Here is a detailed project plan, broken down into sequential phases and modules.

This plan is built around a **"crawl, walk, run"** philosophy. You will first build a functional, *manual* system, then make it *real-time*, and finally make it *predictive* with AI. This is the most stable and logical way to build a complex application.

Project: "CuraConnect" Predictive Operations Platform

Phase 0: Foundation & Core Setup (The "Plumbing")

Goal: Set up the project skeleton, database, and security. By the end, you will have a secure, running application that users can log into.

- **Module 0.1: Project Skeleton & Version Control**
    - **Task:** Create the main project directory.
    - **Task:** Initialize a git repository.
    - **Task:** Create two sub-folders:
        - backend/: This will hold your Flask application.
        - frontend/: This will hold your React application.
    - **Task:** Create your docker-compose.yml file to manage your services.
- **Module 0.2: Database & Core Models**
    - **Task:** Add **PostgreSQL** to your docker-compose.yml.
    - **Task:** In your Flask app, install and configure Flask-SQLAlchemy and Flask-Migrate.
    - **Task:** Define your *core* models.py with the first, most essential tables:
        - User: For staff (doctors, nurses, admins, housekeepers). Include username, hashed_password, and role (e.g., "nurse", "admin").
        - Patient: For patient records.
        - Bed: For all hospital beds. Include room_number, unit (e.g., "ICU", "Cardiac"), and status (e.g., "available", "occupied").
- **Module 0.3: Authentication & Security**
    - **Task:** Install and configure Flask-JWT-Extended for token-based authentication.
    - **Task:** Create your first API endpoints in Flask:
        - POST /api/auth/login: Takes username and password, returns a JWT token.
        - POST /api/auth/register-staff (Protected, admin-only): Creates a new User.

- **Task:** In your frontend/ React app, create a simple LoginPage that can call the login API and save the token.

Phase 1: The "Digital Front Door" (Patient Intake)

Goal: Allow a new patient to be created in the system via a simple web app. This phase focuses on the **Patient Intake** module.

- **Module 1.1: Patient Registration API**
  - **Task:** Create a *public* API endpoint: POST /api/patient/register.
  - **Task:** This endpoint takes patient data (name, DOB, etc.) and creates a new Patient in your PostgreSQL database.
- **Module 1.2: Patient Registration UI (The Web App)**
  - **Task:** Create a new, simple React application (or a separate route in your main app) that is mobile-friendly.
  - **Task:** This is the form the patient fills out. It should be simple, clean, and public (no login required).
  - **Task:** On submit, it calls your POST /api/patient/register endpoint.
  - **Task:** Generate QR codes that link directly to this web page.
- **Module 1.3: AI-Assisted Intake (OCR)**
  - **Task:** Add **Celery** and **Redis** to your docker-compose.yml.
  - **Task:** Install pytesseract or easyocr in your Flask app.
  - **Task:** Modify your registration form to include an <input type="file"> for an insurance card.
  - **Task:** Modify the /api/patient/register endpoint:
    1. It saves the patient data with a status of "pending_verification".
    2. It sends the uploaded image to a new **Celery background task** (process_ocr_image).
  - **Task:** The Celery task runs the OCR, extracts the text, and updates the Patient record in the database.

Phase 2: Core Operations (The *Manual* Hospital)

Goal: Build the internal staff dashboard. By the end, a logged-in manager can *manually* assign a patient to a bed and discharge them. This builds the foundation for your **Bed Management** and **Staff Management** modules.

- **Module 2.1: The "Hospital View" API (Read-Only)**

    - **Task:** Create *protected* (login required) API endpoints:
        - GET /api/beds: Returns a JSON list of all beds, their status, and the patient in them (if any).
        - GET /api/patients: Returns a list of all patients, including their current status (e.g., "waiting", "admitted").

- **Module 2.2: The Staff Dashboard UI (Read-Only)**
    - **Task:** In your React app, create the main "Hospital Dashboard" page (this page is protected by your login).
    - **Task:** Create a "Bed Map" component that fetches from /api/beds and displays all beds visually (e.g., as colored boxes).
    - **Task:** Create a "Patient List" component that fetches from /api/patients.
- **Module 2.3: The "Action" API (Manual Control)**
    - **Task:** Create the key endpoints for manual patient flow:
        - POST /api/patient/admit: Takes a patient_id and bed_id. Logic: assign the patient to the bed, change bed status to "occupied".
        - POST /api/patient/discharge: Takes a patient_id. Logic: remove the patient from the bed, change bed status to "pending_cleaning".
        - POST /api/bed/mark-clean: Takes a bed_id. Logic: change bed status from "pending_cleaning" to "available".
- **Module 2.4: The Interactive Dashboard UI**
    - **Task:** In your React "Bed Map", make the beds clickable.
    - **Task:** Add buttons (e.g., "Admit Patient", "Discharge Patient") that call the new "Action" APIs and refresh the dashboard on success.
    - **At this point, you have a functional, *manual* hospital management system.**

Phase 3: Real-Time Workflows (The *Automated* Hospital)

Goal: Automate the manual steps from Phase 2. This is where your modules truly start to connect.

- **Module 3.1: The Task Model**
    - **Task:** Add a Task model to your models.py (flask db migrate).
    - **Task:** It should include description, status ("pending", "complete"), priority, and foreign keys for assigned_to_user_id and patient_id.
- **Module 3.2: Real-time Backend (SocketIO)**
    - **Task:** Integrate Flask-SocketIO into your Flask server.

- **Task:** Modify your "Action" APIs from Module 2.3:
    - When /api/patient/discharge is called, it now *also* automatically creates a new Task in the database (e.g., "Clean Room 405", assigned to role="housekeeping").
    - After saving the task, use SocketIO to emit a 'new_task' event to the housekeeping staff.
    - When /api/bed/mark-clean is called (which will be a Task completion), it should emit a 'bed_status_update' event to all managers.

- **Module 3.3: Real-time Frontend (SocketIO)**
    - **Task:** In your React dashboard, add a "My Tasks" component.
    - **Task:** This component connects to SocketIO and *listens* for the 'new_task' event. When it receives one, it adds the task to the list *instantly* (no refresh needed).
    - **Task:** Your "Bed Map" component now listens for the 'bed_status_update' event and automatically changes the bed's color (e.g., from "Blue" to "Green") in real-time.

Phase 4: The Predictive "Brain" (The *Smart* Hospital)

Goal: Add the AI models to move your app from reactive to *predictive*.

- **Module 4.1: The AI Models (Offline)**
    - **Task:** Create a notebooks/ folder.
    - **Task:** Use Jupyter Notebook, Pandas, and Scikit-learn to train your models on (exported) historical data.
    - **Task:** Create and save discharge_model.pkl (predicts length of stay) and staffing_model.pkl (predicts ER admissions).
- **Module 4.2: The Predictive Engine (Celery)**
    - **Task:** Create a new **scheduled Celery task** (using Celery Beat) to run every hour.
    - **Task:** This task loads discharge_model.pkl, queries the DB for all current inpatients, and updates a new predicted_discharge_date column in the Patient table.
    - **Task:** When the prediction is made (e.g., predicted_discharge_date is today), it *automatically* changes the Bed status to "pending_discharge" (the yellow color).
    - **Task:** This status change emits a 'bed_status_update' via SocketIO, just like a manual change would, triggering the discharge workflow.
- **Module 4.3: The AI-Powered API**
    - **Task:** Create a new endpoint: GET /api/staffing/forecast.

    - **Task:** This endpoint loads staffing_model.pkl, runs a prediction for the next 24 hours, and returns a JSON forecast.

- **Module 4.4: The Predictive UI**
    - **Task:** In your React dashboard, add a "Staffing Forecast" page that calls and displays the forecast data.
    - **Task:** Your "Bed Map" now has a new color: "Yellow" (Pending Discharge), which is set *automatically* by the AI.

Phase 5: Deployment & Integration

Goal: Prepare the application for the real world.

- **Module 5.1: Production Deployment**
    - **Task:** Configure **Gunicorn** as your production WSGI server for Flask.
    - **Task:** Configure **Nginx** as a reverse proxy to manage traffic to Gunicorn and serve your *built* React static files.
    - **Task:** Harden all security, set environment variables, and run the full docker-compose.yml stack on a secure server.
- **Module 5.2: EHR Integration (The Final Hurdle)**
    - **Task:** This is a major R&D step. You must investigate the hospital's existing Electronic Health Record (EHR) system.
    - **Task:** Your goal is to use their **FHIR** (or HL7) API.
    - **Task:** Create new Celery tasks to periodically sync data:
        - **Pull:** Read new patient admissions from the EHR into your database.
        - **Push:** Push critical updates (like bed assignments and discharge status) *from* your app *back* to the EHR. This ensures your app is a "helper" and the EHR remains the "single source of truth."