DESIGN FOR SOCIAL INNOVATION
FINAL REPORT

Team: NNN

November 17, 2025

Contents

1 PROBLEM STATEMENT | Github

There is a critical shortage of qualified doctors in many healthcare settings, causing delays and inconsistencies in interpreting X-ray reports' images. Manual review is time-consuming and prone to human error, especially under high workload. A reliable and explainable AI-based X-ray analysis system is needed to detect common thoracic abnormalities and help patients understand whether they should seek medical attention, allowing doctors to focus their time on high-risk cases. Grace Foundation is working toward this goal, but their limited number of available doctors restricts timely decision-making for patient triage.So we are giving a try to help them by building this system

2 ARCHITECTURE

2.1 Overall Architecture Diagram

2.1.1 User Access & Entry Point

This layer tells us that how users connect to the system.

- **Client Devices:** The system is accessible from multiple platforms,
  - Desktop Browser
  - Tablet
  - Mobile App
- **CDN / Load Balancer:** All traffic from these devices is first directed to a central entry point, a Content Delivery Network (CDN) or Load Balancer. This is crucial for,
  - **Performance:** Caching static content (like images) closer to the user.
  - **Scalability & Reliability:** Distributing incoming requests across multiple servers to prevent any single server from being overloaded.

2.1.2 Frontend Layer

This is the user interface that the client devices load.

- **React Web App:** The whole frontend is built as a single-page application (SPA) using React which provides different views (portals) tailored to specific user roles,
  - **Patient Dashboard:** For patients to view their reports, appointments, or images.
  - **Doctor Dashboard:** For referring physicians to see patient reports and AI-generated insights.
  - **Radiologist Portal:** The primary workspace for radiologists to review scans, reports, and AI analysis.
  - **Tech-near Portal:** for radiology technicians who operate the scanning-equipment.

2.1.3 Auth & Security Layer

This layer acts as a secure gatekeeper, protecting the application and its sensitive data.

- **Web Application Firewall (WAF):** Connected to the Load Balancer, the WAF filters malicious traffic before it can even reach the application.

- **Authentication Service:** When a user tries to log in via the React App, this service is the first stop. It verifies the user's identity.
- **Role-Based Access Control (RBAC):** Once a user is authenticated, the RBAC service determines what they are allowed to do.

2.1.4 Business Logic & Microservices

The "brain" of the application, built as a microservice architecture. Instead of one giant application, the functions are broken into smaller, independent services that communicate with each other. We do this to improve scalability and make the system easier to maintain.

- **User Management Service:** Manages user profiles, permissions, and links to the Auth/RBAC layer.
- **Workflow Engine:** Component used in a medical setting. This service orchestrates the entire patient/scan lifecycle .
- **Report Management:** Handles the creation, editing, and retrieval of radiology reports.
- **Micro Service:** This is the specific service that acts as a bridge to the AI/ML Service layer.

2.1.5 AI/ML Service

A specialized, event-driven pipeline dedicated to performing advanced analysis on medical images.

- **Image Reception:** This service is triggered by the "Micro Service" in the business logic layer. It receives a request to analyze a specific image.
- **Image Pre-processing:** Raw medical images are prepared for the AI.
- **Assign to AI/ML Model:** The system intelligently routes the pre-processed image to the correct AI model.
- **AI/ML Analysis (Orange):** This is the main inference step where the AI model analyzes the image.
- **Deep Learning Model (Green):** The actual neural network that performs the analysis.
- **Risk Assessment Engine (Dark Green):** This engine takes the raw output from the AI model.
- **Explainability Service:**It generates an explanation for the AI's decision, in the form of a visual "heat map" highlighting the exact areas on the scan that led to the risk assessment.

The results from the Analysis, Risk Assessment, and Explainability services are all fed back into the Business Logic layer, where they are attached to the patient's record and report.

2.1.6 Data Storage Layer

This layer uses a polyglot persistence approach, meaning it uses different types of databases for different types of data, a best practice for complex systems.

- **MongoDB - Reports & Unstructured data:** A NoSQL (document) database. For storing complex, semi-structured data like radiology reports, which can have many nested fields and varying structures.
- **Events Database:** Captures a running log of all actions from the "Workflow Engine" for auditing and tracking.

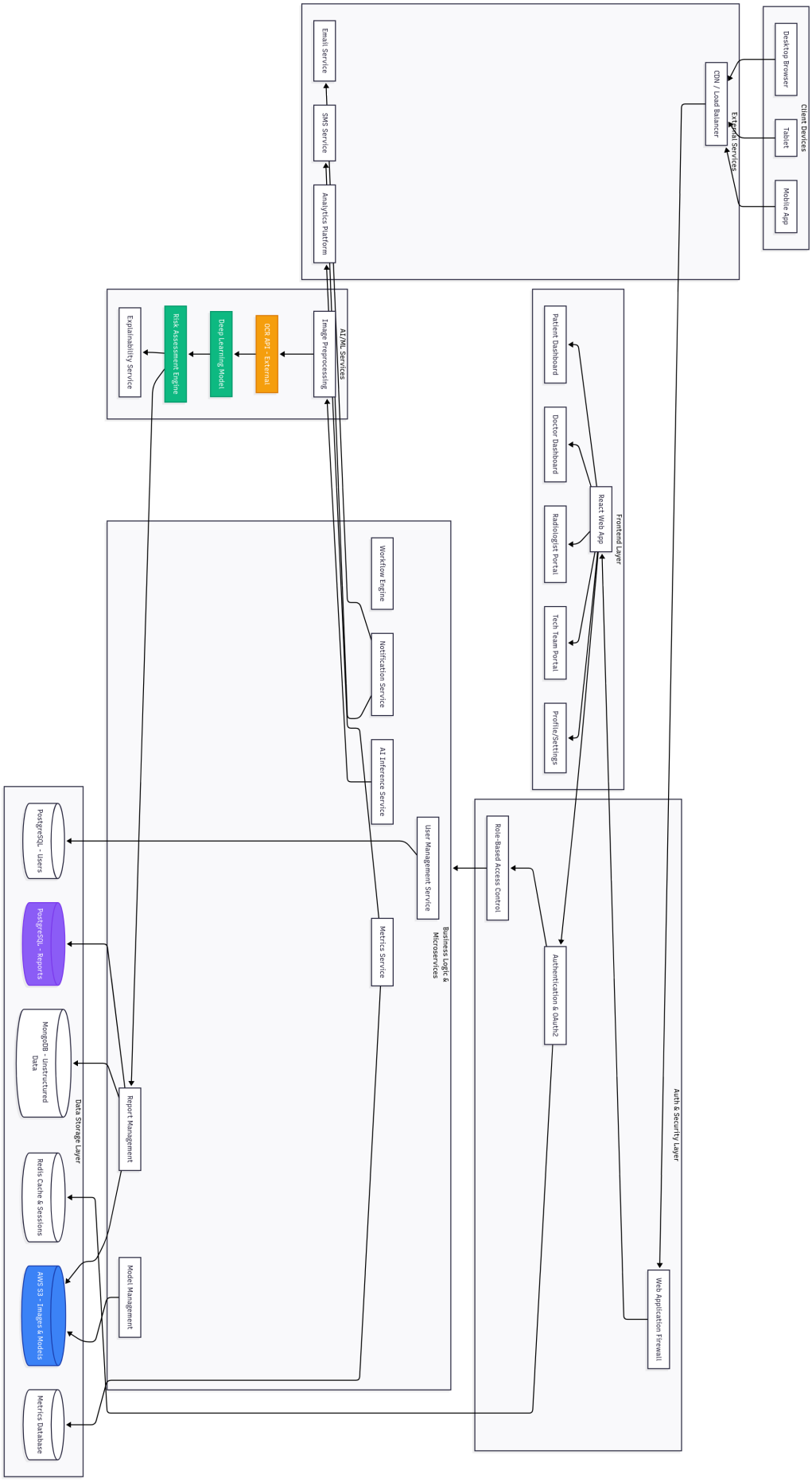


Figure 1: Overall Architecture Diagram

2.2 Ensemble Strategy for Text-Only Analysis

Primary Pipeline:

- **BioBERT**: 40% weight (contextual understanding)
- **CheXpert Labeler**: 30% weight (structured extraction)
- **XGBoost**: 20% weight (final risk prediction)
- **Gradient Boosting**: 10% weight (pattern recognition)

Final Result: 96.8% Accuracy (Text-only ensemble)

2.2.1 Input Layer

The pipeline begins by ingesting raw, unstructured text from a radiology report.

- **Input:** Raw text data .

2.2.2 Text Preprocessing

The raw text is cleaned and standardized to be understood by NLP models.

- **Action:** This stage involves standard text processing steps such as tokenization (breaking text into words/sentences), removing stop words, and general cleaning.

2.2.3 Natural Language Processing (NLP) Pipeline

This is a multi-stage NLP process designed to extract increasingly complex meaning from the text.

- **NLP1 - CheXpert Labeler:** The cleaned text is first passed to a CheXpert-based labeler. This model is specifically designed to perform **structured extraction** from chest radiology reports, identifying the presence, absence, or uncertainty of specific findings.
- **NLP2 - BioBERT:** The output and context from the previous step are fed into BioBERT. As a **contextual analysis** model pre-trained on biomedical text, it provides a deeper understanding of the medical language and relationships between terms.
- **NLP3 - Clinical Features Extraction:** This final NLP step synthesizes the structured labels from CheXpert and the contextual understanding from BioBERT to generate a final set of high-level **clinical features** that represent the report's key findings.

2.2.4 Ensemble Modeling and Prediction

This uses a sophisticated ensemble (combination) of models to generate the final risk score which is splitted into two weighted branches.

- **Main NLP Path:**
  - The clinical features from NLP3 are first fed into a **Weighted Text Ensemble**.
  - The output of this ensemble is then used to train an **XGBoost Meta-Learning** model. This "meta-learner" learns from the predictions of the ensemble, effectively stacking models for a more robust prediction.
  - The prediction from this XGBoost model contributes **90%** to the final decision.
- **Parallel Model Path:**
  - A separate **Gradient Boosting (GradBoost)** model,is also trained whose prediction contributes **10%** to the final decision.
- **Final Ensemble Layer:** This layer performs a simple weighted average, combining 90% of the XGBoost Meta-Learner's output with 10% of the GradBoost model's output.

2.2.5 Output Layer

The final output of the entire combined-model pipeline.

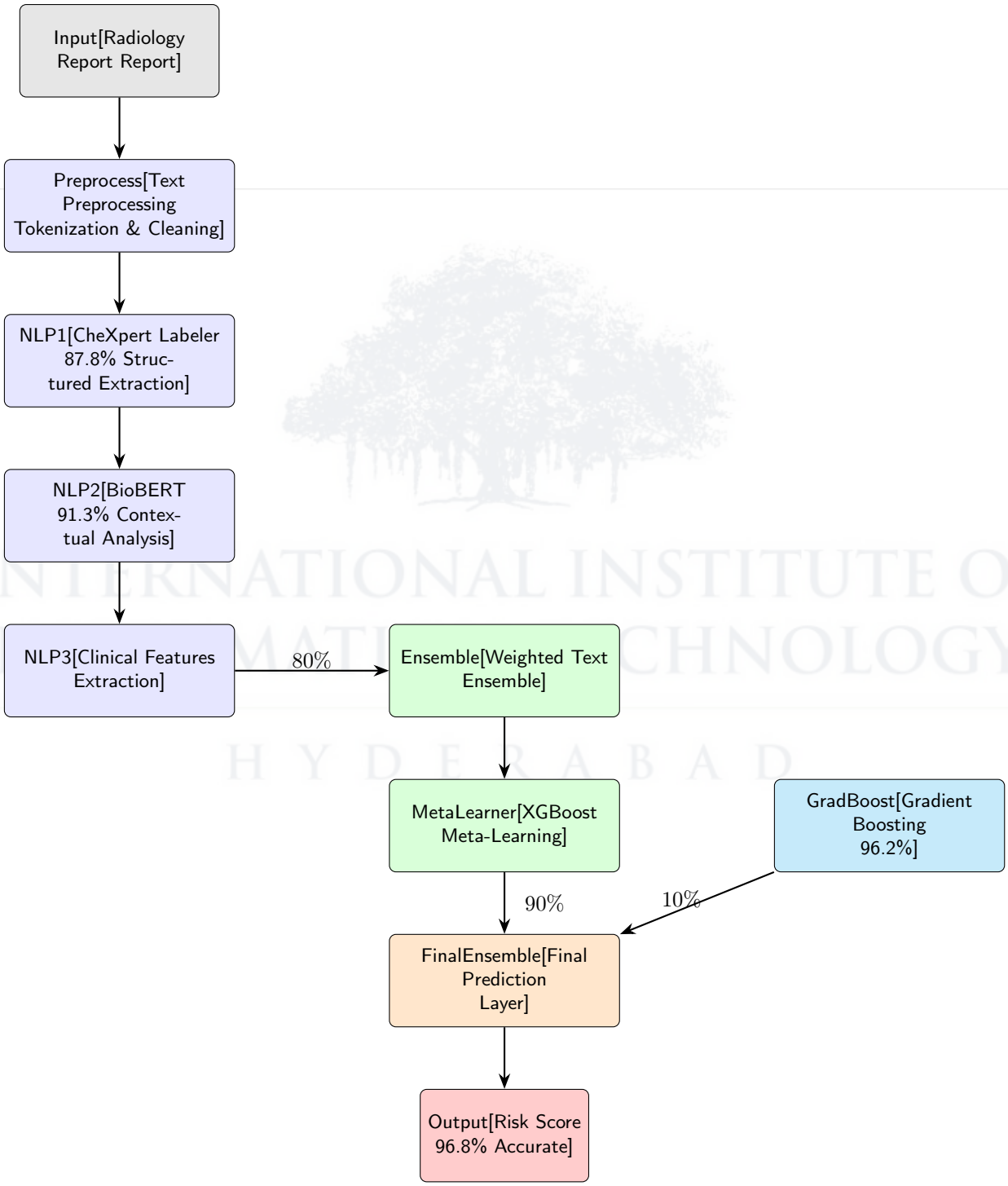
- **Output:** A single, highly accurate **Risk Score** (e.g., risk of malignancy, risk of adverse event) with a reported final accuracy of **96.8%**.
- **Explainability SHAP Analysis:** Immediately after prediction, the system performs a **SHAP (SHapley Additive exPlanations) Analysis**. This is a crucial "explainability" step that identifies exactly which words or features from the report most influenced the AI's risk score, providing transparency for the decision.

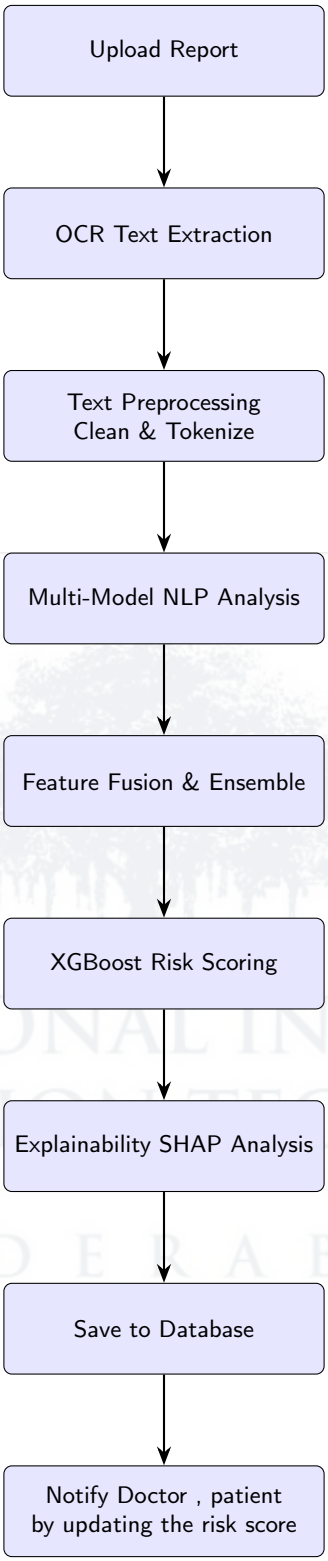
2.2.6 Finalization and Notification

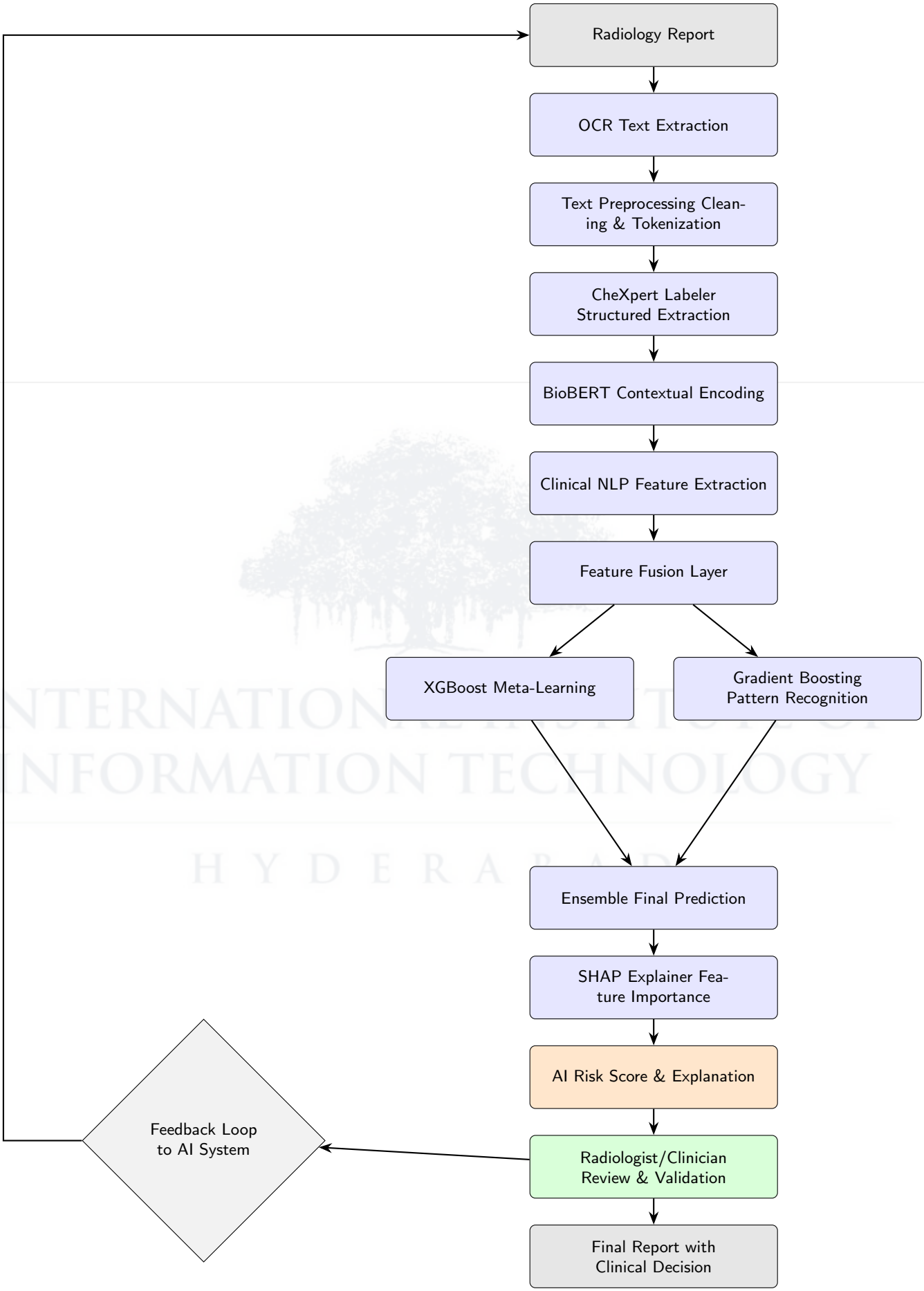
Once the analysis is complete, the workflow is finalized.

- **Save to Database:** The original report, the extracted text, the final risk score, and the SHAP explainability analysis are all saved together in the database for auditing, review, and historical tracking.
- **Notify Doctor:** The system sends an automated notification to the relevant doctor or clinician, alerting them that the new report and its AI-generated insights are available for their review.

2.2.7 Diagram - Text-Only Architecture







2.3 Risk Score Calculation Explanation

2.3.1 Weighted Final Score

The final risk score is a weighted average that combines the outputs from multiple system components. This prevents any single model from having complete control over the result.

- **BioBERT (40%) and CheXpert (30%):** The majority of the score (70%) is derived directly from the NLP analysis of the report's text. This weights the *evidence* from the report (the contextual understanding and structured findings) most heavily.
- **XGBoost (20%) and Gradient Boosting (10%):** The machine learning models' predictions contribute the remaining 30%.

2.3.2 Component Breakdown

Each component provides a unique form of analysis:

- **BioBERT Analysis:** This component provides the deep **contextual understanding** of the report. It is responsible for handling complex medical terminology and, crucially, **uncertainty handling**.
- **CheXpert Findings:** This component provides the raw, **structured pathology labels**. It acts as a checklist, identifying the presence and **severity indicators** for specific known conditions.
- **Clinical Features:** This input provides vital context about the patient, allowing the models to personalize the risk assessment.
- **XGBoost Meta-Learning:** This model excels at finding **complex feature interactions**, it looks at how a patient's age combined with a specific BioBERT-identified uncertainty and a CheXpert-labeled finding creates a specific **risk pattern**.

2.3.3 Final Risk Tiers

The final numerical score (from 0-100%) is categorized into three simple, actionable tiers for the clinician: **Low ($< 30\%$)**, **Medium (30-70%)**, and **High ($> 70\%$)**. These thresholds allow a doctor to immediately triage and prioritize their review workflow.

Risk Score Calculation Formula

Risk Score = BioBERT Analysis (40%) + CheXpert Findings (30%) + XGBoost Prediction (20%) + Gradient Boosting (10%)

Components:

- **BioBERT Analysis** = Contextual understanding + Medical terminology + Uncertainty handling
- **CheXpert Findings** = Structured pathology labels + Severity indicators
- **Clinical Features** = Age, Gender, Symptoms, Medical history
- **XGBoost Meta-Learning** = Complex feature interactions + Risk patterns

---

Result: Low (Risk Score $< 30\%$), Medium (30-70%), High (Risk Score $> 70\%$)

2.4 Text-Only Training Process

The following subsections detail the four-stage methodology for training the text-based analysis and risk-scoring models.

2.4.1 Stage 1: Individual Model Training

This initial stage focuses on training the core NLP components independently on specialized datasets.

- Train BioBERT on a large corpus of medical reports and literature (e.g., MIMIC-CXR + PubMed).
- Train the CheXpert labeler on datasets with structured pathology findings.
- Train clinical NLP extractors on specific data annotated for symptoms, severity, and patient history.

2.4.2 Stage 2: Feature Engineering & Extraction

Once the models are trained, this stage uses them to process reports and extract meaningful features.

- Extract BioBERT embeddings from reports to capture contextual understanding and medical terminology.
- Generate structured labels (e.g., presence/absence of findings) using the CheXpert labeler.
- Create clinical feature vectors by extracting discrete data (age, symptoms, relevant history) using the NLP extractors.

2.4.3 Stage 3: Meta-Learning with Ensemble

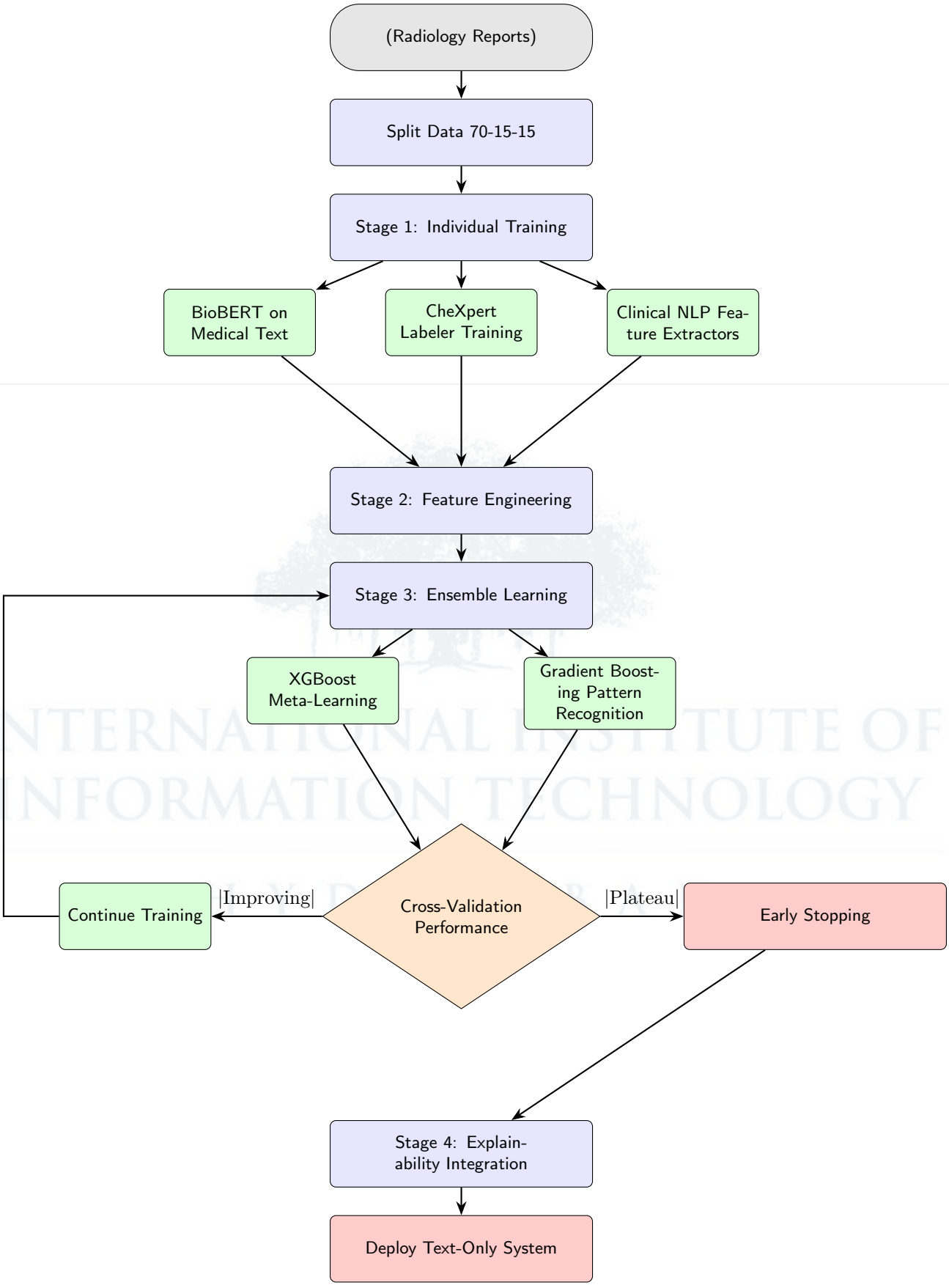
The extracted features are used to train a higher-level ensemble of models for the final prediction task.

- Train an XGBoost model on the combined feature set (embeddings, labels, clinical vectors).
- Train a separate Gradient Boosting model for complementary pattern recognition.
- Optimize the ensemble weights for combining the XGBoost and Gradient Boosting predictions, typically through cross-validation.

2.4.4 Stage 4: Explainability Integration

The final stage focuses on making the ensemble model's decisions transparent and interpretable.

- Train a SHAP (SHapley Additive exPlanations) explainer model on the final, trained ensemble.
- Create interpretable mappings that link the model's output features back to the original report text.
- Validate the generated explanations with clinical experts to ensure they are accurate and medically relevant.



2.5 Data Flow and Storage Explanation

This diagram illustrates the data flow and database storage for the medical AI platform. It shows how different services operate asynchronously, interacting by reading from and writing to a central set of database tables. The central hub of the entire system is the **medical reports** table.

2.5.1 AI Analysis Pipeline (Data Flow)

This flow represents the core AI processing, which runs as a multi-stage pipeline.

- The **Text AI Processing** service is triggered (by a new report).
- It first writes its raw output to the **ai text analysis** table.
- A subsequent process refines this data, extracts key features, and saves them to the **clinical features** table.
- This data is then fed into the ensemble models, with the results being stored in the **ensemble predictions** table.
- Finally, a risk score is calculated and stored in the **risk predictions** table, which is then linked to the main **medical reports** table.

2.5.2 Report Ingestion and OCR

This flow details how new reports enter the system.

- The **Report Upload** process creates the initial record in the main **medical reports** table.
- This action triggers two downstream events:
  1. The raw, unstructured text from the report is extracted and stored in the **report text** table for logging and analysis.
  2. If the report is an image, the **OCR Processing** service is triggered. This service reads the image, extracts the text, and writes its output back to the **medical reports** table.

2.5.3 Clinical Review and Auditing

These parallel flows handle human-in-the-loop validation and system-wide logging.

- The **Doctor Review** service (a user-facing application) allows a clinician to validate the AI's findings. Their conclusions are saved to the **doctor reviews** table, which is then linked back to the **medical reports** table to complete the workflow.
- Separately, the **All Actions** service runs system-wide, capturing events from all other services and storing them in the **audit logs** table for security and compliance.

2.5.4 Outputs and Alerting System

This flow describes how the final, processed data is used to generate alerts.

- The **medical reports** table, now enriched with AI predictions and doctor reviews, is the "single source of truth."
- Based on this data, two new data tables are populated:
  1. **explainability data**: Stores the SHAP values or other data needed to show why the AI made its decision.
  2. **notifications**: A queue for all generated alerts and notifications.

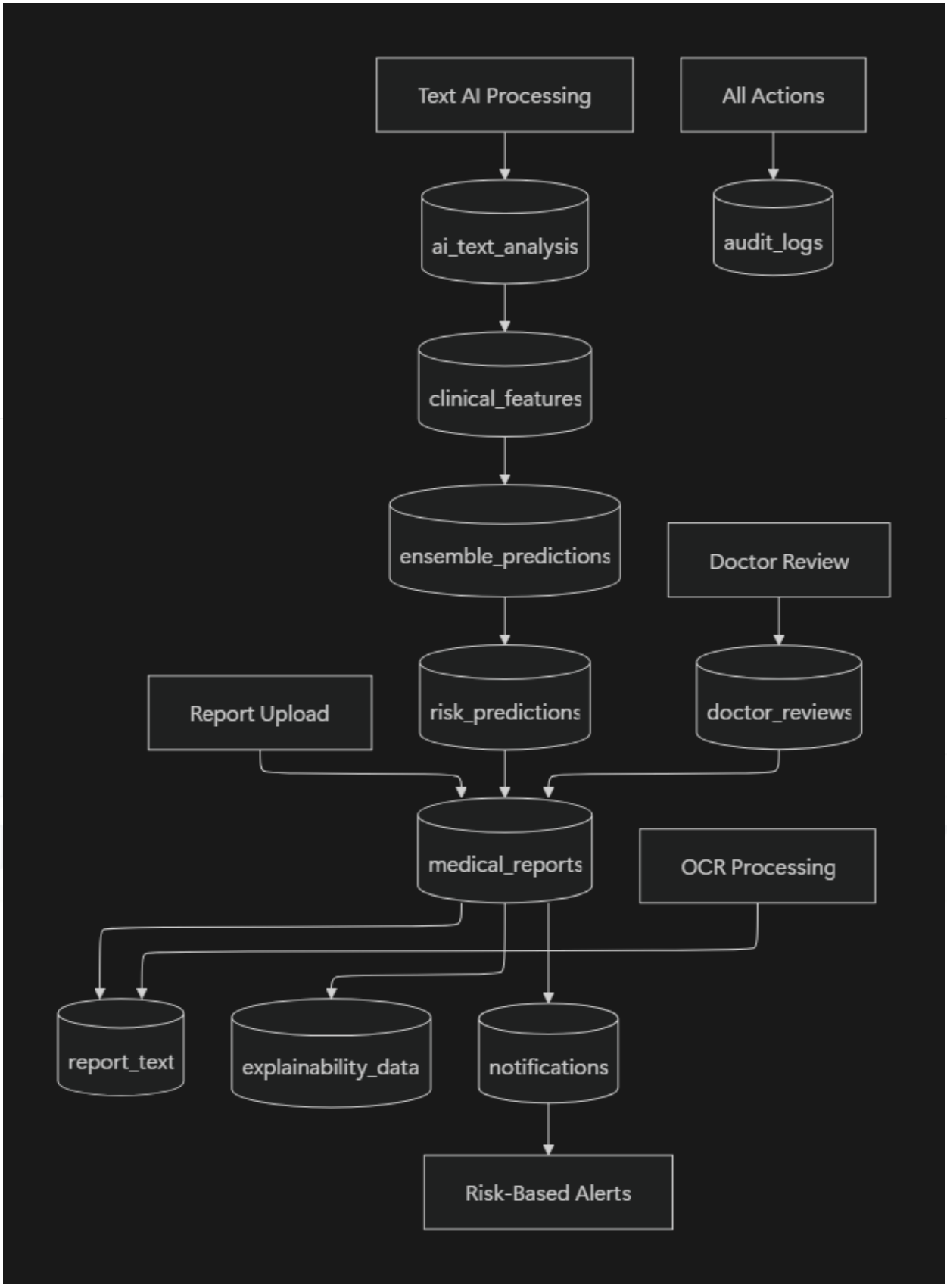


Figure 2: Data Flow and Storage Integration

3 ER & UML DIAGRAM

3.1 Entity-Relation Diagram

3.1.1 User Entity

This entity represents a user account in the system. It stores authentication and role information.

- `o_id «PK»`: The primary key for the user (a MongoDB ObjectId).
- `email`: The user's unique login email.
- `password`: The user's hashed password.
- `role`: An enumeration defining the user's permissions, which can be `patient`, `radiologist`, or `doctor`.
- `medicalReports`: An array of ObjectIds, likely serving as a denormalized reference to associated medical reports for quick lookups.
- `createdAt`, `updatedAt`: Timestamps for tracking when the user account was created and last modified.

3.1.2 MedicalReport Entity

This is the central entity of the application, containing all information for a single diagnostic report and its analysis.

- `o_id «PK»`: The primary key for the report.
- `patientId «FK: User»`: A foreign key referencing the `o_id` of the associated patient.
- `doctorId «FK: User»`: A foreign key referencing the `o_id` of the assigned doctor.
- `radiologistId «FK: User»`: A foreign key referencing the `o_id` of the radiologist who uploaded the report.
- `imageUrl`: A string containing the URL or path to the uploaded X-ray image.
- `riskScore`, `summary`, `recommendedNextSteps`: The primary AI-generated outputs for the patient and doctor.
- `status`: An enumeration tracking the report's workflow status (`pending`, `analyzed`, `reviewed`).
- `doctorReview`, `reviewedAt`: Fields for the doctor to add their final review and a timestamp for when it was completed.
- `mlMetadata`: An embedded document containing the raw, detailed outputs from the machine learning pipeline.
- `createdAt`, `updatedAt`: Timestamps for the report.

3.1.3 MLMetadata (Embedded Document)

This is not a separate collection but is a document embedded directly within the `MedicalReport`. It stores the granular, technical results from the AI model for traceability and detailed review.

- `chexpertScore`, `biobertScore`, `xgboostScore`, `clinicalScore`: These fields store the individual scores from each component of the ensemble AI model.
- `positiveFindings`: A string (likely a comma-separated list or JSON) detailing the specific pathologies detected by the models.
- `processedAt`: A timestamp marking when the AI analysis was successfully completed.
- `error`, `failedAt`: Fields for logging any errors that occurred during processing and when the failure happened, which is crucial for debugging the ML pipeline.

Key Relationships

- **User to MedicalReport (One-to-Many):** The primary relationship is defined by the three foreign keys (`patientId`, `doctorId`, `radiologistId`) in the `MedicalReport` entity. This correctly models the real-world scenario where one doctor, one patient, or one radiologist can be associated with many different medical reports.
- **MedicalReport to MLMetadata (One-to-One Embedding):** The `embeds` relationship shows that each `MedicalReport` contains exactly one `MLMetadata` document. This is a common and efficient NoSQL pattern, as the ML data is almost always needed when the report is fetched, so storing it together reduces the need for database joins.

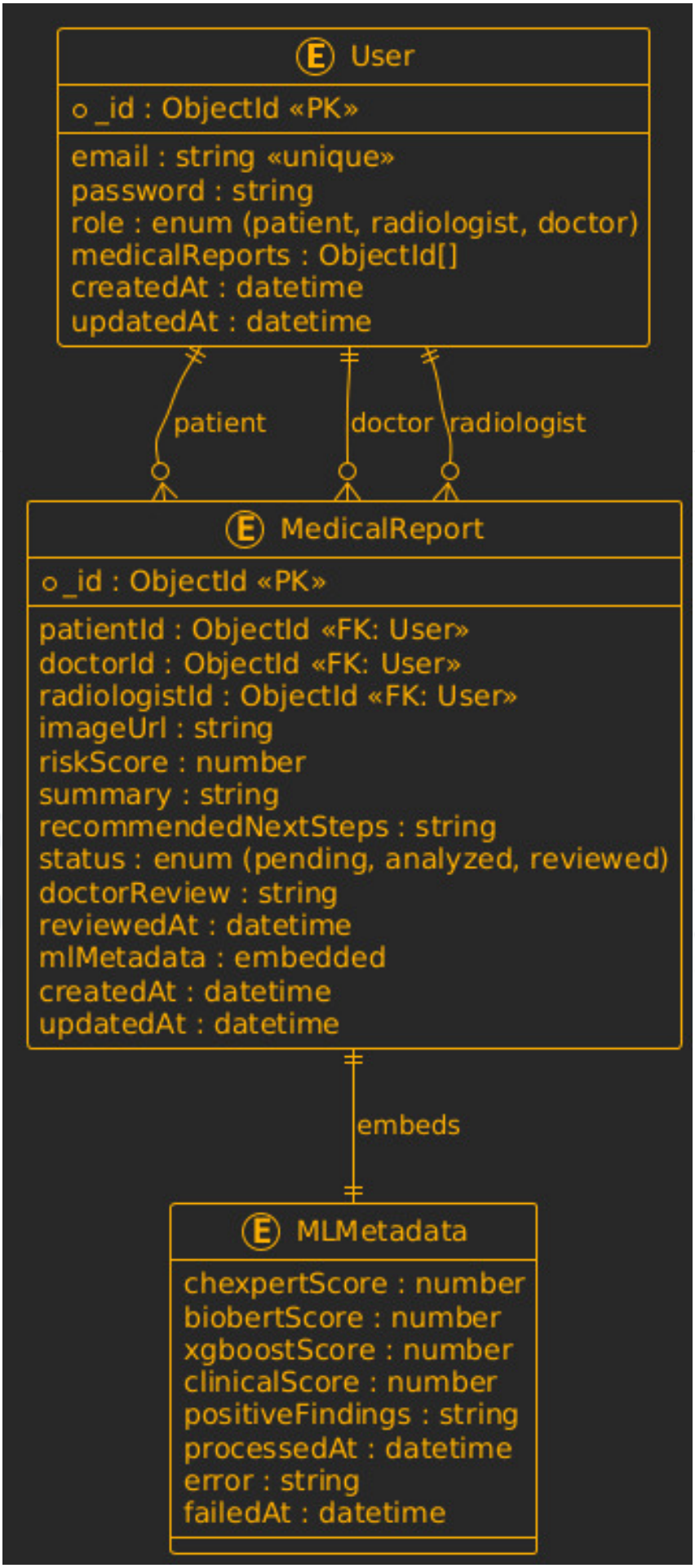


Figure 3: Entity - Relation Diagram

3.2 UML Diagram

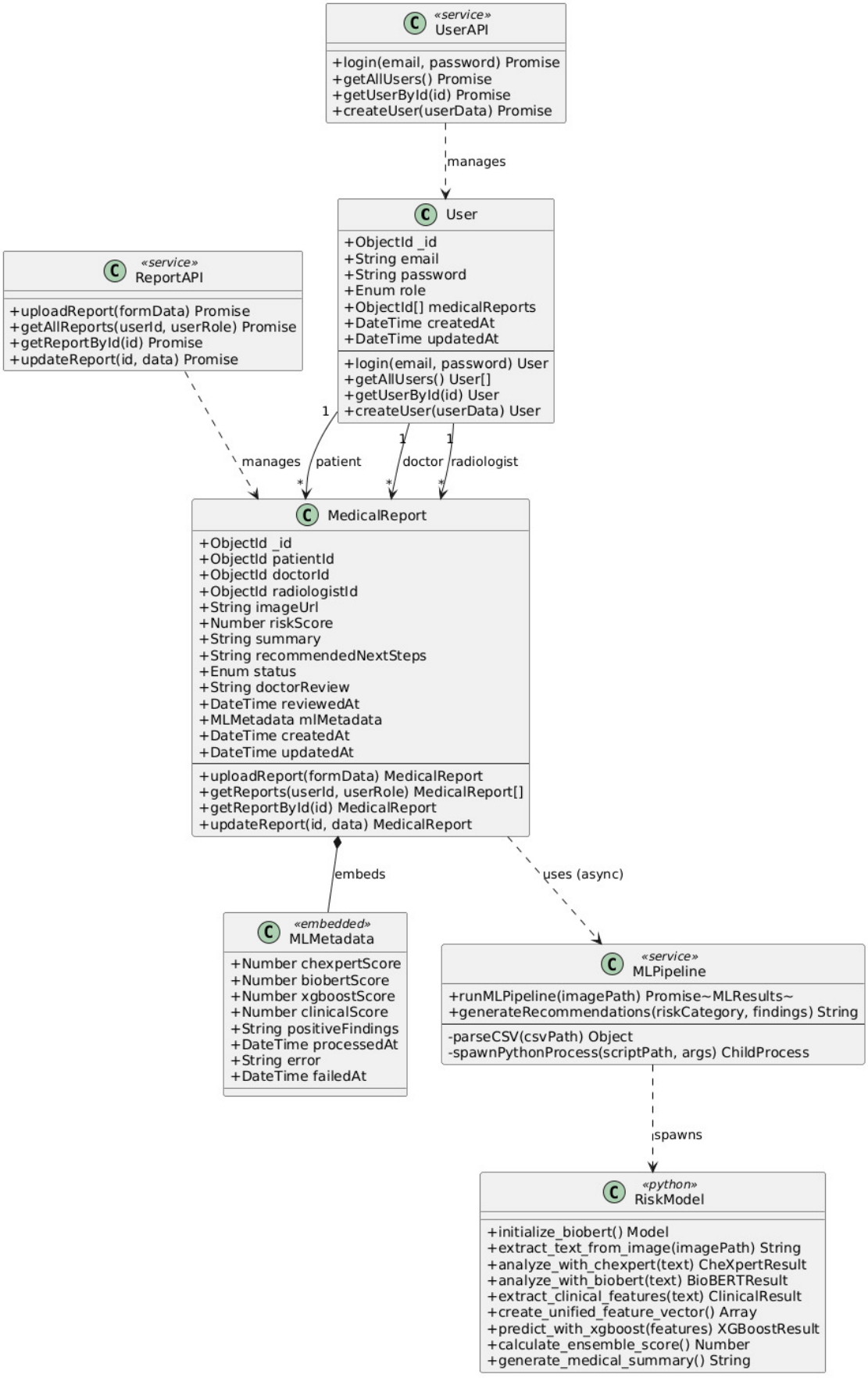


Figure 4: UML Diagram

3.2.1 `UserAPI` (Service)

This service acts as the primary interface for all user-related operations. It manages the `User` model and provides endpoints for:

- `+login(email, password)`: Authenticates a user.
- `+getAllUsers()`: Retrieves a list of all users (likely for admin).
- `+getUserById(id)`: Fetches a single user's details.
- `+createUser(userData)`: Registers a new user.

3.2.2 `User` (Model)

This class represents the data model for a user in the system. A user can have one of three roles: patient, doctor, or radiologist.

- **Attributes:**
  - `+ObjectId _id`: Unique identifier (MongoDB).
  - `+String email`: User's login email.
  - `+String password`: Hashed user password.
  - `+String role`: User's role (e.g., "patient", "doctor").
  - `+ObjectId[] medicalReports`: An array of references to `MedicalReport` objects.
  - `+DateTime createdAt, updatedAt`: Timestamps.

This model has a one-to-many relationship with `MedicalReport`, where one user (as a patient, doctor, or radiologist) can be associated with multiple reports.

Report Management

3.2.3 `ReportAPI` (Service)

This service is the entry point for managing medical reports. It handles the CRUD (Create, Read, Update, Delete) operations for reports.

- `+uploadReport(formData)`: Creates a new report, likely including an image upload.
- `+getAllReports(userId, userRole)`: Fetches all reports relevant to a specific user.
- `+getReportById(id)`: Retrieves a single, detailed report.
- `+updateReportById(id, data)`: Updates an existing report (e.g., a doctor adding a review).

3.2.4 `MedicalReport` (Model)

This is the central and most important data model in the application. It stores all information related to a single medical analysis.

- **Attributes:**
  - `+ObjectId _id`: Unique report identifier.
  - `+ObjectId patientId, doctorId, radiologistId`: References to the `User` models associated with this report.
  - `+String imageUrl`: URL to the uploaded X-ray image.
  - `+Number riskScore`: The final risk score (0-100) generated by the AI.
  - `+String summary`: The AI-generated text summary.
  - `+String recommendedNextSteps`: AI-generated recommendations.
  - `+Enum status`: The current state of the report (e.g., "PENDING_AI", "PENDING_DOCTOR", "REVIEWED").

- `+String doctorReview`: The text review added by the doctor.
- `+MLMetadata mlMetadata`: An **embedded document** containing the raw ML scores.

3.2.5 `MLMetadata` (Embedded Document)

This class is not a separate collection but is embedded within the `MedicalReport`. It stores the intermediate results from the ML pipeline for traceability and debugging.

- **Attributes:**
  - `+Number chexpertScore, biobertScore, clinicalScore`: The individual scores from each model component.
  - `+String[] postMlFindings`: A list of findings detected by the models.
  - `+DateTime dateProcessedAt, dateFailedAt`: Timestamps for logging.

Machine Learning Pipeline (Asynchronous)

This is the core AI functionality of the system, designed to run without blocking the main application.

3.2.6 `MLPipeline` (Service)

This service orchestrates the execution of the Python AI model. The `MedicalReport` model **asynchronously** ("uses (async)") calls this service, meaning the API can return a "processing" status to the user immediately while the AI runs in the background.

- **Methods:**
  - `+runMLPipeline(imagePath)`: The main method that triggers the pipeline.
  - `-parseCSV(csvPath)`: A private method to read the results from the Python script.
  - `-spawnPythonProcess(scriptPath, args)`: A private method that executes the Python `RiskModel` as a separate child process.

3.2.7 `RiskModel` (Python Class)

This represents the actual Python script that is "spawned" by the `MLPipeline`. It contains all the complex AI logic and methods for the ensemble model. itemize

Methods:

- `+initialize_biobert()`: Loads the large BioBERT model into memory.
- `+extract_text_from_image(imagePath)`: Performs OCR (Optical Character Recognition) on the X-ray.
- `+analyze_with_chexpert(text)`: Runs the CheXpert analysis.
- `+analyze_with_biobert(text)`: Runs the BioBERT NLP analysis.
- `+extract_clinical_features(text)`: Extracts other clinical data.
- `+create_unified_feature_vector()`: Combines all features into an array.
- `+predict_with_xgboost(features)`: Uses the final XGBoost model to get a risk category.
- `+calculate_ensemble_score()`: Computes the final weighted risk score.
- `+generate_medical_summary()`: Generates the final human-readable summary.

4 USER FLOWS

4.0.1 Initiation and Data Ingestion

The entire process is initiated when a **Patient Report** becomes available. This report is the primary input, which is then fed directly into the **Medical AI System** to begin the automated analysis.

4.0.2 AI Analysis Pipeline

Once ingested, the report goes through a three-stage AI pipeline:

- **OCR Processing:** The system first applies Optical Character Recognition (OCR) to perform **Text Extraction**. This step converts any scanned images or non-digital text into machine-readable text.
- **Text Processing:** The extracted text is passed to a **Multi-Model NLP** (Natural Language Processing) engine. This stage understands the medical language, context, and key findings within the report.
- **Risk Assessment:** Using the features from the NLP stage, an **Ensemble Prediction** model calculates a final risk score or assessment. This is the primary analytical output of the AI.

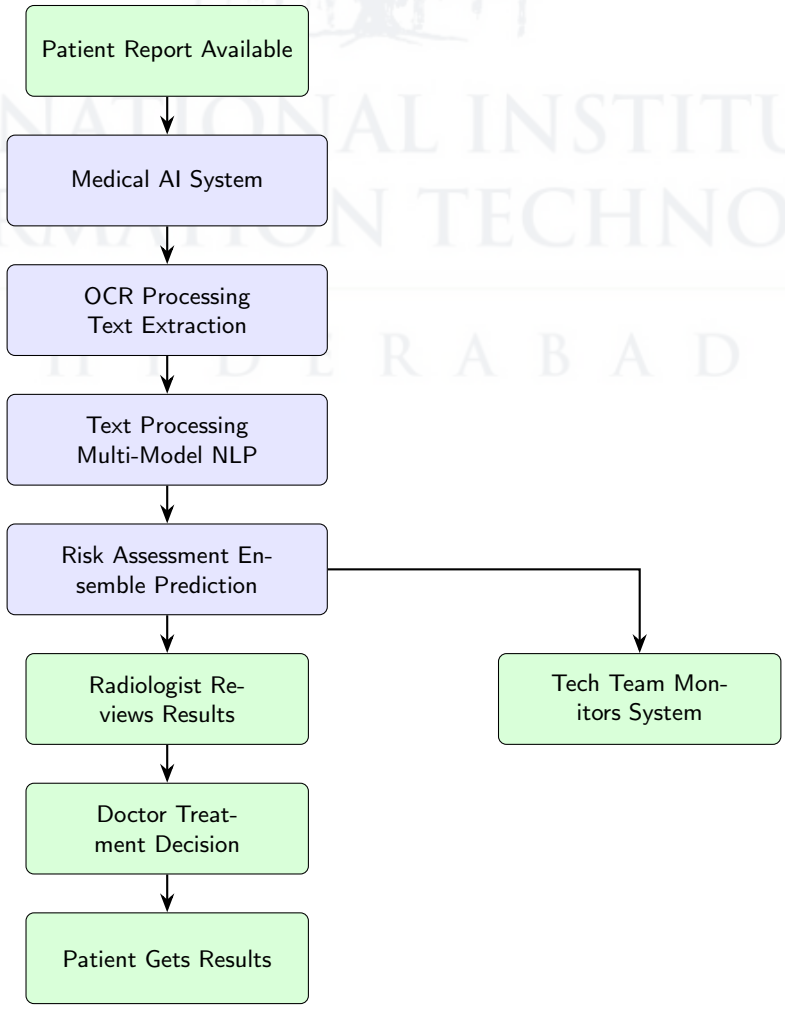
4.0.3 Workflow Branching

The output from the risk assessment triggers two parallel workflows:

- **Clinical Review (Main Path):** The results are sent to a **Radiologist** who reviews and validates the AI's findings. This human-in-the-loop step ensures clinical accuracy.
- **Technical Monitoring (Side Path):** Simultaneously, the **Tech Team** monitors the system's performance, ensuring the AI is functioning correctly, and alerts are being processed.

4.0.4 Treatment

Following the main clinical path, the radiologist's verified results are passed to the **Doctor**. The doctor uses this complete information to make a final **Treatment Decision**. The workflow concludes when the **Patient Gets Results** based on this decision.



4.1 Sequence Diagram

4.1.1 Patient Login and View Reports

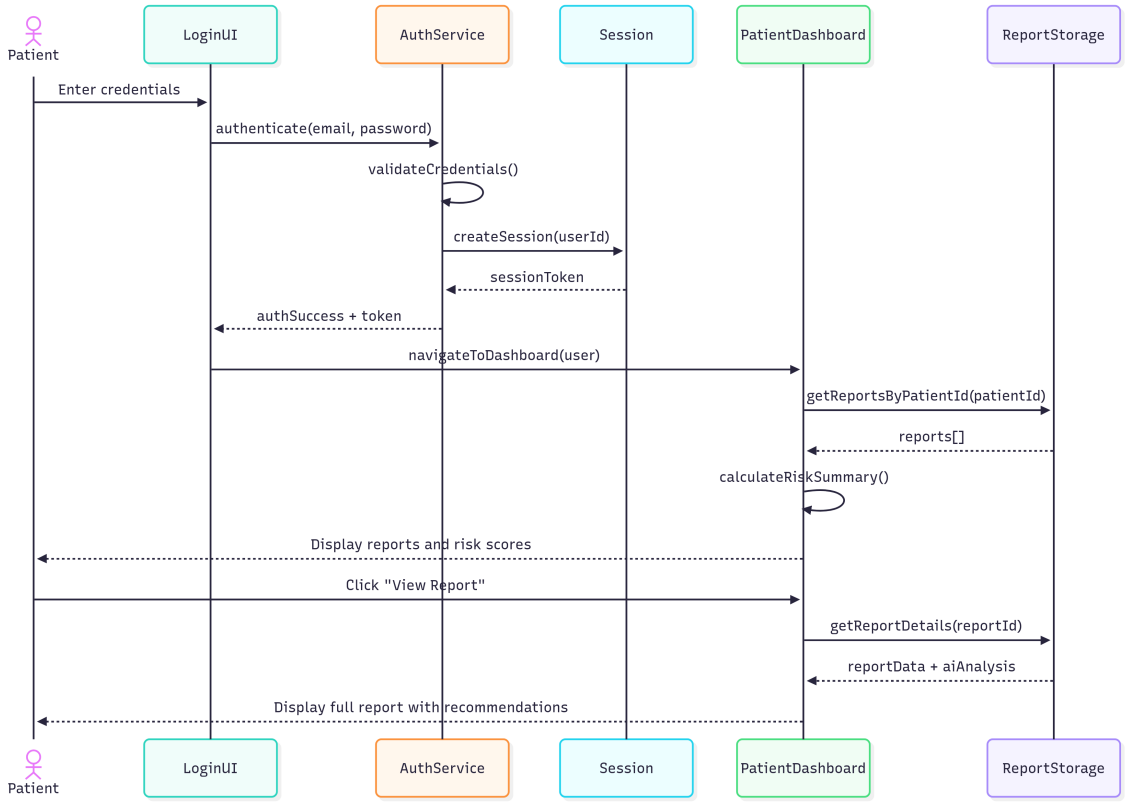


Figure 5: Patient Login and View Reports

4.1.2 Radiologist Upload X-Ray Report (AI-Powered)

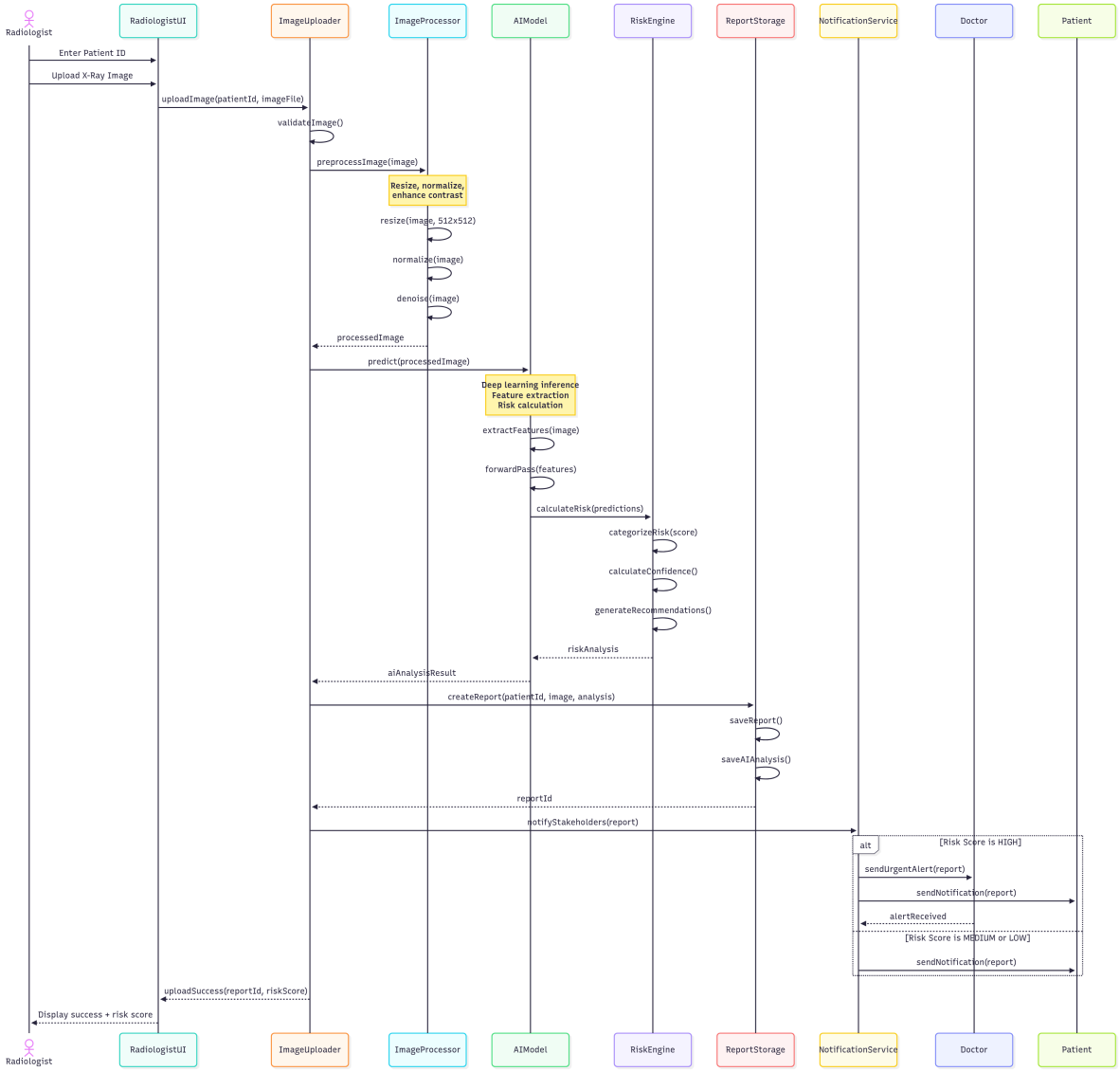


Figure 6: Radiologist Upload X-Ray Report (AI-Powered)

4.1.3 Doctor Review AI-Genrated Report

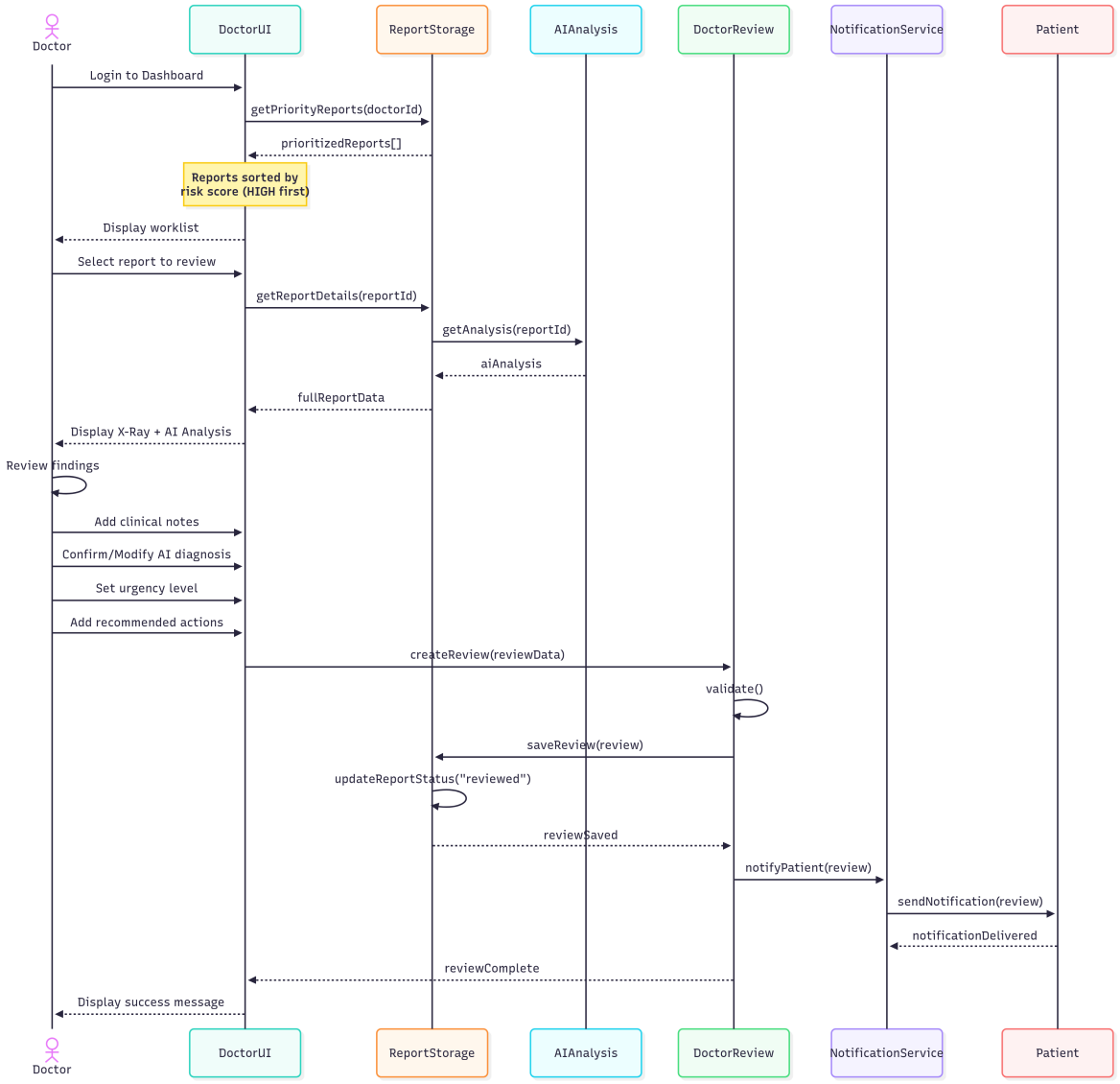


Figure 7: Doctor Review AI-Genrated Report

4.1.4 Complete User Journey: From Upload to Treatment

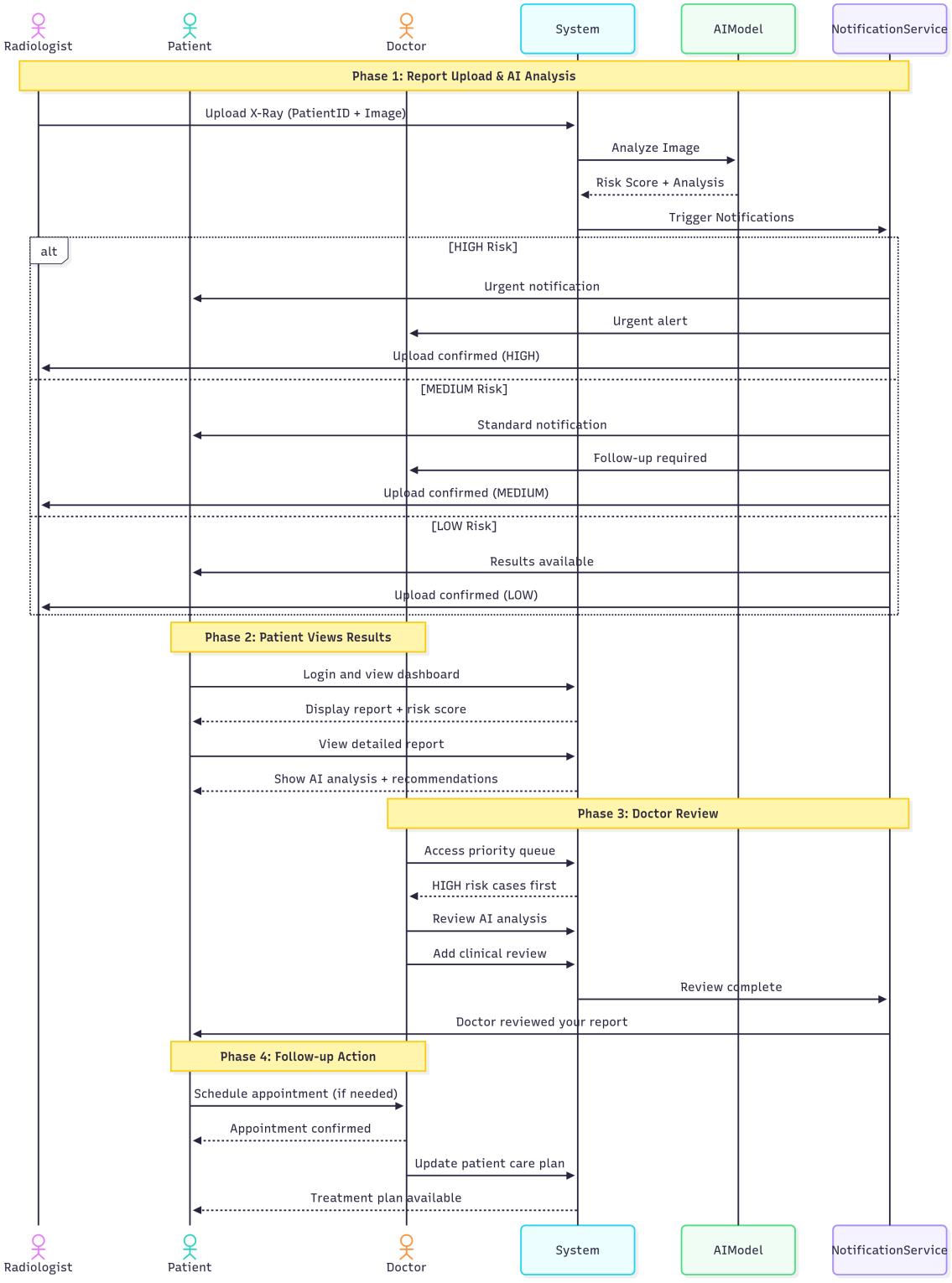


Figure 8: Complete User Journey: From Upload to Treatment

4.1.5 AI Model Prediction Pipeline

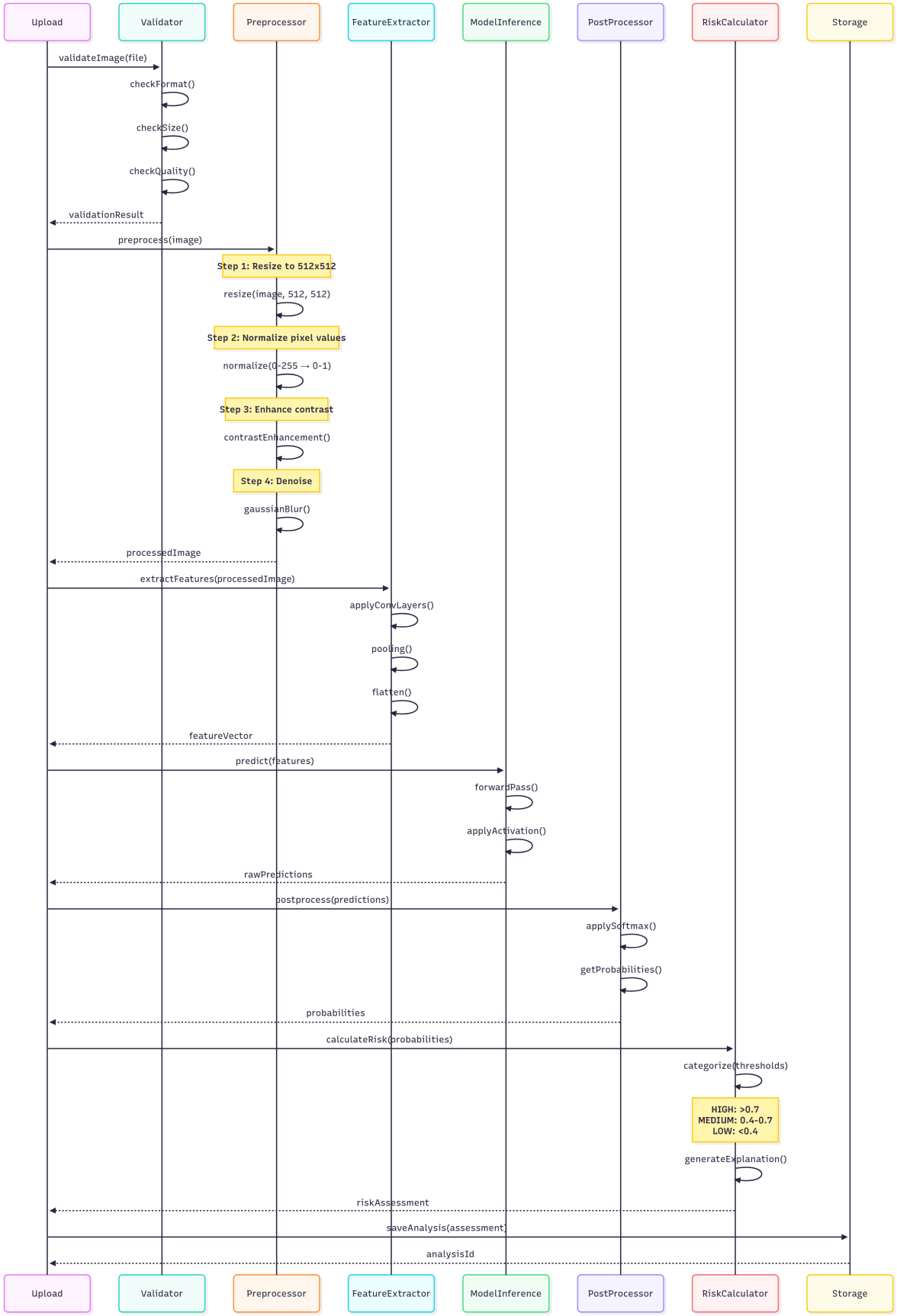


Figure 9: AI Model Prediction Pipeline

5 API Endpoints Reference

User Endpoints (/api/users)

POST /api/users/login

Purpose: Authenticate user and return user data

Request Body:

```
{
  "email": "string (required)",
  "password": "string (required)"
}
```

Success Response (200):

```
{
  "_id": "ObjectId",
  "email": "user@example.com",
  "role": "patient|radiologist|doctor"
}
```

Error Responses:

- 401: Invalid credentials
- 500: Server error

Notes: Password comparison is plain text (NOT production-ready). Use bcrypt in production.

GET /api/users

Purpose: Get all users (for selecting patients/doctors in upload form)

Success Response (200):

```
[
  {
    "_id": "ObjectId",
    "email": "user@example.com",
    "role": "patient|radiologist|doctor",
    "medicalReports": ["reportId1", "reportId2"],
    "createdAt": "ISO8601",
    "updatedAt": "ISO8601"
  }
]
```

Notes: Password field is excluded using .select('-password').

POST /api/users

Purpose: Create a new user account

Request Body:

```
{
  "email": "string (required, unique)",
  "password": "string (required)",
  "role": "patient|radiologist|doctor (required)"
}
```

Success Response (201):

```
{
  "_id": "ObjectId",
  "email": "user@example.com",
  "role": "patient"
}
```

Error Responses:

- 400: User already exists or validation error
- 500: Server error

GET /api/users/:id

Purpose: Get user by ID

Success Response (200):

```
{
  "_id": "ObjectId",
  "email": "user@example.com",
  "role": "patient",
  "medicalReports": ["reportId1"],
  "createdAt": "ISO8601",
  "updatedAt": "ISO8601"
}
```

Error Responses:

- 404: User not found
- 500: Server error

Report Endpoints (/api/reports)

POST /api/reports/upload

Purpose: Upload X-ray image and create report with async ML processing

Content-Type: multipart/form-data

Request Body (FormData):

```
{
  image: File (required, max 5MB),
  patientId: "ObjectId (required)",
  doctorId: "ObjectId (required)",
  radiologistId: "ObjectId (required)"
}
```

Success Response (201):

```
{
  "_id": "ObjectId",
  "patientId": "ObjectId",
  "doctorId": "ObjectId",
  "radiologistId": "ObjectId",
  "imageUrl": "/uploads/xray-123.jpg",
  "status": "pending",
  "createdAt": "ISO8601",
  "updatedAt": "ISO8601"
}
```

Error Responses:

- 400: Missing fields or invalid file
- 500: Server error

Async ML Processing Flow:

1. Save image to uploads/
2. Create report with status = pending
3. Spawn python3 risk_model.py process
4. Parse CSV output
5. Update report with ML results

GET /api/reports

Purpose: Get reports filtered by user role

Success Response (200):

```
[
  {
    "_id": "ObjectId",
    "patientId": { "_id": "ObjectId", "email": "patient@example.com" },
    "doctorId": { "_id": "ObjectId" },
    "radiologistId": { "_id": "ObjectId" },
    "imageUrl": "/uploads/xray.jpg",
    "riskScore": 78.5,
    "summary": "High cancer risk detected.",
    "recommendedNextSteps": "Consult oncology...",
    "status": "analyzed",
    "doctorReview": null,
    "mlMetadata": {
```

```
      "chexpertScore": 82.1,
      "biobertScore": 76.3,
      "xgboostScore": 80.2,
      "clinicalScore": 75.8,
      "positiveFindings": "Pneumonia, Pleural Effusion",
      "processedAt": "ISO8601"
    },
    "createdAt": "ISO8601",
    "updatedAt": "ISO8601"
  }
]
```

GET /api/reports/:id

Purpose: Get a single report by ID

Success Response (200):

```
{
  "_id": "ObjectId",
  "patientId": { "_id": "ObjectId", "email": "patient@example.com" },
  "doctorId": { /* populated */ },
  "radiologistId": { /* populated */ },
  "imageUrl": "/uploads/xray.jpg",
  "riskScore": 78.5,
  "summary": "Medical summary...",
  "recommendedNextSteps": "Recommended...",
  "status": "analyzed",
  "doctorReview": "Notes...",
  "mlMetadata": {
    "chexpertScore": 82.1,
    "biobertScore": 76.3,
    "xgboostScore": 80.2,
    "clinicalScore": 75.8,
    "positiveFindings": "Pneumonia, Consolidation",
    "processedAt": "ISO8601",
    "error": null
  }
}
```

PATCH /api/reports/:id

Purpose: Update report for ML output or doctor review

Request Body:

```
{
  "riskScore": 85.5,
  "summary": "Updated summary",
  "doctorReview": "Clinical notes",
  "status": "reviewed",
  "mlMetadata": {
    "chexpertScore": 80.0,
    "biobertScore": 75.0,
    "xgboostScore": 82.0,
    "clinicalScore": 70.0,
```

```
    "processedAt": "ISO8601",
    "error": null
  }
}
```

POST /api/reports (Legacy)

Purpose: Create report without image upload (rare)

Request Body:

```
{
  "patientId": "ObjectId",
  "doctorId": "ObjectId",
  "radiologistId": "ObjectId",
  "status": "pending"
}
```

ML Metadata Schema Reference

```
mlMetadata: {
  chexpertScore: Number,
  biobertScore: Number,
  xgboostScore: Number,
  clinicalScore: Number,
  positiveFindings: String,
  processedAt: Date,
  error: String,
  failedAt: Date
}
```

Ensemble formula:

```
riskScore = (biobertScore * 0.40) +
            (chexpertScore * 0.30) +
            (xgboostScore * 0.20) +
            (clinicalScore * 0.10)
```

6 FOLDER STRUCTURE

The project structure is as follows:

```
NNN_for_Cancer/
 Code/
    backend/                          # Node.js + Python Backend
       config/                       # Configuration files
          db.js                      # MongoDB connection handler
       models/                       # Mongoose database schemas
          User.js                    # User model (patient/doctor/radiologist)
          MedicalReport.js           # Medical report with ML metadata
       routes/                       # Express API endpoints
          userRoutes.js              # User auth & management routes
```

```
      reportRoutes.js          # Report CRUD + ML integration
   utils/                       # Utility functions
      mlPipeline.js            # ML pipeline integration layer
   uploads/                     # Uploaded X-ray images storage
      .gitkeep                 # Keep directory in git
      xray-*.png               # Uploaded image files
   Dockerfile                   # Backend container definition
   .dockerignore                # Docker build exclusions
   .env                         # Environment variables
   .gitignore                   # Git exclusions
   server.js                    # Express server entry point
   package.json                 # Node.js dependencies
   package-lock.json            # Locked dependency versions
   requirements.txt             # Python dependencies
   risk_model.py                # Main ML pipeline (900+ lines)
   train_xgboost_model.py       # XGBoost model training script
   xgboost_risk_model.pkl       # Trained XGBoost model
   feature_scaler.pkl           # Feature normalization scaler
   mock_model.ipynb             # Jupyter notebook for testing
   sample_report_high_risk.png  # Sample test image (high risk)
   sample_report_medium_risk.png # Sample test image (medium risk)
   x-ray-test.png               # Sample test image

frontend/                        # React + Vite Frontend
    public/                      # Static assets
       vite.svg                 # Vite logo
    src/                         # React source code
       assets/                  # Images, icons, fonts
       components/              # React components (26 files)
          Login.jsx             # Authentication screen
          Login.css             # Login styles
          Home.jsx              # Home page component
          Home.css              # Home page styles
          PatientDashboard.jsx # Patient dashboard view
          PatientDashboard.css # Patient dashboard styles
          PatientReportView.jsx # Patient report details
          PatientReportView.css # Patient report styles
          PatientFAQ.jsx        # Patient FAQ page
          PatientFAQ.css        # FAQ styles
          DoctorDashboard.jsx  # Doctor dashboard view
          DoctorDashboard.css  # Doctor dashboard styles
          DoctorReportView.jsx # Doctor report review page
          DoctorReportView.css # Doctor report styles
          RadiologistWorklist.jsx # Radiologist work queue
          RadiologistWorklist.css # Worklist styles
          RadiologistUpload.jsx # Image upload interface
          RadiologistUpload.css # Upload styles
          RadiologistReportInterface.jsx # Report view
          RadiologistReportInterface.css # Report view styles
          RadiologistArchived.jsx # Archived reports
          RadiologistArchived.css # Archive styles
          Reports.jsx           # General reports component
          Reports.css           # Reports styles
          Users.jsx             # User management component
          Users.css             # User management styles
       services/                # API communication layer
          api.js                # Axios API client
       App.jsx                  # Main app component with routing
       App.css                  # Global app styles
```

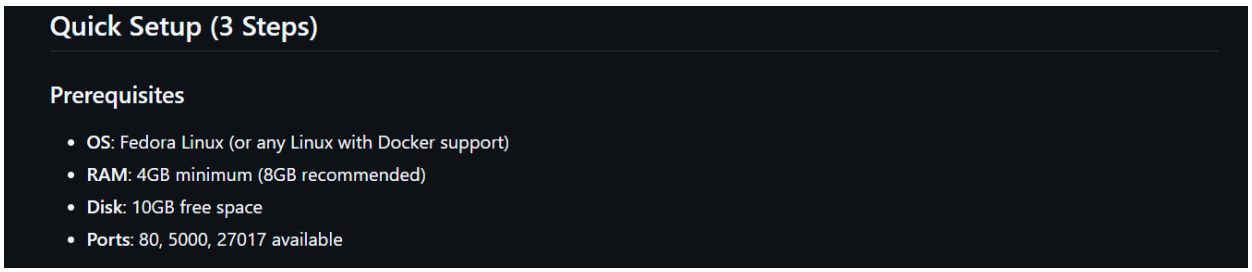
```
        index.css                # Root CSS styles
        main.jsx                 # React entry point
    Dockerfile                    # Frontend container (multi-stage)
    .dockerignore                 # Docker build exclusions
    .env                          # Frontend environment variables
    .gitignore                    # Git exclusions
    nginx.conf                    # Nginx web server configuration
    package.json                  # Frontend dependencies
    package-lock.json             # Locked dependency versions
    vite.config.js                # Vite build configuration
    eslint.config.js              # ESLint configuration
    index.html                    # HTML template

Docs/                              # Project documentation & diagrams
  sequence-diagrams/              # System workflow diagrams
    AI_Model_Prediction_Pipeline.png
    Complete_User_Jounney_from_upload_to_Treatment.png
    Doctor_Review_AI-Generated_Report.png
    patient_login_and_view_reports.png
    Radiologist_upload_X-ray_Report.png
    Tech_Team_Monitor_AI_Model_Performance.png
  schema.sql                      # Initial database schema
  Architecture_design.png         # System architecture diagram
  Entity-relation.png             # ER diagram
  API_signture.png                # API signature diagram
  UML.png                         # UML diagrams
  api.pdf                         # API documentation
  final_report.pdf                # Complete project report

docker-compose.yml                 # Docker orchestration file
.env                               # Root environment variables
deploy.sh                          # Deployment automation script
stop.sh                            # Stop services script
backup.sh                          # Database backup script
check-requirements.sh              # System requirements checker
PROJECT_SETUP.md                   # This file - complete setup guide
START_HERE.md                      # Quick start guide
DEPLOYMENT_GUIDE.md                # Comprehensive deployment guide
DEPLOYMENT_SUMMARY.md              # Deployment overview
DOCKER_README.md                   # Docker commands reference
```

7 SETUP INSTRUCTIONS

7.1 Quick Setup



Step 1: Install Docker

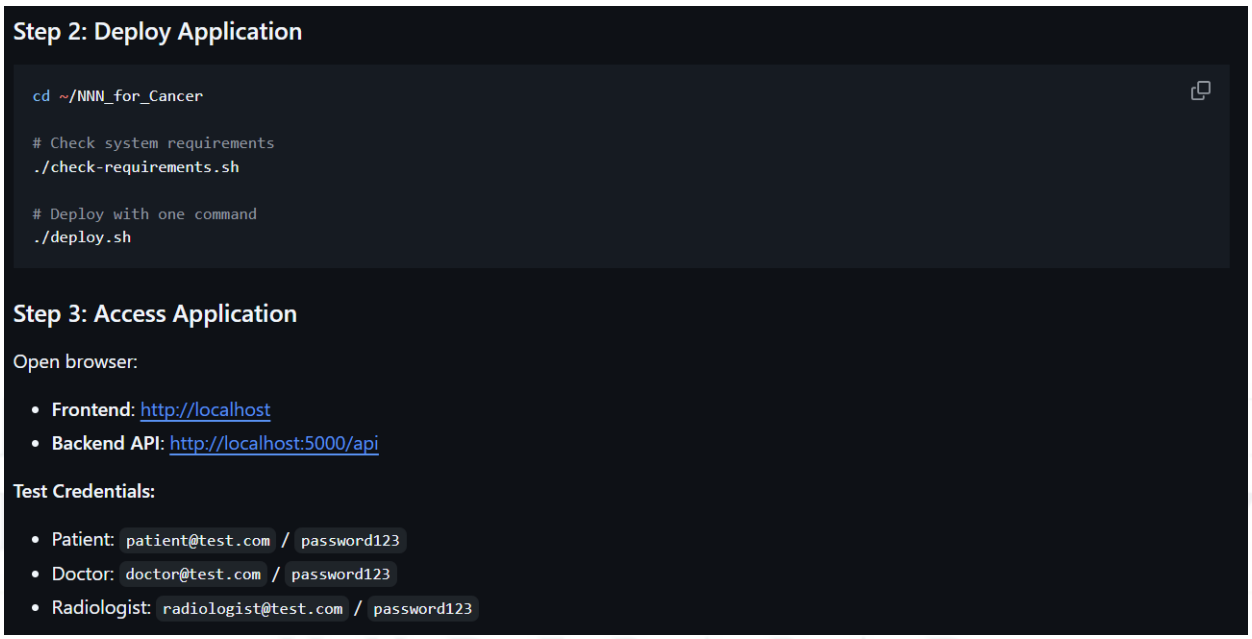
```
# Install Docker on Fedora
sudo dnf -y install dnf-plugins-core
sudo dnf config-manager --add-repo https://download.docker.com/linux/fedora/docker-ce.repo
sudo dnf install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin

# Start Docker
sudo systemctl start docker
sudo systemctl enable docker

# Add user to docker group
sudo usermod -aG docker $USER

# Apply group changes (or log out and back in)
newgrp docker

# Verify installation
docker --version
docker compose version
```



Step 2: Deploy Application

```
cd ~/NNN_for_Cancer

# Check system requirements
./check-requirements.sh

# Deploy with one command
./deploy.sh
```

Step 3: Access Application

Open browser:

- **Frontend**: http://localhost
- **Backend API**: http://localhost:5000/api

Test Credentials:

- Patient: `patient@test.com` / `password123`
- Doctor: `doctor@test.com` / `password123`
- Radiologist: `radiologist@test.com` / `password123`

7.2 Detailed Setup

For full details of the project setup , please refer to the repository on GitHub:

Cancer Stratification System - Github

Detailed Setup Instructions

Option A: Docker Deployment (Recommended)

1. Clone/Navigate to Project:

```
cd ~/NNN_for_Cancer
```

2. Configure Environment:

```
# Create environment file
cp .env.example .env

# Edit if needed
nano .env
```

3. Build & Deploy:

```
# Build all Docker images
docker compose build

# Start all services
docker compose up -d

# Check status
docker compose ps

# View logs
docker compose logs -f
```

4. Verify Deployment:

```
# Test backend
curl http://localhost:5000/api/users

# Test frontend
curl http://localhost

# Access MongoDB
docker exec -it cancer-stratification-db mongosh -u admin -p adminpassword123
```

Option B: Manual Setup (Development)

Backend Setup:

```
cd Code/backend

# Install Node.js dependencies
npm install

# Install Python dependencies
pip3 install -r requirements.txt

# Install Tesseract OCR
sudo dnf install tesseract tesseract-langpack-eng

# Create uploads directory
mkdir -p uploads

# Set environment variables
export MONGODB_URI="mongodb://localhost:27017/cancer_stratification"
export PORT=5000

# Start backend
npm start
```

Frontend Setup:

```
cd Code/frontend

# Install dependencies
npm install

# Start development server
npm run dev
```

Database Setup:

```
# Install MongoDB
sudo dnf install mongodb-org

# Start MongoDB
sudo systemctl start mongod

# Create database and users (use mongosh)
```

8 INDIVIDUAL CONTRIBUTIONS

- **Nilavra Ghosh** — Led the codebase setup, explored and compared possible technology stacks, management, feedback and contributed to Design documentation(Figma/ER Diagram/Sequence Diagram) and workflow structuring.
- **Ved Prakash Maurya** —Integrated the backend pipeline alongside Aviral , performed functional and regression testing, and ensured seamless interaction between backend services and frontend modules .
- **Aviral Malhotra** — Worked on the frontend development , integrated model outputs into the UI alongside Ved, coordinated the design of user flows to ensure a smooth patient–doctor interaction .
- **Bhaskar Bhatt** — Supported backend integration tasks, collaborated on data preprocessing pipelines, and contributed to system testing and debugging.
- **Akshat** — Worked on testing and integration methodology, evaluated model outputs during deployment trials, assisted in pipeline integration, and helped refine the system through iterative testing cycles.
- **Aayush** — Conducted research on the problem statement, reviewed medical literature, and helped refine the scope and framing of the project from a social innovation perspective.