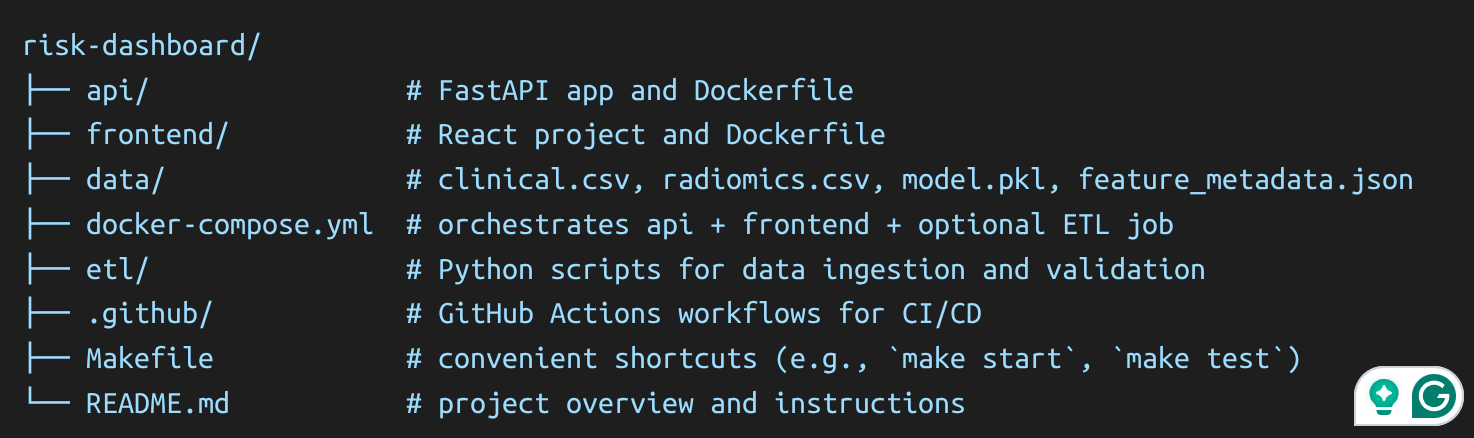
**Implementation Plan: Risk Prediction Dashboard**

*This narrative outlines each phase in detail, explaining what you’ll build, why it matters, and how to execute it step by step.*

## **1. Project Foundation & Environment**

Before writing any code, we’ll lay a solid groundwork:

1. **Initialize Version Control**
   * Create a new GitHub repository named risk-dashboard.
   * In the repo root, add a README.md summarizing the project, a .gitignore (ignoring node\_modules, \_\_pycache\_\_, etc.), and a license of your choice.
   * Decide on a branching strategy: e.g. main for production-ready code and dev for ongoing work.
2. **Set Up Local Development**
   * Install Docker and Docker Compose on your Linux VM or workstation.
   * Add a docker-compose.yml that defines two services: api (FastAPI) and frontend (React).
   * Mount the data/ folder into both containers so you can share CSVs and model files easily.
3. **Define Directory Structure**Organize the repo so code is discoverable and maintainable:



## **2. Data Ingestion & Validation**

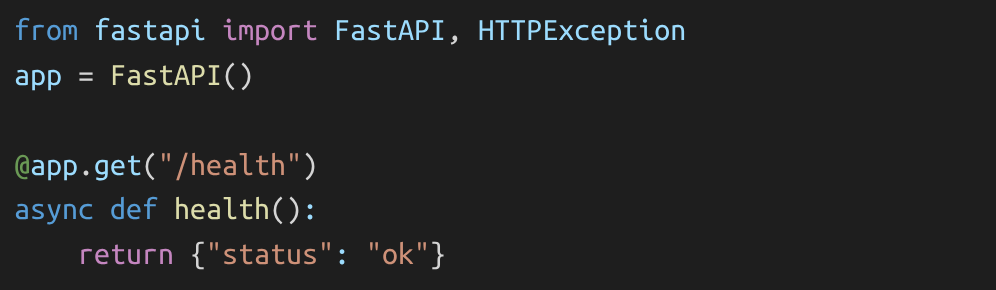
Reliable data flow is crucial. We’ll build a small ETL (Extract–Transform–Load) pipeline:

1. **Script: etl/ingest.py**
   * **Extract:** Use Pandas to load clinical.csv and radiomics.csv.
   * **Transform:** Validate columns against a JSON schema (e.g., schema/patient\_schema.json). If a required column is missing, log an error and halt.
   * **Merge:** Join both tables on patient\_id to form a unified view.
   * **Load:** Write the merged output to data/merged\_data.json for quick access.
2. **Scheduling the ETL**
   * In your docker-compose.yml, add a third service, etl-job, that runs etl/ingest.py every night via a simple shell wrapper and cron.
   * This ensures merged\_data.json stays up to date without manual intervention.
3. **Local Testing**
   * Run python etl/ingest.py interactively to verify no errors and inspect merged\_data.json.
   * Add a pytest suite under etl/tests/ that simulates missing columns and invalid data types.

## **3. API Development with FastAPI**

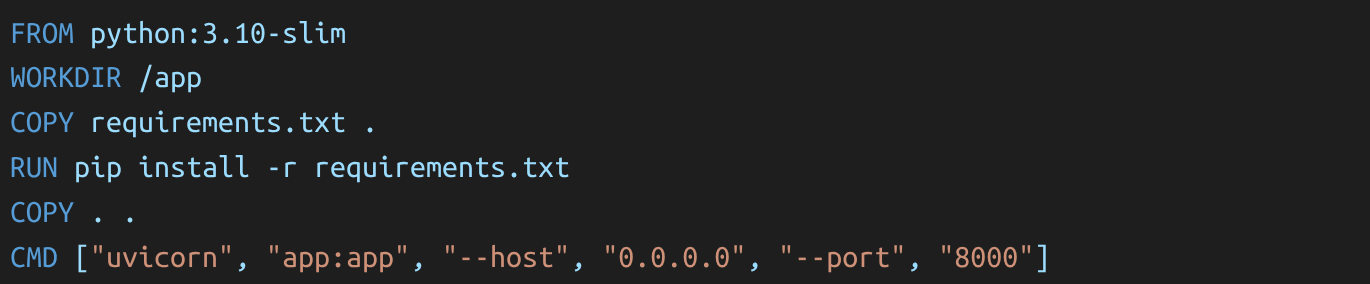
Next, we’ll create an HTTP service to serve patient data and run predictions:

1. **Scaffold the Application**
   * In api/app.py, start with:



* + Test via curl http://localhost:8000/health.

1. **Implement Data Endpoint**
   * **GET /patients/{id}**: Load merged\_data.json, find matching patient\_id, and return the record. If not found, raise HTTPException(status\_code=404, detail="Patient not found").
2. **Implement Prediction Endpoint**
   * At startup, deserialize the scikit-learn model: model = joblib.load("data/model.pkl").
   * **POST /predict/{id}**: Fetch the same merged record, extract the top-5 features (from feature\_metadata.json), run model.predict\_proba(...), and return a JSON with risk\_score and the feature values.
3. **Error Handling & Caching**
   * Return 400 for invalid IDs (e.g. non-numeric).
   * Use @lru\_cache(maxsize=128) to cache recent predict outputs, reducing latency.
4. **Containerization**
   * Write api/Dockerfile:



* + Build and verify: docker build -t risk-api api/.

1. **Testing**
   * Under api/tests/, write pytest cases for both endpoints (valid, invalid, error scenarios).

## **4. Frontend Development with React**

With the API ready, we’ll craft a responsive user interface:

1. **Initialize React**
   * From frontend/, run npx create-react-app . --template blank or use Vite for faster iteration.
   * Install core libraries: npm install axios react-router-dom react-table chart.js.
2. **Configure Routing**
   * In src/App.js, set up BrowserRouter with two routes:
     + /: renders PatientLookup.
     + /dashboard/:id: renders DashboardView.
3. **Patient Lookup Component**
   * A simple form with a controlled input for patientId. On submit, navigate to /dashboard/${patientId}.
4. **Dashboard View Component**
   * **Data Fetch:** On mount, call GET /patients/${id} and POST /predict/${id} via Axios.
   * **Clinical Table:** Feed the response’s clinical fields into a react-table instance.
   * **Radiomics Chart:** Use Chart.js to render a horizontal bar chart of feature names vs. values.
   * **Risk Score Display:** Show the numeric probability in a large font, colored based on risk thresholds.
5. **Error & Loading States**
   * Display a spinner while requests are pending.
   * If either call fails, show a user-friendly message (e.g., “Unable to load patient data. Try again later.”).
6. **Styling & Responsiveness**
   * Use Tailwind CSS or plain CSS modules for a clean, mobile-friendly layout.

## **5. Integration & Local Orchestration**

1. **Docker Compose**
   * Ensure docker-compose.yml brings up api, frontend, and etl-job together.
   * Map ports: 8000 → API, 3000 → React, Cron logs visible via docker-compose logs etl-job.
2. **Environment Variables**
   * Use .env files for API base URL in React and for model path in the API.
3. **End-to-End Verification**
   * In your browser, navigate to http://localhost:3000, enter a test ID, and confirm the full flow works.

## **6. Production Deployment**

1. **Static Build & NGINX**
   * Run npm run build in frontend, then configure NGINX to serve build/ on 80 and proxy /api to http://localhost:8000.
2. **Docker Compose for Production**
   * Create docker-compose.prod.yml mirroring local, but with volume mounts for persistent logs and appropriate restart policies.
3. **Launch on VM**
   * Pull the repo, run docker-compose -f docker-compose.prod.yml up -d, and verify accessibility over HTTPS.

## **7. Monitoring & Ongoing Maintenance**

* **Logging:** Configure both API and NGINX to write timestamped logs to disk.
* **Health Check:** Set up a simple cron job or service manager check to hit /health and restart containers on failure.
* **Model Updates:** Document the process: drop a new model.pkl and restart the api container; update feature\_metadata.json and redeploy.
* **Documentation:** Keep the README.md current with architecture diagrams, swagger API docs (via FastAPI’s auto-generated docs), and contribution guidelines.

hf\_NJgBitqvmAQowpwkKtjEDyssebUGHKTbdw