# Introduction

The Drone-Based Medication distribution System is a full-stack solution designed to change the distribution of important medicinine in remote or high-demand areas. Leveraging cutting-edge drone technology, this system ensures that crucial medical supplies reach their destination promptly and effectively, eliminating reliance on older, slower transportation methods.

This project incorporates a powerful backend, developed using Node.js and the Serverless Framework v3, which installs AWS Lambda functions to oversee fundamental drone operations. The backend communicates with MongoDB Atlas using Mongoose for smooth data management and validation. Key functions include drone registration, state management (including automated transitions to RETURNING or FAILED states based on battery levels-extra action item by developer), medication loading, battery loss simulation, error logging, and thorough reporting.

On the frontend, a responsive React.js dashboard (made with Vite and Minimals UI based on Material UI components) delivers real-time insights into drone operations. Interactive charts made using Recharts illustrate drone battery levels and state distributions, while notifications warn users to key occurrences like low power or system failures. The user is allowed to register drones, load medicines and generate PDF reports based on a date range here as well.

Overall, the system not only displays excellent full-stack development methods but also handles real-world difficulties in hospital logistics by ensuring timely and reliable medicine delivery through new drone-based solutions.

# Overview of Architecture

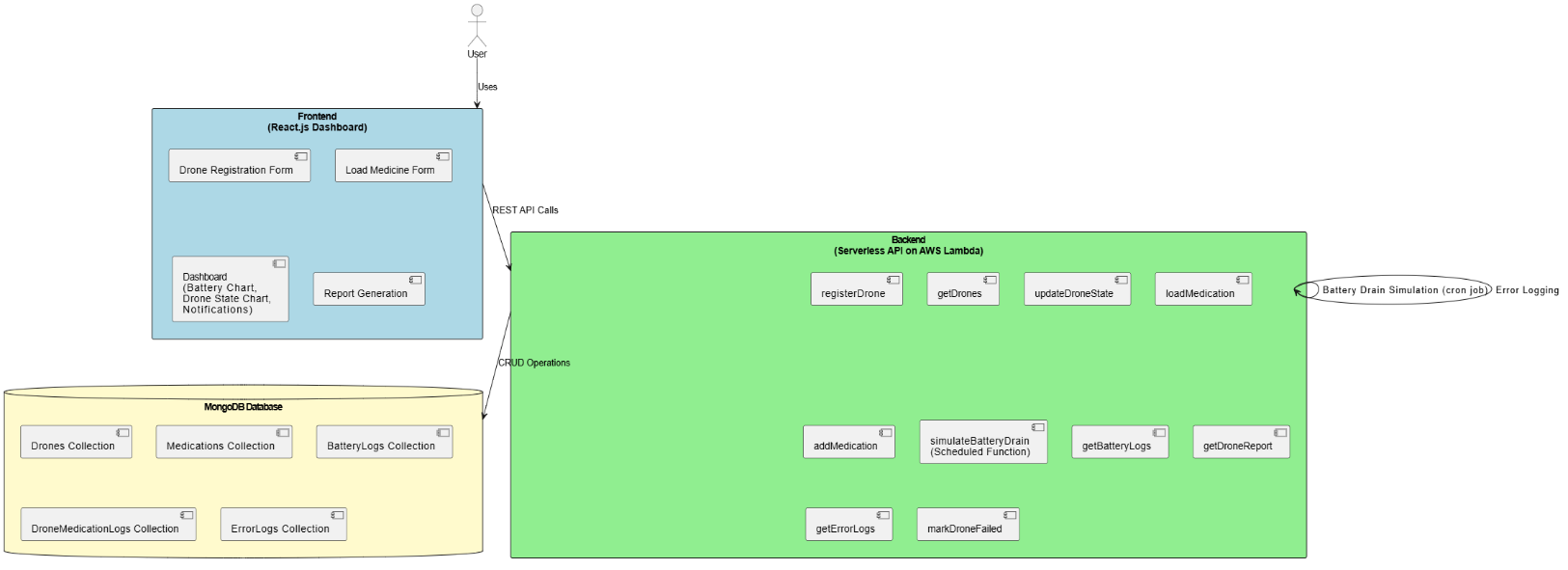


Figure 1 Architecture Diagram

The system comprises two primary components(excluding database):

## Backend

* Constructed with Node.js and the Serverless v3 Framework.
* **Deployment**: Being delivered to AWS Lambda; local development is facilitated with the serverless-offline plugin.
* **Database**: MongoDB (utilising MongoDB Atlas).
* Key Functionalities:
  + **Drone** **Registration**: Register new drones with distinct identities.
  + **Drone** **State** **Management**: Update and enforce drone state rules (e.g., transitioning to RETURNING when battery is low or labelling as FAILED).
  + **Medication** **Loading**: Load pharmaceuticals onto drones while adhering to weight restrictions and battery level regulations.
  + **Battery** **Drain** **Simulation**: A scheduled operation that mimics battery drainage for drones in DELIVERING and RETURNING stages and tracks battery statuses.
  + **Error** **Logging**: Log errors such as low battery events or system failures.
  + **Reporting**: Generate drone reports in JSON or PDF formats and obtain battery logs within a given time range.

## Frontend

* Built with: React.js (using Vite for quick development) and stylised with Minimals UI components.
* Key Functionalities:
  + **Dashboard**: Displays drone battery levels in a bar chart and the number of drones in each state in another bar chart.
  + **Notifications**: Shows alarms for drones with low battery or failure states.
  + **Forms**: Allows drone registration and medicament loading.
  + **Report** **Generation**: Provides options to generate PDF reports of the drone fleet.

# Data Model

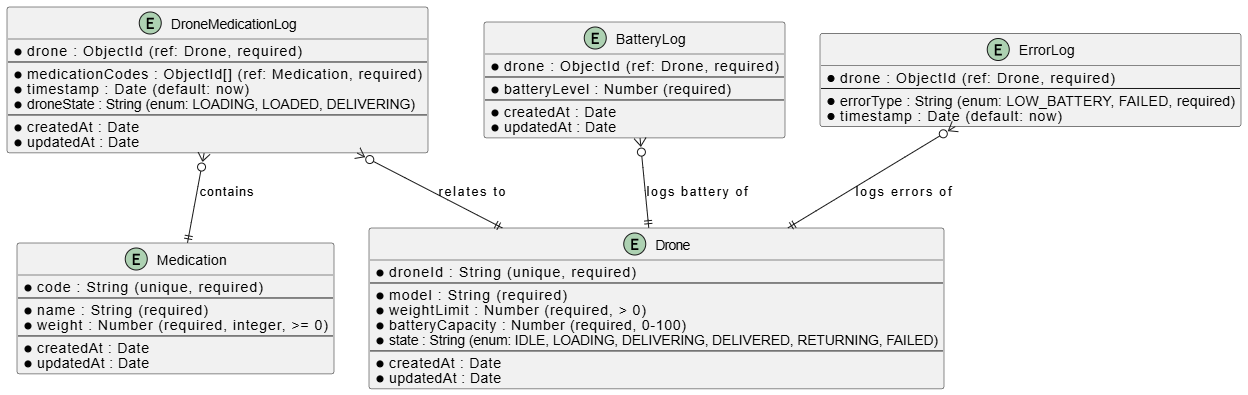


Figure 2 ER Diagram

## Entities

Each entity represents a Mongoose model

* Medication
* Drone
* DroneMedicationLog
* BatteryLog
* ErrorLog

Attributes are listed within the entity, including data types and constraints.

ObjectId refers to a mongoose object ID.

## Relationships

* **DroneMedicationLog }o--|| Drone**: One DroneMedicationLog relates to one Drone (one-to-one).
* **DroneMedicationLog }o--|| Medication**: One DroneMedicationLog includes one or more Medication (one-to-many).
* **BatteryLog }o--|| Drone**: One BatteryLog refers to one Drone.
* **ErrorLog }o--|| Drone**: One ErrorLog relates to one Drone.

# API

## ****Drone Management****

| HTTP Method | Path | Description |
| --- | --- | --- |
| **POST** | /drones | Register a new drone in the system. |
| **GET** | /drones | Retrieve a list of all registered drones and their current statuses. |
| **PUT** | /drones/{droneId}/state | Update the state of a specific drone (e.g., IDLE, LOADING, DELIVERING, etc.). |
| **PUT** | /drones/{droneId}/fail | Mark a drone as "FAILED" due to malfunctions. |

## ****Medication Management****

| HTTP Method | Path | Description |
| --- | --- | --- |
| **GET** | /dev/medications | Retrieve a list of available medications in the system. |
| **POST** | /medications | Add a new medication to the system. |
| **POST** | /drones/{droneId}/medications | Load medications onto a drone before delivery. |

## ****Battery & System Monitoring****

| HTTP Method | Path | Description |
| --- | --- | --- |
| **Scheduled** | (Every 2 minutes) | Simulate battery drain for drones in DELIVERING or RETURNING states. |
| **GET** | /batteryLogs?startTime={time}&endTime={time} | Retrieve battery logs for a specific time range. |

## ****Reports & Logs****

| HTTP Method | Path | Description |
| --- | --- | --- |
| **GET** | /droneReport?format={pdf/json} | **Generate a report of drone activities** (optional PDF or json). |
| **GET** | /errorLogs | Retrieve system error logs for debugging. |

# Features

1. **Core Drone Operations:**

* Register drones with attributes such as drone ID, model, weight limit, and battery capacity.
* Update drone states and enforce business rules (e.g., prevent loading if battery is below 25%).
* Load medicine products onto drones with validations for weight and battery capacity.

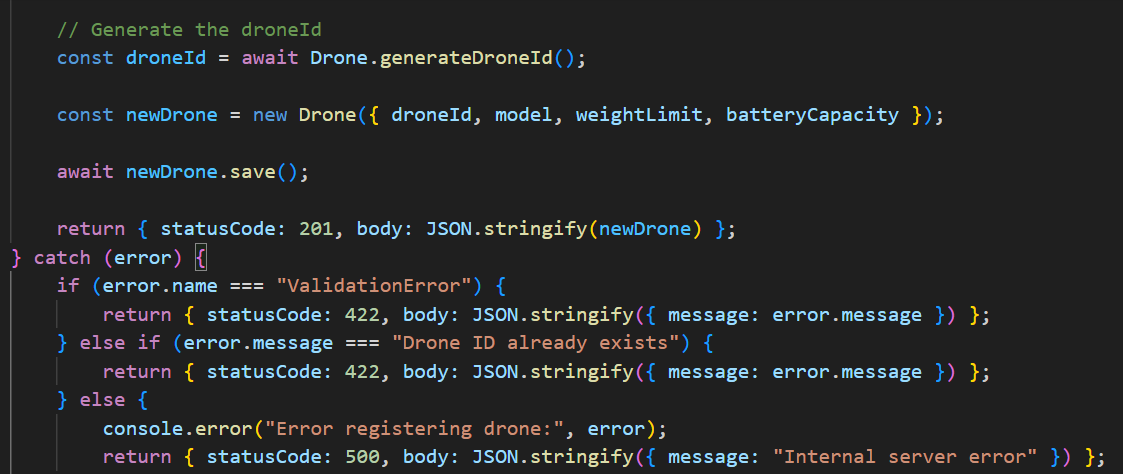


Figure 3 Register Drone

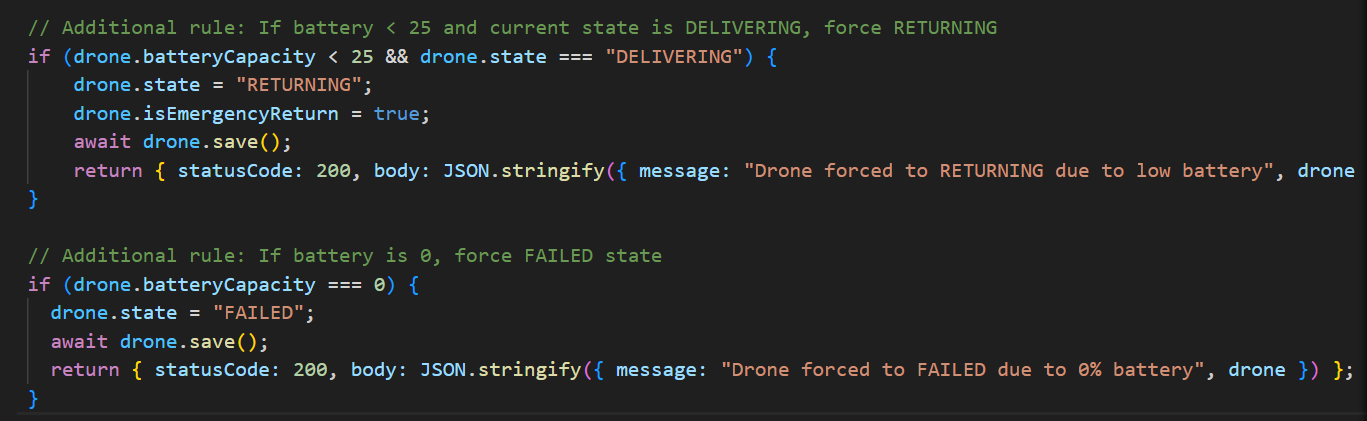


Figure 4 Drone State Management

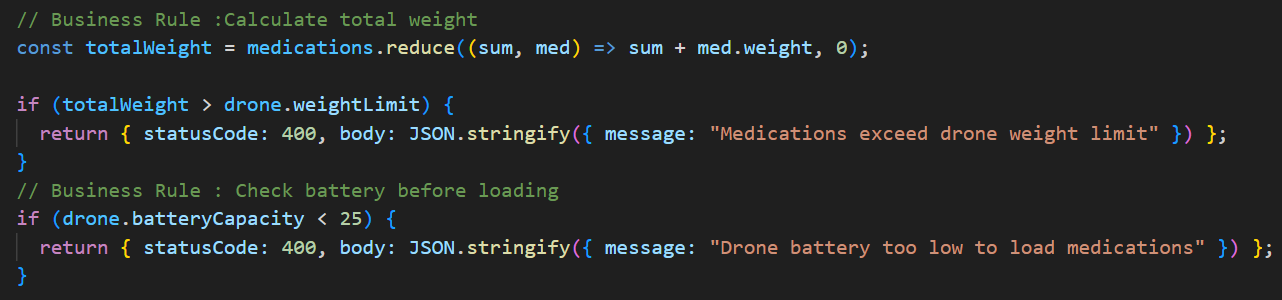


Figure 5 Load Medicine Validation

1. **Battery Management:**

* Simulate battery drain with scheduled jobs.
* Automatically switch drones to RETURNING if battery levels fall below a threshold.
* Mark drones as FAILED if battery reaches critical levels (e.g., 0-5%).

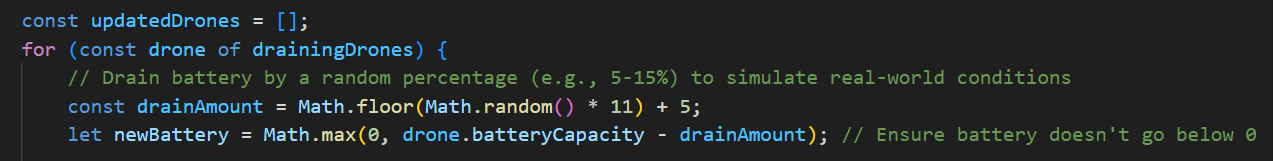


Figure 6 Battery Drain Simulation

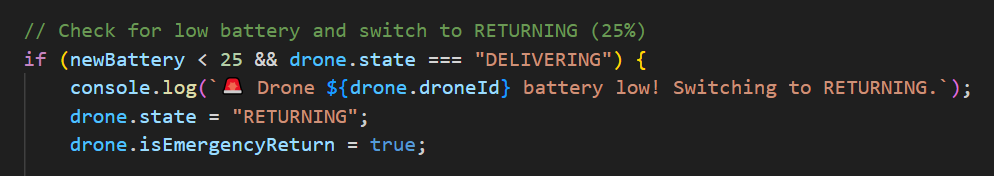


Figure 7 Return State when Battery is Low

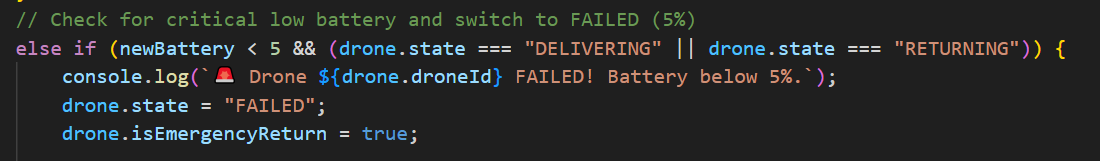
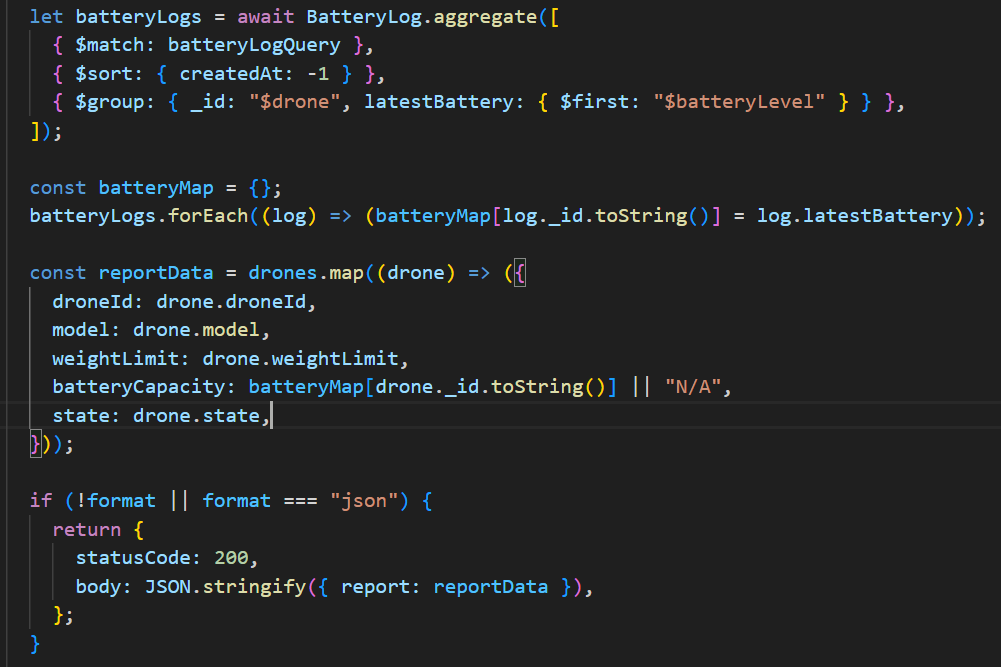


Figure 8 Failed State when Battery is critical

1. **Reporting & Logging:**

* Generate thorough reports on drone statuses and battery levels.
* Retrieve battery logs for defined time ranges.
* Log and view error events such as low battery warnings and drone failures.

Figure 9 Report Generation

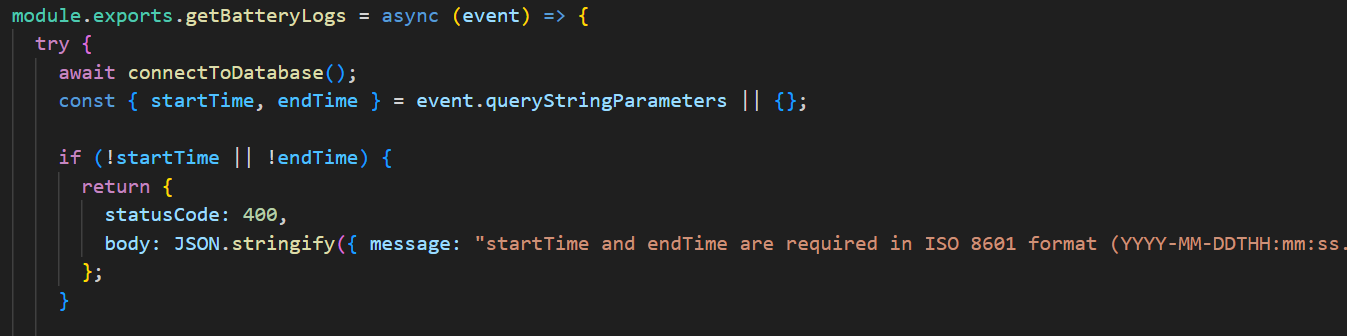


Figure 10 Logging Drone Battery

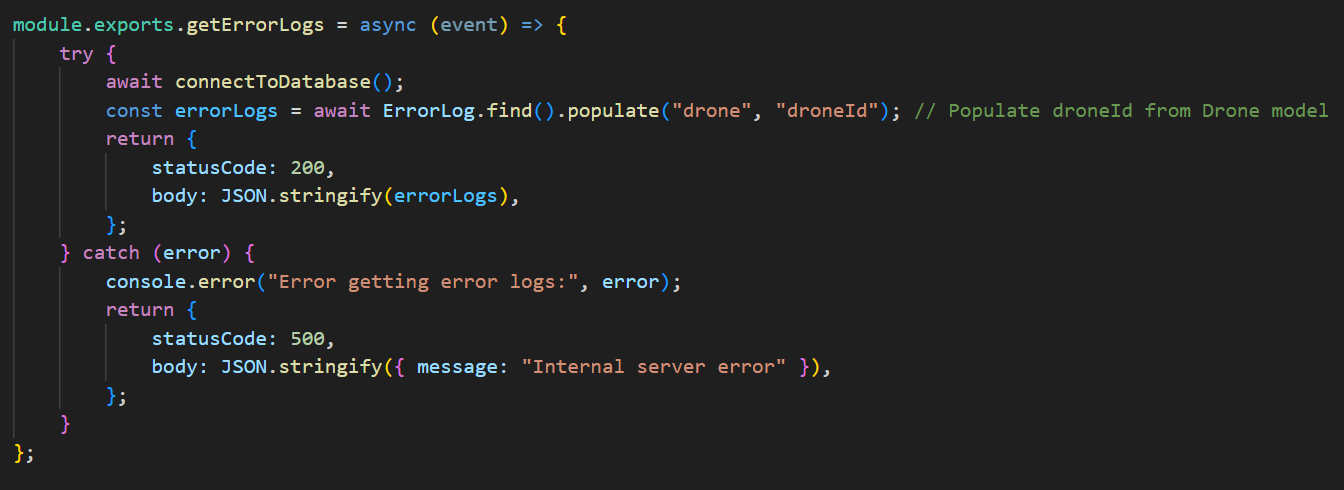


Figure 11 Error Logging

1. **User Dashboard:**

* Visualize current battery levels and drone states with interactive bar charts.
* Display notifications for operational concerns in real time.
* User interactive drone registration, medical loading and PDF report generation.

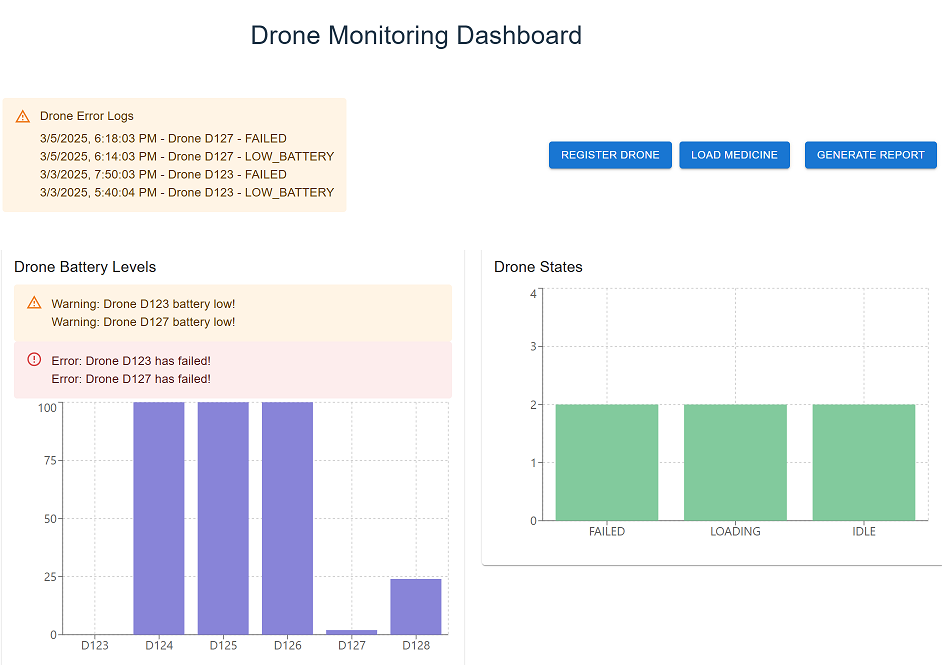


Figure 12 Dashboard

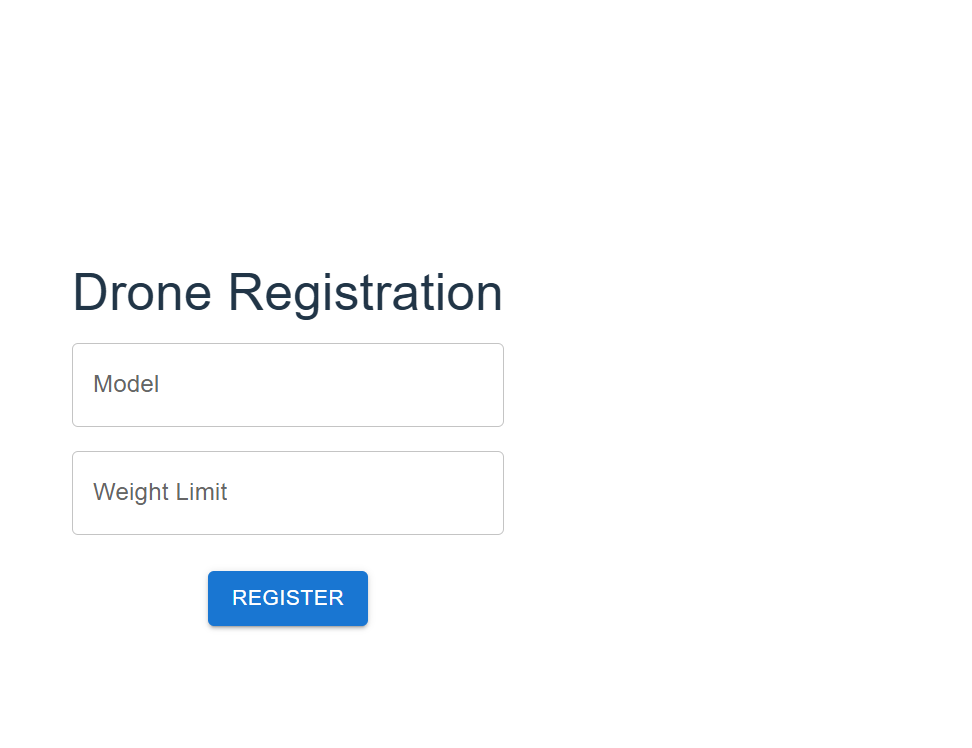


Figure 13Drone Registration Menu

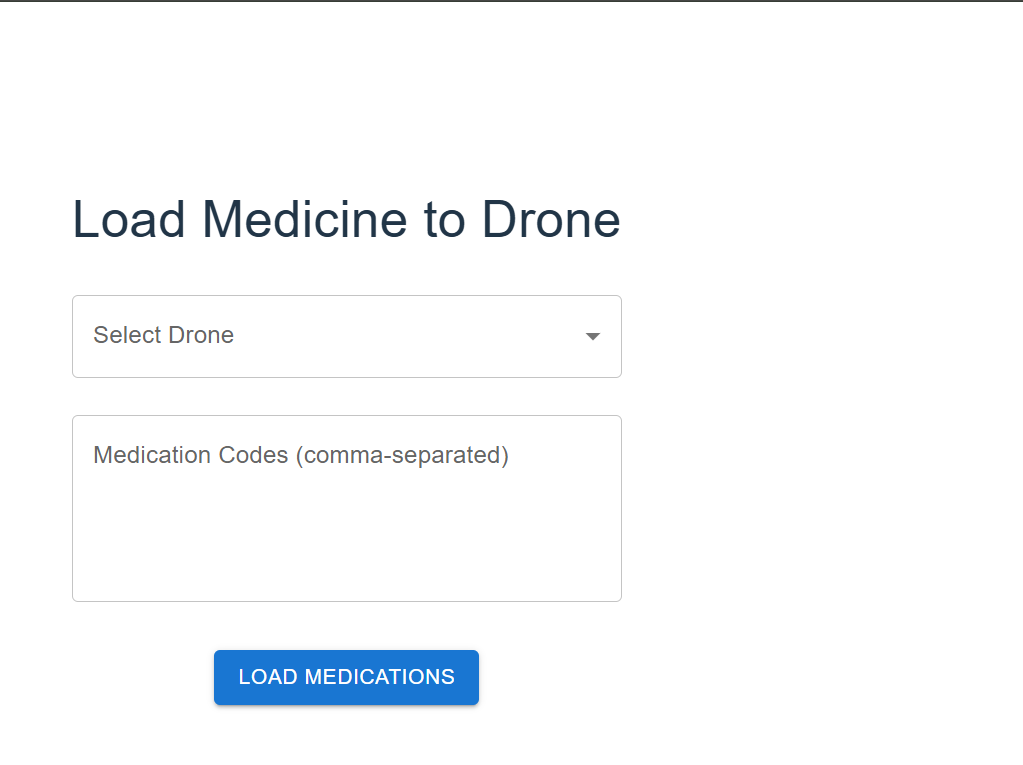


Figure 14 Medicine Loading Menu

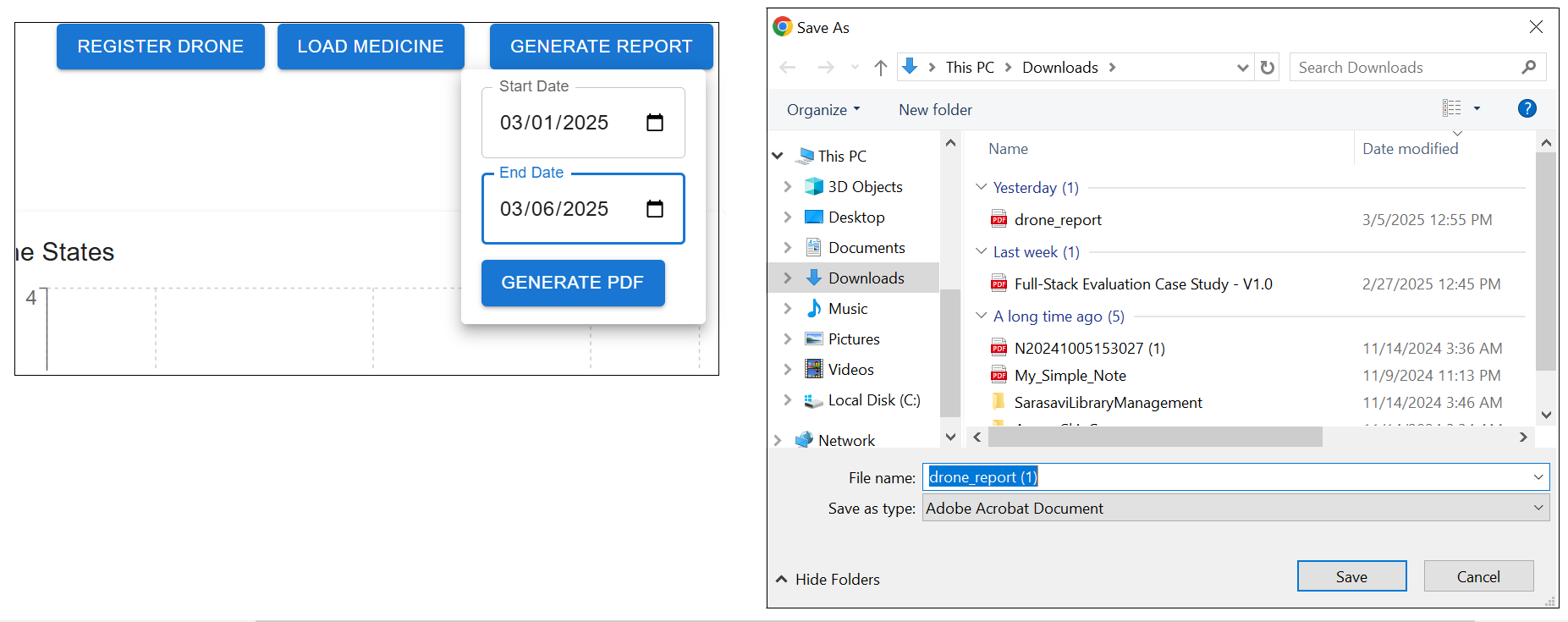


Figure 15 PDF Report Generation\\

# Technology Stack

**Backend:**

* Node.js v18
* Serverless Framework v3
* MongoDB (Atlas)
* Mongoose for MongoDB ORM v7

**Frontend:**

* React.js (via Vite) v18
* Vite v5
* Minimals UI (Material UI components) v5
* Recharts for charting v2
* Axios for API calls v1

# Assumptions

**Drone Identification & Registration**

* Each drone is automatically allocated a unique ID in the format DNNN (e.g., D001, D002).
* A drone must submit basic characteristics such as model, weight limit, and battery capacity during registration.
* The initial drone state is set to IDLE.

**Battery Management & Drone States**

* Battery capacity is indicated as a percentage (0–100%).
* Drones cannot be loaded with medicine if the battery level is below 25%.
* If a drone is in the DELIVERING state and its battery dips below 25%, it is automatically switched to the RETURNING state.
* If the battery level becomes critically low (below 5%), the drone is reported as FAILED.
* Battery drain is simulated by randomly lowering the battery by 5–15% per programmed cycle.

**Medication Handling**

* Medications contain unique alphanumeric numbers and are checked to meet a certain format.
* The system presupposes that medication details (name, code, weight) are appropriately provided and stored.
* When loading medicine onto a drone, the system assumes that the overall weight of the medications is within the drone’s weight restriction.

**API & Data Flow**

* All API requests and answers are in JSON format.
* The RESTful endpoints are built to meet the defined functional requirements and are used by the frontend dashboard.
* Endpoints such as registration, state update, medication loading, battery log retrieval, and report production are thought to be sufficient for managing drone operations.
* The backend relies on AWS Lambda functions via the Serverless Framework for execution.
* A scheduled battery drain simulation is assumed to execute successfully at given intervals (every 2 minutes in this scenario).

**Database & Environment**

* MongoDB Atlas is utilised as the primary database, and Mongoose is used for data modeling and validation.
* Collections for drones, medications, battery logs, drone medication logs, and error logs are considered to capture all relevant data.
* Environment variables (such as MONGO\_URI) are assumed to be properly specified using the serverless-dotenv-plugin.

**Frontend & User Experience**

* The React.js dashboard (made with Vite and Material UI components) is assumed to poll or refresh data at a regular interval (e.g., every 30 seconds) to present near real-time information.
* The dashboard uses Recharts for data visualization and Axios for making API requests.
* Users engaging with the dashboard are presumed to have proper access, and the UI is designed to indicate issues like low power or drone malfunctions.

**Reporting & Logging**

* The system may generate reports in JSON or PDF format, and the PDF generation is handled via pdfkit.
* Error logs are provided for crucial occurrences such as low battery warnings and drone failures.
* Battery logs are combined to display the most recent battery status for each drone.

**Deployment & Regional Settings**

* The backend is meant to run in the AWS environment, specifically in the ap-south-1 region.
* CORS is enabled for cross-origin requests between the frontend and backend.