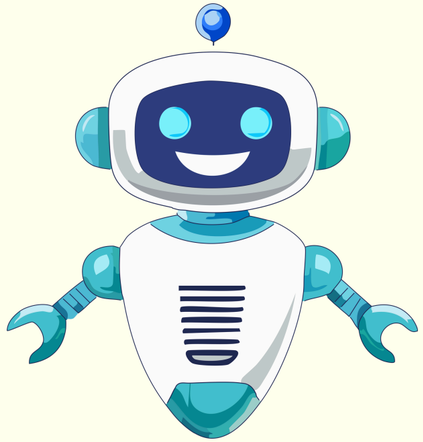
CHRONIC DISEASE PREDICTION AND MANAGEMENT SYSTEM

PROJECT RECORD



NAME: Saurashmi Bhattacharyya

Registration ID: 25BOE10062

SUBMISSION DATE: 25th November, 2025

INTRODUCTION

This project aims to leverage computational methods, specifically Machine Learning (ML), to address the critical public health issue of chronic disease management. We developed a web-based system designed for early risk prediction and personalized patient support, focusing on a specific condition, Type 2 Diabetes. The system provides tools for both patients and healthcare providers to monitor, predict, and manage disease progression, aligning with modern preventive healthcare strategies.

PROBLEM STATEMENT

The increasing global burden of chronic diseases requires proactive and scalable intervention tools. The core problem is the lack of accessible, personalized risk assessment tools that can identify high-risk individuals before symptoms manifest and provide actionable, timely advice. Our system addresses this by integrating predictive modeling with a user-friendly management interface.

FUNCTIONAL REQUIREMENTS

- Risk Assessment Module: Accepts health data (biometrics, labs) and uses the ML model to output a risk probability score (e.g., 75%).
- Recommendation Module: Generates personalized advice (diet, exercise) based on the risk score and patient's data.
- Provider/Data Management Module: Allows doctors to view, monitor, and manage patient records (CRUD operations).

NON FUNCTIONAL REQUIREMENTS

While not part of the core task, these factors define the system's quality. You need to focus on at least four:

- Security: Ensure secure handling of sensitive patient data, implementing secure logins and considering data encryption.
- Performance: The ML prediction must be fast, ideally returning the risk score within a defined timeframe (e.g., 5 seconds).
- Usability: The interface must be straightforward and intuitive for patients who input data.
- Scalability: The system structure should be robust enough to handle a growing load of patient records over time.

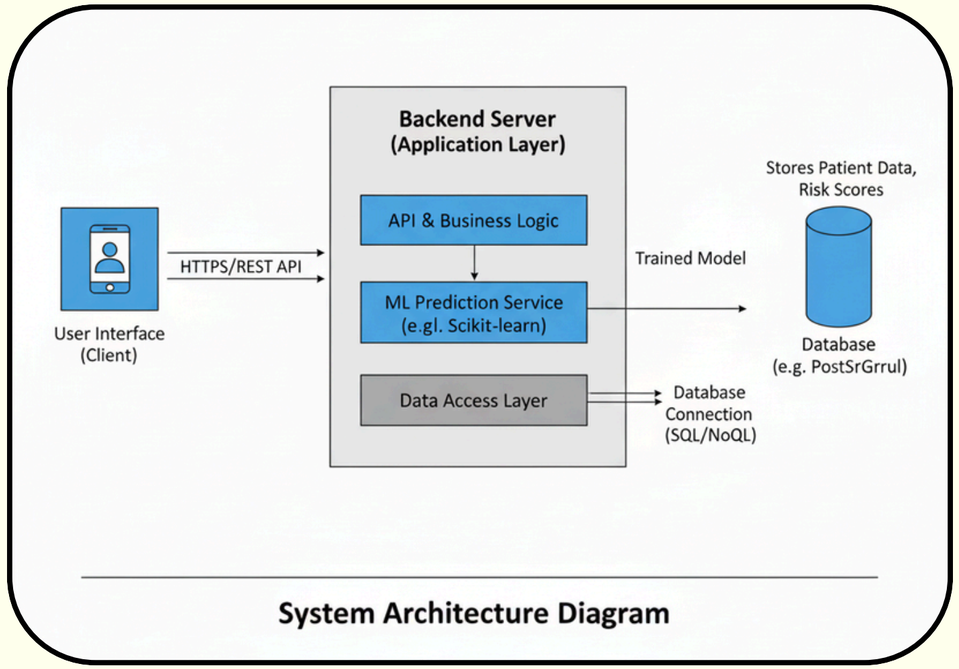
SYSTEM ARCHITECTURE

The system employs a Three-Tier Architecture:

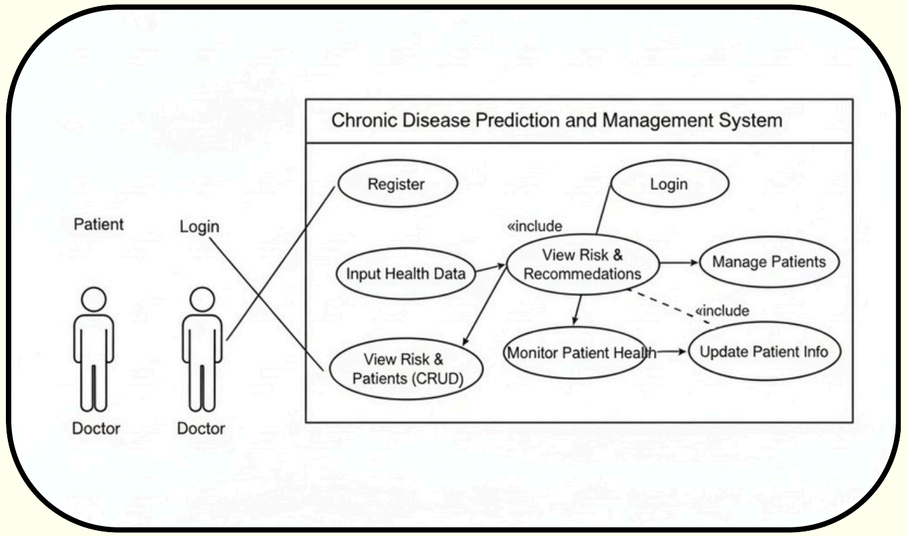
1. Presentation Tier: The web interface (HTML/CSS/JS) accessed via a browser/mobile device.
2. Application Tier (Backend Server): Implemented using Python/Flask. It contains the business logic, REST APIs, and the ML Prediction Service.
3. Data Tier (Database): A relational database (e.g., PostgreSQL/SQLite) used for persistent storage of patient records, health data, and risk history.

DESIGN DIAGRAMS

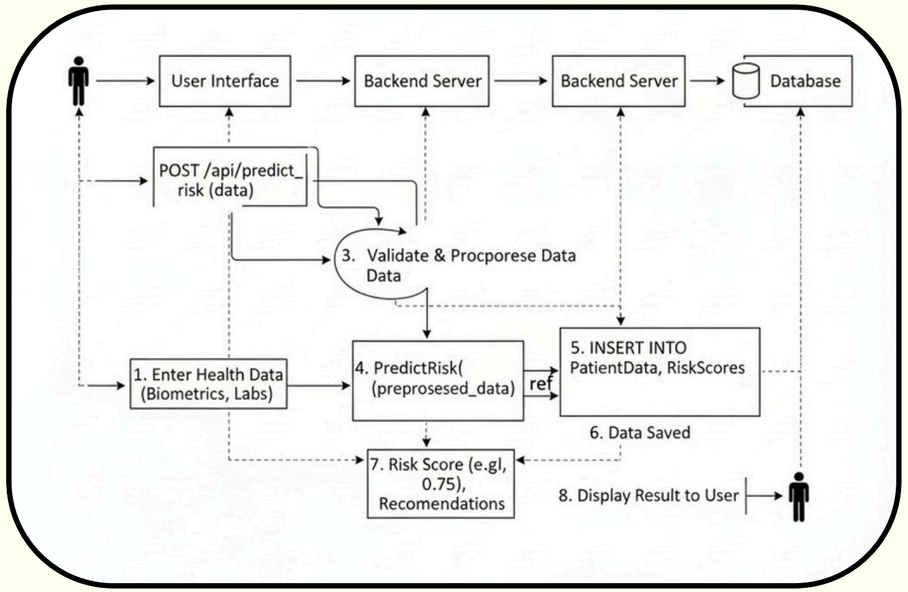
SYSTEM ARCHITECTURE DIAGRAM:



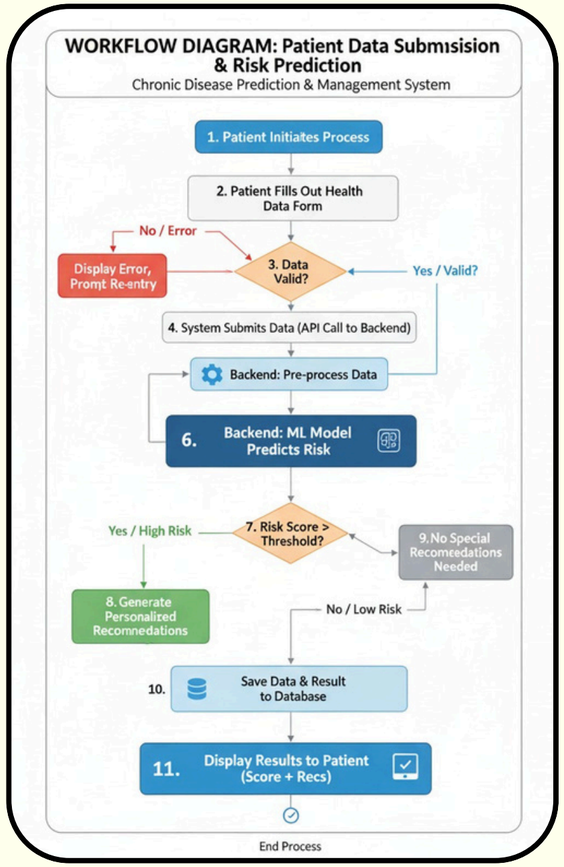
USE CASE DIAGRAM:



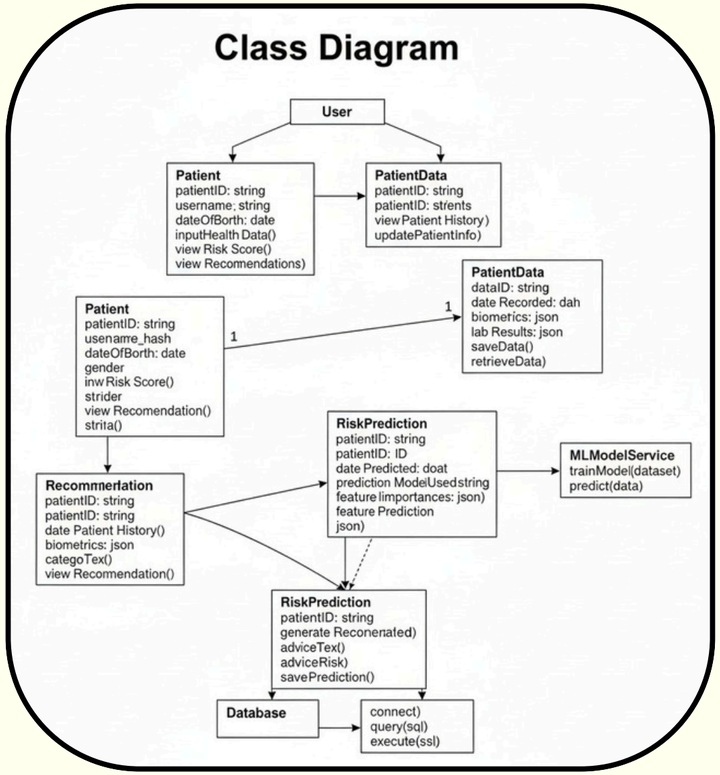
SEQUENCE DIAGRAM:



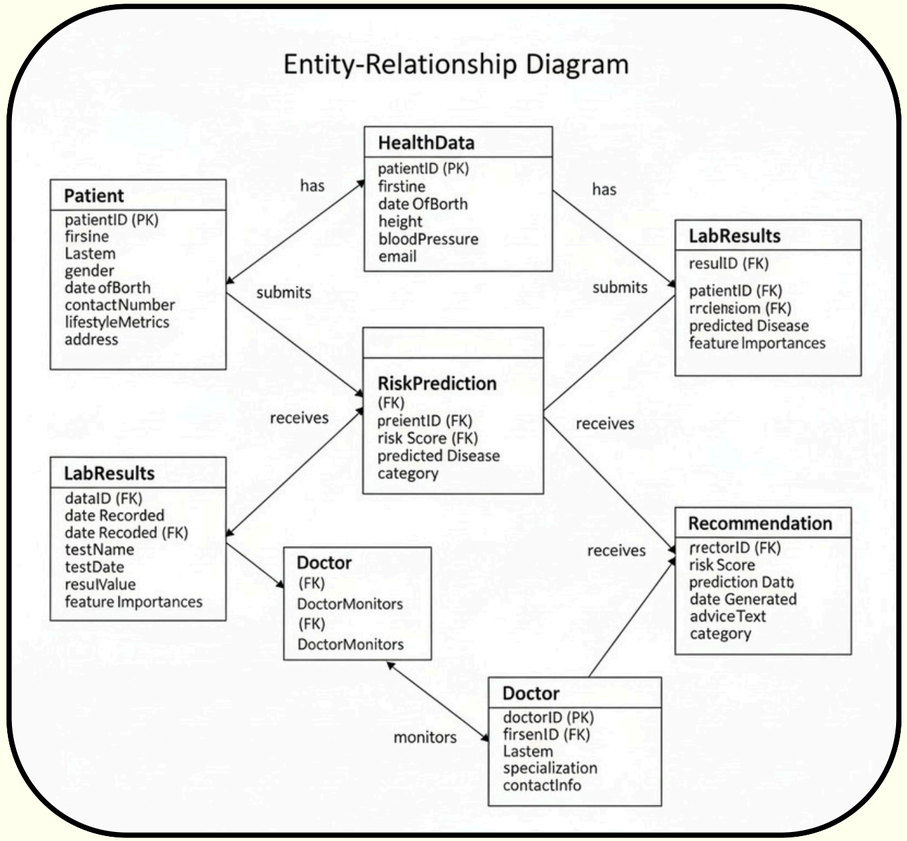
WORKFLOW DIAGRAM



CLASS DIAGRAM:



ER DIAGRAM:


Entity-Relationship Diagram
HealthData
patientID (PK)
firstine
date OfBorth
height
bloodPressure
email
Patient
patientID (PK)
firsine
Lastem
gender
date ofBorth
contactNumber
lifestyleMetrics
address
has
has
submits
submits
LabResults
resullD (FK)
patientID (FK)
predicted Disease
feature Importances
RiskPrediction
(FK)
preientID (FK)
risk Score (FK)
predicted Disease
category
receives
receives
LabResults
dataID (FK)
date Recorded
date Recoded (FK)
testName
testDate
resulValue
feature Importances
Doctor
(FK)
DoctorMonitors
(FK)
DoctorMonitors
receives
Recommendation
rrectorID (FK)
risk Score
prediction Datı
date Generated
adviceText
category
monitors
Doctor
doctorID (PK)
firsenID (FK)
Lastem
specialization
contactInfo

DESIGN DESICIONS AND RATIONALE :

COMPONENT	TECHNOLOGY	RATIONALE
Backend framework	Python	Chosen for its lightweight, rapid development capabilities, and strong integration with popular ML libraries (Scikit-learn, Pandas).
ML Algorithm	Random Forest Classifier	Rationale: Chosen because it handles non-linear relationships, is less prone to overfitting than complex neural networks for small/medium datasets, and provides feature importance, which is crucial for medical rationale.
Database	SQLite (for prototype) / PostgreSQL (for production)	SQLite was used for development simplicity. PostgreSQL is recommended for production due to its robustness, ACID compliance, and secure data handling capabilities necessary for healthcare.

COMPONENT	TECHNOLOGY	RATIONALE
System Architecture	RESTful API	Provides clean separation between the frontend (UI) and the backend (ML logic), facilitating future mobile app development.

IMPLEMENTATION DETAILS:

The implementation was divided into four main components:

- Data Preprocessing Pipeline: Handled data cleaning, missing value imputation, and one-hot encoding of categorical variables required for the ML model input.
- ML Service (MLModelService): Loaded the pre-trained Random Forest model and exposed a single predict(data) method via an internal API call from the Flask application.
- Backend API: Flask routes were set up to handle user authentication, data submission (/api/predict_risk), and CRUD operations for the doctor's dashboard.
- Frontend Interface: Simple HTML/Bootstrap pages implemented for patient data input and visualization of the risk score (e.g., low, medium, high) and recommendations.

```python
# services/ml_service.py - Handles all machine learning operations

class MLModelService:
    def __init__(self, model_path):
        """Loads the pre-trained model into memory."""
        # Use simple file handling for loading the model object
        try:
            with open(model_path, 'rb') as file:
                self.model = pickle.load(file)
            print(f"INFO: Model loaded successfully from {model_path}")
        except FileNotFoundError:
            self.model = None
            raise Exception(f"ML Model file not found at {model_path}. Run training script first!")

    def predict_risk(self, feature_vector):
        """Calculates the probability of chronic disease."""
        if not self.model:
            raise Exception("Model is not initialized.")

        # Convert the feature vector (array/list) into a DataFrame
        features_df = pd.DataFrame([feature_vector], columns=Config.MODEL_FEATURES)
```

```python
# data/db_connector.py - Database connector using SQLite

class DBConnector:
    def __init__(self, db_uri):
        self.db_uri = db_uri

    def _execute(self, query, params=(), fetch=False):
        # Implementation using sqlite3 to connect, execute, commit, and close
        conn = sqlite3.connect(self.db_uri)
        cursor = conn.cursor()
        cursor.execute(query, params)
        conn.commit()
        return cursor.fetchall() if fetch else None

    def init_db(self):
        """Creates the necessary tables (Patient and HealthData)."""
        self._execute("CREATE TABLE IF NOT EXISTS Patient (patient_id TEXT PRIMARY KEY, nar
        self._execute("CREATE TABLE IF NOT EXISTS HealthData (data_id INTEGER PRIMARY KE

    def save_patient_data(self, patient_id, raw_data, risk_score):
        """Saves a new submission and its prediction score."""
        date_recorded = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

```python
# config.py - Central Configuration Settings (No change from previous version)

class Config:
    # Database settings
    DATABASE_URI = 'sqlite:///health_data.db'

    # ML settings
    ML_MODEL_PATH = 'models/diabetes_rf_model.pkl'

    # Business logic constants
    HIGH_RISK_THRESHOLD = 0.60
    # List of features expected by the model in order
    MODEL_FEATURES = ['pregnancies', 'glucose', 'bp', 'bmi', 'age']
```

```python
# utils/data_preprocessing.py - Cleans and transforms raw data

def preprocess_data(raw_data):
    """
    Cleans, validates, and transforms raw form data into a feature vector
    expected by the ML model.
    """
    feature_list = []

    # Check for required fields and convert types
    for key in Config.MODEL_FEATURES:
        try:
            value = float(raw_data.get(key, 0))

            # Simple placeholder imputation (e.g., replace 0 with a default)
            if key in ['glucose', 'bp', 'bmi'] and value == 0:
                value = 75.0

            feature_list.append(value)
        except ValueError:
            raise ValueError(f"Invalid numeric input for feature: {key}. Please enter a number.")
```

```python
# services/recommendation_engine.py - Generates advice based on prediction

def generate_advice(score, patient_data):
    """
    Generates personalized advice based on the risk score and patient's data.
    """
    recs = []

    if score >= Config.HIGH_RISK_THRESHOLD:
        recs.append("🟢 URGENT: Your risk is high. Please consult a health professional immediately.")

        if float(patient_data.get('bmi', 0)) > 30:
            recs.append("💡 Action: Focus on reducing BMI through consistent exercise.")

        if float(patient_data.get('glucose', 0)) > 140:
            recs.append("🟢 Action: Strictly limit sugary drinks and monitor carbohydrate intake.")

    elif score >= 0.35: # Medium Risk
        recs.append("🟡 Risk is medium. Focus on prevention: maintain a balanced diet and regular activ

    else:
        recs.append("🟢 Risk is low. Continue healthy lifestyle habits. Repeat checkup annually.")
```

```python
# utils/data_preprocessing.py - Cleans and transforms raw data

def preprocess_data(raw_data):
    """
    Cleans, validates, and transforms raw form data into a feature vector
    expected by the ML model.
    """
    feature_list = []

    # Check for required fields and convert types
    for key in Config.MODEL_FEATURES:
        try:
            value = float(raw_data.get(key, 0))

            # Simple placeholder imputation (e.g., replace 0 with a default)
            if key in ['glucose', 'bp', 'bmi'] and value == 0:
                value = 75.0

            feature_list.append(value)
        except ValueError:
            raise ValueError(f"Invalid numeric input for feature: {key}. Please enter a number.")
```

TESTING APPROACH

A three-pronged testing strategy was employed:

- Unit Testing: Used the unittest framework in Python to test individual functions (e.g., data_cleaning(), generate_recommendation()).
- Model Validation: Performed cross-validation (k=10) during training and used a dedicated 20% hold-out test set to evaluate performance metrics (Accuracy, F1-score).
- System/Integration Testing: Tested the full prediction workflow (from patient input to result display) to ensure all components, including the API and database connection, worked seamlessly.

CHALLENGES FACED

- Data Imbalance: The public dataset used had a disproportionately low number of positive (diabetic) cases, which required careful handling (e.g., using SMOTE or adjusting class weights) to prevent the model from becoming biased towards predicting negative outcomes.
- Deployment Complexity: Integrating the Python-based ML Service with the web server required configuring environment variables and ensuring the model file was securely loaded and accessible.

LEARNINGS AND KEY TAKEAWAYS

- Domain Specificity: Learned that even a high-accuracy model requires interpretability (e.g., feature importance output) in a healthcare context to be trustworthy.
- Modular Design Importance: The separation of the ML prediction logic into a dedicated service (Component Diagram) proved essential for scalability and maintenance.
- Version Control Discipline (Git): Learned to use branches effectively for developing the backend and frontend simultaneously, preventing integration conflicts.

FUTURE ENHANCEMENTS

- Integration with Wearables: Allow patients to sync data directly from devices like smartwatches (e.g., heart rate, step count).
- Multi-Disease Prediction: Extend the system to predict the risk for other chronic conditions (e.g., hypertension, heart disease) using separate, specialized models.
- Teleconsultation Feature: Integrate a secure video/chat functionality for doctor-patient communication directly within the platform.

REFERENCES

- Dataset Source, e.g., Pima Indians Diabetes Database from UCI Repository]
- [Key Library Documentation, e.g., Scikit-learn documentation]
- [Academic Papers/Articles Referenced]