1. Project Vision & Core Modules

To create a unified, web-based operations platform that moves a mid-scale hospital from reactive to predictive. It will consist of three core, integrated modules built around a central API:

1. **AI-Powered Patient Intake:** A patient-facing web app for fast, accurate, AI-assisted registration.
2. **AI-Driven Staff Management:** An internal dashboard for optimizing and dispatching staff based on predictive models.
3. **AI-Driven Patient Flow & Bed Management:** A "logistics hub" dashboard that predictively manages patient movement and bed allocation.

---

2. System Architecture: The API-First Model

We will **not** build a single, monolithic "laptop app." We will build a **Web Application** based on a **Client-Server** architecture.

- **The "Server" (Backend):** This is your **Python/Flask application**. It will run on a central server. It has no user interface. Its only job is to be the "brain" and provide a **REST API**. It will handle all business logic, database operations, and AI predictions.
- **The "Clients" (Frontends):** These are two separate **JavaScript web applications** that "talk" to your Flask API.
    1. **Patient App:** A simple, mobile-friendly website for registration.
    2. **Hospital App:** A complex, data-rich dashboard for staff.

This "decoupled" approach is the modern standard. It lets you use the best tool for each job (Python for AI/data, JavaScript for fast UIs) and is highly scalable.

---

3. Core Technical Stack

This stack is chosen to leverage your existing Python skills while providing a powerful, real-time, and modern application.

Layer	Technology	Why?
Backend Framework	**Flask**	Your core strength. Perfect for building a robust API.
Database	**PostgreSQL**	A powerful, open-source SQL database that can handle complex hospital data.

Layer	Technology	Why?
Database ORM	**Flask-SQLAlchemy**	Lets you interact with your database using Python objects, which is much cleaner than writing raw SQL.
API Standard	**Flask-RESTful** or **Flask-Smorest**	Helps you build a clean, well-documented REST API for your frontend to consume.
Authentication	**Flask-JWT-Extended**	Secure, token-based authentication. Essential for protecting patient data (HIPAA) and securing your API.
Real-time Engine	**Flask-SocketIO**	**This is critical.** It gives you a real-time (WebSocket) connection to your frontend dashboards. When a bed status changes or a new task is created, the server *pushes* the update to the dashboard instantly without the user hitting "refresh."
Async Tasks	**Celery & Redis**	**This is also critical.** You can't run a 30-second AI prediction inside a web request. Celery is a task queue that handles long-running processes (like AI models, sending emails) in the background. Redis acts as the "broker" to pass messages to Celery.
Frontend Framework	**React.js** (Recommended) or **Vue.js**	You *need* a JavaScript framework to build a complex dashboard. React is the industry standard with a huge ecosystem. This will be a separate project/folder from your Flask app.
Frontend UI	**Material-UI (MUI)** or **Bootstrap**	A pre-built component library so you don't have to write CSS from scratch. Gives you a professional-looking hospital dashboard *fast*.
AI / ML	**Scikit-learn, Pandas**	For your core predictive models (Discharge, Staffing, Burnout).
AI / OCR	**Pytesseract** or **EasyOCR**	For extracting text from uploaded insurance cards and IDs.
Deployment	**Docker, Gunicorn, Nginx**	The modern standard. **Docker** packages your Flask app, PostgreSQL, and Redis into containers. **Gunicorn** is the Python web server (replaces the weak Flask dev server). **Nginx** is the web server that faces the internet, manages security, and directs traffic to Gunicorn.

---

4. Detailed Database Schema (High-Level)

Your Flask-SQLAlchemy models will look something like this. These tables are the "single source of truth" for your entire application.

Python

```
# (This is pseudo-code for your models.py)


# For Staff Management
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String, unique=True)
    hashed_password = db.Column(db.String)
    role = db.Column(db.String) # e.g., 'doctor', 'nurse', 'admin', 'housekeeper'
    current_status = db.Column(db.String) # e.g., 'available', 'on-task', 'on-break'
    tasks = db.relationship('Task', backref='assignee')


# For Patient Intake & Flow
class Patient(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String)
    last_name = db.Column(db.String)
    dob = db.Column(db.Date)
    insurance_policy_num = db.Column(db.String)
    # ... other registration data
    current_bed_id = db.Column(db.Integer, db.ForeignKey('bed.id'))
    predicted_discharge_date = db.Column(db.DateTime)


# For Bed Management
class Bed(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    room_number = db.Column(db.String)
    unit = db.Column(db.String) # e.g., 'Cardiac', 'ER', 'ICU'
    status = db.Column(db.String) # 'available', 'occupied', 'pending_cleaning'
```

```
    patient = db.relationship('Patient', backref='bed', uselist=False)


# For Task Dispatching
class Task(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    description = db.Column(db.String) # e.g., 'Clean Room 405', 'Transport Patient 72 to Radiology'
    status = db.Column(db.String) # 'pending', 'in_progress', 'complete'
    priority = db.Column(db.String) # 'low', 'medium', 'high'
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    patient_id = db.Column(db.Integer, db.ForeignKey('patient.id'))
```

---

5. API Structure (High-Level Examples)

Your Flask app will expose these endpoints. Your React frontend will call them.

- POST /api/login (Takes username/password, returns a JWT token)
- POST /api/patient/register (Takes form data, runs OCR, creates new Patient)
- GET /api/beds/dashboard (Returns a JSON list of all beds and their status. This will be pushed via SocketIO instead of a GET request.)
- GET /api/staff/tasks/{user_id} (Returns all tasks for one user)
- POST /api/tasks (Admin creates a new task)
- PUT /api/tasks/{task_id}/complete (Staff marks a task as done)
- GET /api/staffing/forecast (Runs the staffing AI model and returns a JSON forecast)

---

6. Detailed Module Approach

Module 1: AI-Powered Patient Intake

1. **Frontend (React):** Create a simple, mobile-first form (HTML) for registration. Use JavaScript's fetch API to send the form data, including the image files from <input type="file">.
2. **Backend (Flask):** Create a /api/patient/register endpoint.
3. **OCR Task (Celery):** The endpoint receives the data. It **immediately** sends the image file to a **Celery worker task**. This frees up the web server.

4. **Celery Worker:** The worker runs pytesseract on the image to get the text. It cleans the text, finds the policy number/name, and updates the Patient record in the PostgreSQL database.
5. **Insurance API:** (Future Step) The worker could then call a 3rd-party insurance verification API.

Module 2: AI-Driven Staff Management

1. **AI Model (Python):** Use scikit-learn to train a **Staffing Model** (RandomForestClassifier) on historical data to predict ER admissions. Save this model as a .pkl file.
2. **AI Model (Python):** Train a **Burnout Model** (LogisticRegression) on anonymized staff data (overtime, hours, task count) to predict burnout risk.
3. **Backend (Flask):** Load the models into your app. Create a /api/staffing/forecast endpoint that calls staffing_model.predict().
4. **Task Dispatch (Real-time):**
    - **Frontend (React):** A manager's dashboard. The React app opens a persistent **SocketIO connection** on load: const socket = io.connect('http://your-server');
    - **Backend (Flask):** When an admin creates a new task (e.g., "Clean Room 405"), your Flask app:
        1. Saves the Task to the database.
        2. Finds the best staff member (e.g., nearest available housekeeper).
        3. Uses **flask_socketio.emit()** to send the new task *only* to that user's specific dashboard. emit('new_task', {task_data}, room=that_user_id)
    - **Frontend (React):** The React dashboard has a listener socket.on('new_task', (data) => ...) that receives the task and instantly adds it to the user's task list without a page refresh.

Module 3: AI-Driven Patient Flow & Bed Management

1. **AI Model (Python):** This is your most important model. Use scikit-learn to train a **Discharge Prediction Model** (a RandomForestRegressor) on historical patient data (diagnosis, age, labs, days in) to predict predicted_length_of_stay.
2. **Scheduled Task (Celery Beat):** You will configure Celery to run a **scheduled task** every hour.
3. **The Task:** This task will:
    1. Query the database for all current inpatients.
    2. Run the **Discharge Model** on each patient.
    3. Update the predicted_discharge_date in the Patient table.
    4. Check if any patient's status has changed (e.g., predicted_discharge_date is now *today*). If so, update the Bed status to 'pending_discharge'.

4. **Backend (Flask-SocketIO):** After the task runs, it will tell Flask to update *all* manager dashboards.
    - celery_task.py -> flask_socketio.emit('bed_dashboard_update', {dashboard_data}, broadcast=True)
5. **Frontend (React):** All open Bed Management Dashboards have a socket.on('bed_dashboard_update', ...) listener. They receive the new dashboard data and their React UI updates instantly, showing the bed flipping from "Occupied" to "Pending Discharge." This triggers the automated discharge workflow.

---

7. Key Challenges & Risks

- **HIPAA / Data Security:** This is your **#1 risk**. All patient data must be encrypted (at rest and in transit). Your Flask-JWT-Extended authentication must be perfect. All deployments must be on secure, HIPAA-compliant servers.
- **EHR Integration (The Big One):** The hospital already has an Electronic Health Record (EHR) system (like Epic, Cerner, etc.). Your app *must* integrate with it. You cannot have two separate patient databases. This is a massive project that involves using healthcare data standards like **HL7** or **FHIR**. Your Phase 1 *must* be researching the hospital's specific EHR and its API capabilities.
- **AI Model Accuracy & Bias:** Your models are only as good as your data. Getting clean, unbiased historical data from a hospital is extremely difficult. You must spend significant time cleaning data and ensuring your models are not biased (e.g., not predicting longer stays for one ethnicity vs. another).