, flying hours, and linked disaster ID. On submit, it creates a record in the DRONE table (subject to trigger/role rules in the database).



5. Add Operator

- **Description:**
  The **Add Operator** page lets the user add new drone operators,

including operator ID, name, certification level, and experience. This data is stored in the `OPERATOR` table and used later in reports (e.g., available personnel).



6. Add Mission Report

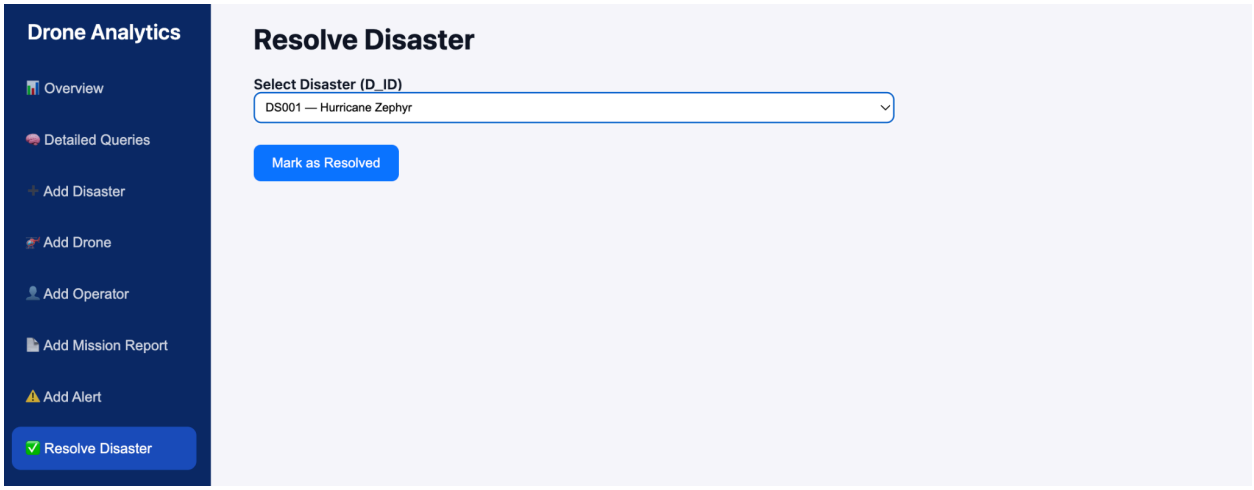
- **Description:**
  The **Add Mission Report** screen is used to log the outcome of completed missions. It collects mission report ID, battery remaining, distance covered, success rate, and number of people aided and stores them in the `MISSION_REPORT` table. These values are shown later in analytics and performance queries.

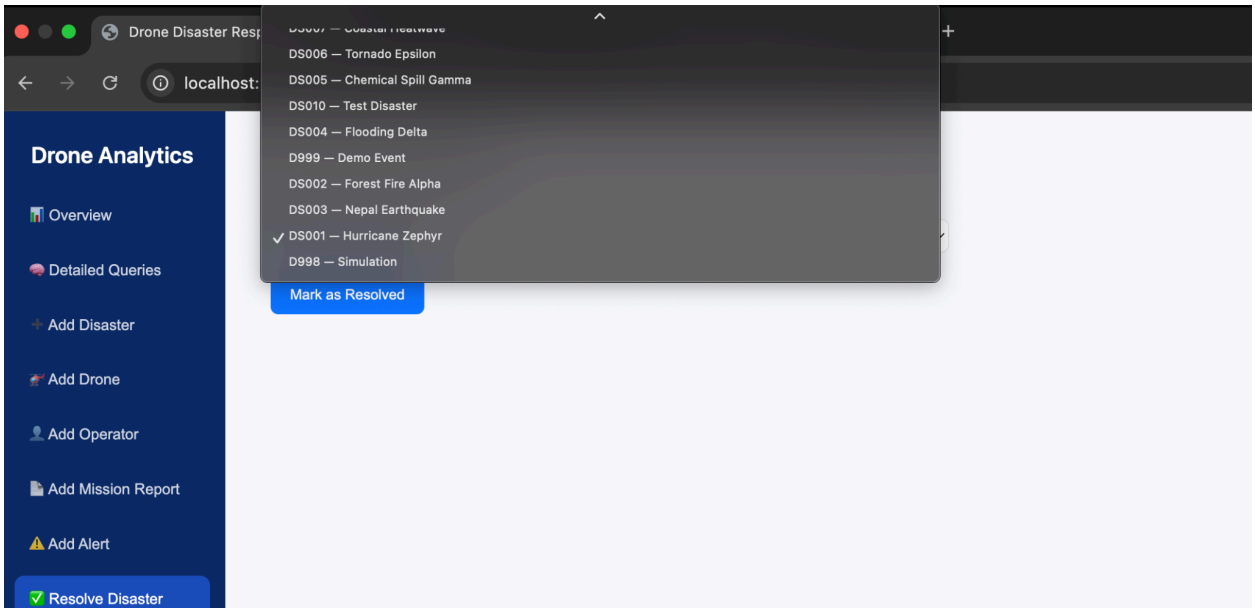
7. Add Alert

- **Description:**
  The **Add Alert** feature allows recording new alerts related to drones, such as low battery, critical battery, GPS error, or obstacle detection. The form saves alert ID, type, severity, time, linked mission, and drone number into the ALERTS table.

8. Resolve Disaster

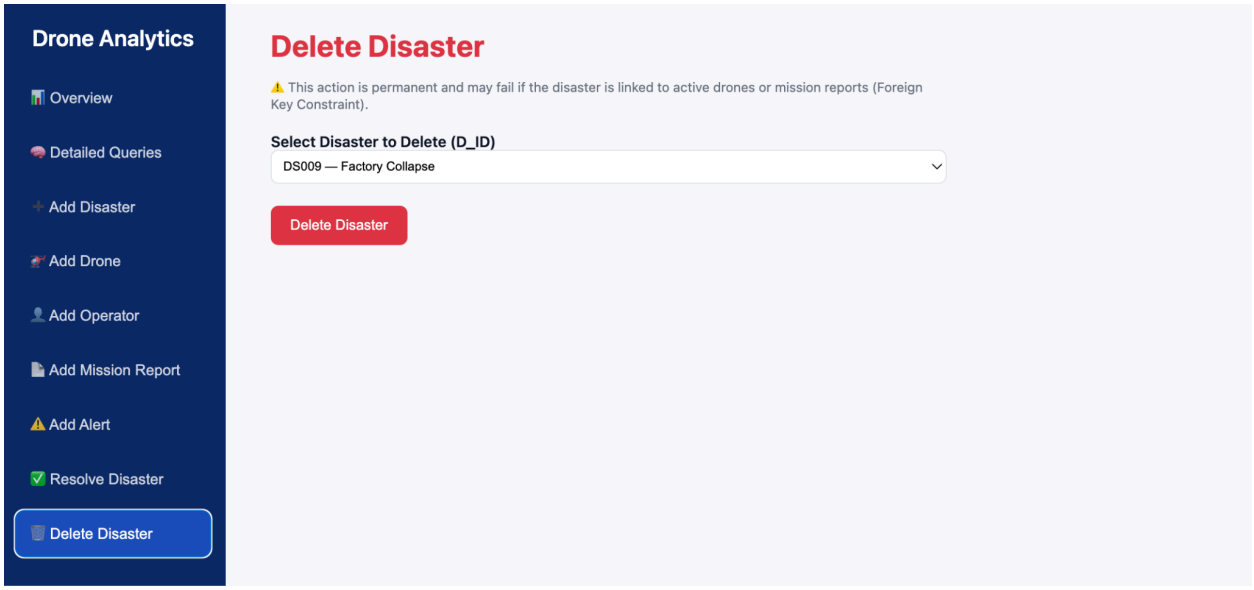
- **Description:**
  The **Resolve Disaster** page lets the user mark an existing disaster as resolved/closed. It typically allows selecting a disaster ID and then triggers an update on the DISASTER table (and possibly related status fields) via the backend. This demonstrates controlled update operations from the frontend.





9. Delete:

- Description: This "Delete Disaster" operation provides a utility within the dashboard for **permanently removing a disaster record** from the underlying MySQL database.



9. Triggers, Procedures / Functions, Nested Query, Join, Aggregate Queries:

10.1 Trigger

Name: `TRG_UPDATE_FLYING_HOURS`
Table: `DRONE_STATUS` (AFTER INSERT)

This trigger automatically updates the `Flying_Hours` of a drone whenever a new status entry is inserted into `DRONE_STATUS.`
For every new status log, it adds **0.50 hours** to the corresponding drone's `Flying_Hours` in the `DRONE` table.

Purpose:

- Keeps `Flying_Hours` in sync with actual usage.
- Removes the need for manual updates and reduces human error.

```
mysql> SELECT D_NO, Flying_Hours
    -> FROM DRONE
    -> WHERE D_NO = 'DRN01';
+-------+--------------+
| D_NO  | Flying_Hours |
+-------+--------------+
| DRN01 |       120.50 |
+-------+--------------+
1 row in set (0.00 sec)

mysql> INSERT INTO DRONE_STATUS (G_ID, D_NO, Latitude, Longitude, Battery)
    -> VALUES ('GS_TEST', 'DRN01', 30.00, -90.00, 99.00);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT D_NO, Flying_Hours
    -> FROM DRONE
    -> WHERE D_NO = 'DRN01';
+-------+--------------+
| D_NO  | Flying_Hours |
+-------+--------------+
| DRN01 |       121.00 |
+-------+--------------+
1 row in set (0.00 sec)
```

What this demonstrates:
Every status update increases flying hours by **0.5 hours** automatically (trigger execution).

10.2 User-Defined Function

Name: `FN_GET_EXPERIENCE_LEVEL(experience INT)`

This function takes an operator's years of experience as input and returns a **text rating**:

- `< 3 years` → *Novice*
- `3–7 years` → *Intermediate*
- `8–10 years` → *Senior*

- `> 10 years` → *Expert*

Purpose:

- Provides a readable seniority label for operators.
- Can be used in SELECT queries to display a “Seniority Rating” column along with raw experience.

```
mysql> SELECT FN_GET_EXPERIENCE_LEVEL(1) AS Level_1_Year,
    ->        FN_GET_EXPERIENCE_LEVEL(5) AS Level_5_Years,
    ->        FN_GET_EXPERIENCE_LEVEL(9) AS Level_9_Years,
    ->        FN_GET_EXPERIENCE_LEVEL(12) AS Level_12_Years;
+-------------+--------------+--------------+---------------+
| Level_1_Year | Level_5_Years | Level_9_Years | Level_12_Years |
+-------------+--------------+--------------+---------------+
| Novice       | Intermediate  | Senior        | Expert         |
+-------------+--------------+--------------+---------------+
1 row in set (0.00 sec)

mysql> SELECT
    ->     O_ID,
    ->     Name,
    ->     Experience,
    ->     FN_GET_EXPERIENCE_LEVEL(Experience) AS Experience_Level
    -> FROM OPERATOR;
+-------+--------------+------------+-----------------+
| O_ID  | Name         | Experience | Experience_Level |
+-------+--------------+------------+-----------------+
| OP001 | Alice Johnson |          2 | Novice           |
| OP002 | Bob Smith     |          1 | Novice           |
| OP003 | Charlie Brown |          1 | Novice           |
| OP004 | Dana Scully   |          5 | Intermediate     |
| OP005 | Elon Tusk     |          9 | Senior           |
| OP006 | Fiona Green   |          1 | Novice           |
| OP007 | George Harris |          1 | Novice           |
| OP008 | Hannah Lee    |          1 | Novice           |
| OP009 | Ivan Petrova  |          1 | Novice           |
| OP010 | Jasmine Khan  |         12 | Expert           |
| OP011 | Kyle Llama    |          1 | Novice           |
| OP012 | Laura Munez   |          1 | Novice           |
| OP013 | Mark Norris   |          1 | Novice           |
| OP014 | Nancy Olsen   |          1 | Novice           |
| OP015 | Oscar Porter  |          1 | Novice           |
+-------+--------------+------------+-----------------+
15 rows in set (0.00 sec)
```

What this demonstrates:

Function returns `Novice`, `Intermediate`, `Senior`, or `Expert` based on the experience field.

10.3 Stored Procedure

Name: `(disaster_id CHAR(10))SP_RESOLVE_DISASTER`

This procedure is used as a **single action to close a disaster**:

1. Checks if the disaster with the given `D_ID` exists and has `End_Time IS NULL` (ongoing).

2. If yes, updates `End_Time` to `NOW()` and returns a success message.

3. If not, returns a message saying the disaster is already resolved or invalid.

Purpose:

- Standardised way to “Resolve Disaster” from the frontend.

- Ensures only valid, ongoing disasters are closed.

```
mysql> SELECT D_ID, Name, End_Time
    -> FROM DISASTER
    -> WHERE D_ID = 'DS004';
+-------+---------------+----------+
| D_ID  | Name          | End_Time |
+-------+---------------+----------+
| DS004 | Flooding Delta | NULL     |
+-------+---------------+----------+
1 row in set (0.00 sec)

mysql> CALL SP_RESOLVE_DISASTER('DS004');
+-----------------------------------------------------------+
| Status                                                    |
+-----------------------------------------------------------+
| Disaster DS004 successfully closed at 2025-11-21 00:22:49 |
+-----------------------------------------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT D_ID, Name, End_Time
    -> FROM DISASTER
    -> WHERE D_ID = 'DS004';
+-------+---------------+---------------------+
| D_ID  | Name          | End_Time            |
+-------+---------------+---------------------+
| DS004 | Flooding Delta | 2025-11-21 00:22:49 |
+-------+---------------+---------------------+
1 row in set (0.00 sec)
```

Before:

After:

What this demonstrates:

The procedure validates if the disaster is active and updates `End_Time` automatically.

10.4 Nested Query

Query: *Available personnel for deployment.*

*This query uses a **nested subquery** inside the `WHERE` clause:*

- *Inner query finds operators currently assigned to **ongoing** disasters (`End_Time IS NULL`).*
- *Outer query returns operators **not in that list**, i.e., free for new deployment.*

Purpose:

- *Powers the "Available Personnel for Deployment" detailed report on the frontend.*

10.5 Join Queries & Aggregate Queries (Detailed Reports)

All five detailed reports on the frontend are based on **multi-table JOINs** combined with **aggregate functions** like `AVG`, `COUNT`, `SUM`, and `GROUP BY`.

1. **Mission performance by Disaster Type**
    - Joins: `DISASTER` + `DRONE` + `ALERTS` + `MISSION_REPORT`
    - Aggregates: `AVG(Success_Rate)`, `COUNT(MR_ID)`
    - *Shows average success rate and total missions for each d*isaster type.
2. **Available Personnel for Deployment**
    - Uses a **nested query + JOIN** between `ASSIGNED_TO` and `DISASTER`.
    - Filters operators who are not assigned to any ongoing disaster.

3. **Drones flagged for maintenance**
    - Joins: `ALERTS` + `DRONE`
    - Aggregates: `COUNT(A.A_ID)` per drone with `HAVING COUNT >= 2`
    - Identifies drones with multiple high/critical alerts for maintenance.
4. **Resources used by ongoing disasters**
    - Joins: `DISASTER` + `DELIVERABLES` + `RESOURCE_ACTION`
    - Aggregates: `SUM(RA.Quantity)` for each ongoing disaster (`End_Time IS NULL`)
    - Shows total resources deployed per disaster.
5. **Drone Payload Tier Summary**
    - Uses a `CASE` expression to categorize drones as **Light / Medium / Heavy** based on `Payload`.
    - Aggregates: `COUNT(D_NO)`, `AVG(Flying_Hours)`, `SUM(Payload)` grouped by payload tier.
    - Provides a high-level view of the fleet composition.

Purpose of these queries:

- Demonstrate use of **JOINs, GROUP BY, HAVING, aggregate functions, CASE, and nested subqueries**.

- Back the **“Detailed Query Reports”** section of the frontend with meaningful analytics.

Thank You!