AI-Powered Multi-Modal Disease Prediction System

Report

Thanh Tran

May 15, 2025

Abstract: This report details the system design and implementation of an advanced AI-powered disease prediction application. The system leverages comprehensive health data, including medical imagery (Chest X-rays from the NIH dataset), tabular patient metadata, and simulated physiological sensor data, to predict the likelihood of 14 common thoracic diseases. Architected as a suite of containerized microservices using Docker, the system employs Python, FastAPI, PyTorch, PostgreSQL, MinIO, and Gradio to deliver an end-to-end solution from data ingestion and preprocessing to model training and interactive prediction. Key features include a multi-modal fusion model, automated data processing pipelines triggered via background tasks, and a user-friendly interface for inference.

Contents

1. Introduction 2
2. System Overview and Architecture 3
3. Microservice Details 4
   3.1. API Gateway (`api_gateway`) 4
   3.2. Data Ingestion Service (`data_ingestion_service`) 4
   3.3. Patient Data Service (`patient_data_service`) 5
   3.4. Image Preprocessing Service (`image_preprocessing_service`) 6
   3.5. Tabular Preprocessing Service (`tabular_preprocessing_service`) 6
   3.6. Model Training Service (`model_training_service`) 7
   3.7. Disease Prediction Service (`disease_prediction_service`) 7
   3.8. Frontend Service (`frontend_service`) 8
4. Data Management 9
   4.1. Data Sources 9
   4.2. Data Ingestion Pipeline 9
   4.3. Data Preprocessing 10
   4.4. Data Storage 11
5. Machine Learning Pipeline 12
   5.1. Feature Preparation (for Inference) 12
   5.2. Model Architecture 12
      5.2.1. Model Training (`model_training_service`) 13
      5.2.2. Model Deployment and Serving (`disease_prediction_service`) 14
6. Prediction Workflow 15
7. Technology Stack Summary 17
8. Deployment 18
9. Conclusion 19

1. Introduction

This report details the system design for an advanced AI-powered disease prediction application. The primary goal of this system is to accurately predict potential diseases an individual may encounter by leveraging comprehensive health data from disparate sources. This includes medical imagery (e.g., chest X-rays), tabular health history (e.g., NIH dataset information), and real-time sensor data.

The system is architected as a microservice-based application, promoting scalability, maintainability, and flexibility. It adheres to the technology stack recommendations, utilizing Python for backend and ML components, FastAPI for API development, PyTorch for model building, Gradio for the user interface, PostgreSQL for structured data storage, MinIO for object storage, and Docker for containerization and deployment.

This document provides an overview of the system architecture, details of each microservice, data management strategies, the machine learning pipeline, prediction workflow, and deployment considerations. The complexity and modularity of the solution aim to demonstrate a robust understanding of modern AI system development practices.

2. System Overview and Architecture

The disease prediction system is designed as a distributed collection of microservices that work in concert to deliver its functionalities. This architecture was chosen for its scalability, resilience, and ease of independent development and deployment of components.

The core components of the system include:

- **API Gateway:** Single entry point for all client requests, routing them to appropriate backend services.
- **Data Ingestion Service:** Responsible for receiving raw data (images, tabular CSVs, sensor readings) and initiating their processing.
- **Patient Data Service:** Manages patient demographic information, study metadata, and paths to processed features. It acts as the central metadata repository.
- **Image Preprocessing Service:** Processes raw medical images, extracts relevant features using a pre-trained deep learning model, and stores these features.
- **Tabular Preprocessing Service:** Processes raw tabular data (both NIH metadata and sensor data), performs cleaning, transformation, and feature engineering, and stores the processed features.
- **Model Training Service:** Orchestrates the training of the core multi-modal fusion model using the processed features from various sources.
- **Disease Prediction Service:** Serves the trained fusion model to make predictions based on input patient data (image features, tabular features, sensor features).
- **Frontend Service:** Provides a user-friendly web interface (built with Gradio) for users to input data and receive disease predictions.
- **Data Stores:**
  - **PostgreSQL:** Used by the Patient Data Service for storing structured metadata about patients and studies.
  - **MinIO:** Used as an object store for raw uploaded data (images, CSVs), processed features, and trained machine learning models.

The services are containerized using Docker and managed via Docker Compose for local development and deployment orchestration.

3. Microservice Details

3.1. API Gateway (`api_gateway`)

The API Gateway serves as the unified entry point for all incoming requests to the system. It simplifies client interaction by providing a single interface and routes requests to the appropriate downstream microservices.

- **Technology:** FastAPI (Python)
- **Responsibilities:**
  - Request routing to `data_ingestion_service`, `disease_prediction_service`, `patient_data_service`, `image_preprocessing_service`, and `tabular_preprocessing_service`.
  - Aggregating responses (if necessary, though currently it primarily forwards requests).
  - Potentially handling cross-cutting concerns like authentication, rate limiting, and logging in a more mature version.
- **Key Files:** `main.py`, `config.py`, router files in `routers/` directory.
- **Endpoints Exposed (example via gateway):**
  - `/api/v1/ingest/...` (routes to Data Ingestion Service)
  - `/api/v1/predict/...` (routes to Disease Prediction Service)
  - `/api/v1/patients/...` (routes to Patient Data Service)
  - `/api/v1/studies/...` (routes to Patient Data Service)
  - `/api/v1/preprocess/image/...` (routes to Image Preprocessing Service)
  - `/api/v1/preprocess/tabular/...` (routes to Tabular Preprocessing Service)

3.2. Data Ingestion Service (`data_ingestion_service`)

This service is responsible for handling the initial intake of various data types. It uploads raw data to MinIO and triggers downstream preprocessing tasks.

- **Technology:** FastAPI (Python), MinIO client, httpx.
- **Responsibilities:**
  - Receiving batch data uploads containing:
    - Chest X-ray images (PNG).
    - NIH metadata CSV (`sampled_nih_metadata.csv`).
    - Simulated sensor data CSV (`simulated_sensor_data.csv`).
  - Storing raw image files in the `raw-images` MinIO bucket.
  - Storing raw NIH metadata and sensor data CSVs in their respective MinIO buckets (`raw-tabular`, `raw-sensor-data-per-study`).
  - Interacting with the `patient_data_service` to create/update patient and study records.
  - Asynchronously triggering the `image_preprocessing_service` and `tabular_preprocessing_service` for each ingested study.
- **Key Files:** `main.py`, `minio_client.py`, `patient_service_client.py`, `utils.py`.
- **Endpoints:**
  - `POST /ingest/batch/`: Accepts lists of image files, a metadata CSV file, and a sensor data CSV file.
- **Tesing Endpoints:**

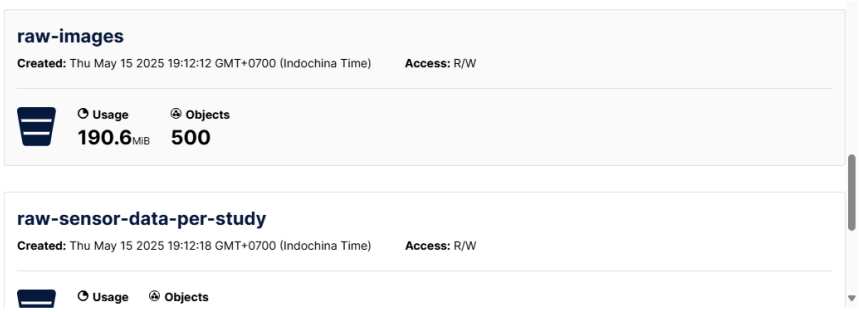

Figure 1: Data Ingested successfully to MinIO. This screenshot shows the MinIO web interface with the `raw-images` and `raw-sensor-data-per-study` buckets populated with files.

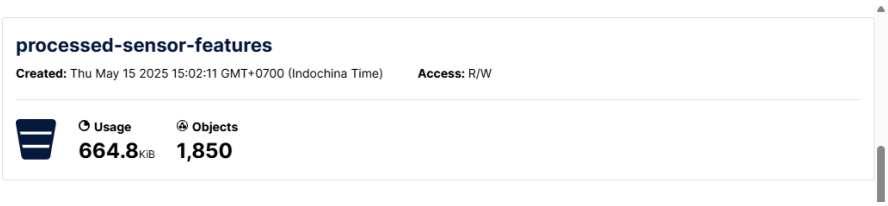

Figure 2: After ingestion, the `data_ingestion_service` triggers the `image_preprocessing_service` and `tabular_preprocessing_service` to process the raw data. This screenshot shows the MinIO web interface with the `processed-sensor-features` buckets populated with processed files.

3.3. Patient Data Service (`patient_data_service`)

This service acts as a central repository for patient demographics, study information (including links to raw and processed data in MinIO), and disease labels.

- **Technology:** FastAPI (Python), PostgreSQL, SQLAlchemy, Alembic (for migrations).
- **Responsibilities:**
  - CRUD operations for Patients (`Patient` model): Stores patient ID (internal and source), age, gender.
  - CRUD operations for Studies (`Study` model): Stores image index, finding labels, view position, paths to raw image data, raw tabular data, raw sensor data, and paths to processed image, tabular, and sensor features in MinIO.
  - Storing extracted disease labels associated with each study.
  - Providing query capabilities for other services to retrieve patient/study metadata.
- **Key Files:** `main.py`, `crud.py`, `models.py`, `schemas.py`, `database.py`, Alembic migration files in `alembic/versions/`.
- **Database Schema:**
  - `patients` table: `id`, `patient_id_source`, `age_years`, `gender`.
  - `studies` table: `id`, `patient_id` (FK), `image_index`, `finding_labels`, `follow_up_number`, `view_position`, `image_raw_path`, `tabular_raw_path`, `sensor_raw_path`, `image_feature_path`, `nih_tabular_feature_path`, `sensor_tabular_feature_path`.
- **Endpoints:**
  - `POST /patients/`: Create or retrieve a patient.
  - `GET /patients/{patient_id_source}`: Get patient by source ID.
  - `POST /studies/`: Create or update a study.
  - `GET /studies/{image_index}`: Get study by image index.
  - `PUT /studies/{study_id}/image-feature-path`: Update image feature path for a study.
  - `PUT /studies/{study_id}/tabular-feature-paths`: Update tabular (NIH and sensor) feature paths for a study.

- `GET /studies/`: Get all studies with optional query parameters.

3.4. Image Preprocessing Service (`image_preprocessing_service`)

This service is dedicated to processing raw medical images (chest X-rays). It extracts features using a pre-trained Convolutional Neural Network (CNN) and stores them.

- **Technology:** FastAPI (Python), PyTorch, torchvision, MinIO client, Pillow.
- **Responsibilities:**
  - Retrieving raw images from the `raw-images` MinIO bucket based on a study ID.
  - Preprocessing images: resizing, normalization (as per `config.py`).
  - Extracting image features using a pre-trained model (e.g., ResNet50, as indicated in `image_processor.py`). The final layer of the CNN is typically used as the feature vector.
  - Storing the extracted image features (as PyTorch tensors) in the `processed-image-features` MinIO bucket.
  - Updating the corresponding study record in `patient_data_service` with the path to the processed image features.
- **Key Files:** `main.py`, `image_processor.py`, `minio_utils.py`, `patient_service_client.py`.
- **Endpoints:**
  - `POST /preprocess/`: Accepts a `study_id` to trigger preprocessing for the associated image.

3.5. Tabular Preprocessing Service (`tabular_preprocessing_service`)

This service handles the preprocessing of two types of tabular data: NIH patient/study metadata and simulated sensor data.

- **Technology:** FastAPI (Python), Pandas, scikit-learn, NumPy, MinIO client.
- **Responsibilities:**
  - **NIH Metadata Processing (`nih_processor.py`):**
    - Retrieving raw NIH metadata CSV from MinIO.
    - Cleaning data (e.g., parsing age from string like "060Y").
    - Performing one-hot encoding for categorical features (e.g., `Patient Gender`, `View Position`).
    - Scaling numerical features.
    - Storing the processed NIH tabular features (as NumPy arrays or pickled DataFrames/ encoders) in the `processed-tabular-nih-features` MinIO bucket.
  - **Sensor Data Processing (`sensor_processor.py`):**
    - Retrieving raw sensor data CSV (per study) from MinIO.
    - Aggregating time-series sensor data into summary statistics (mean, std, min, max, etc.) for relevant columns (e.g., `HeartRate_bpm`, `RespiratoryRate_bpm`).
    - Storing the processed sensor features in the `processed-sensor-features` MinIO bucket.
  - Updating the corresponding study record in `patient_data_service` with paths to these processed tabular features.
- **Key Files:** `main.py`, `nih_processor.py`, `sensor_processor.py`, `minio_utils.py`, `patient_service_client.py`.
- **Endpoints:**
  - `POST /preprocess/nih-metadata/`: Accepts `study_id` and `raw_file_path` to trigger NIH metadata preprocessing.
  - `POST /preprocess/sensor-data/`: Accepts `study_id` and `raw_file_path` to trigger sensor data preprocessing.

3.6. Model Training Service (`model_training_service`)

This service is responsible for training the multi-modal fusion model that combines image, NIH tabular, and sensor features for disease prediction.

- **Technology:** FastAPI (Python), PyTorch, Pandas, NumPy, scikit-learn, MinIO client, subprocess.
- **Responsibilities:**
  - Exposing an endpoint to initiate a model training job.
  - Training jobs run as background tasks (`training_manager.py`).
  - The actual training logic is encapsulated in scripts (`scripts/train_fusion_model.py`):
    - Loading processed features (image, NIH tabular, sensor) and labels from MinIO/Patient Data Service via `data_loader.py`.
    - Defining the fusion model architecture (e.g., `AttentionFusionMLP` from `model_def.py`).
    - Executing the training loop, including loss calculation (e.g., BCEWithLogitsLoss for multi-label classification) and optimization.
    - Evaluating the model using metrics like ROC AUC, F1-score, Precision, Recall, Hamming Loss.
    - Saving the trained model components (fusion model state dict, image feature extractor state dict if fine-tuned, label binarizer, encoders/scalers) to the `models-store` MinIO bucket.
- **Key Files:** `main.py`, `training_manager.py`, `scripts/train_fusion_model.py`, `scripts/data_loader.py`, `scripts/model_def.py`, `scripts/config_training.py`.
- **Endpoints:**
  - `POST /train/`: Accepts a `TrainingJobRequest` to start a new training job. Returns a `TrainingJobResponse` with a job ID.
  - `GET /train/status/{job_id}`: Checks the status of a training job.

3.7. Disease Prediction Service (`disease_prediction_service`)

This service is responsible for loading the trained multi-modal fusion model and making disease predictions based on provided patient data.

- **Technology:** FastAPI (Python), PyTorch, MinIO client, scikit-learn, Pandas, NumPy.
- **Responsibilities:**
  - Loading the trained fusion model (`AttentionFusionMLP`) and associated components (image feature extractor, encoders, scalers, label binarizer) from the `models-store` MinIO bucket during startup (`models_loader.py`).
  - Accepting input data for prediction:
    - Raw image file (for on-the-fly feature extraction).
    - NIH tabular data (as a JSON string or dictionary).
    - Sensor data (as a CSV string or structured format).
  - Preparing input data for the model using logic from `feature_preparation.py`, which mirrors the preprocessing steps from training. This includes:
    - Image feature extraction.
    - NIH tabular data encoding and scaling.
    - Sensor data aggregation and scaling.
  - Performing inference using the fusion model.
  - Returning the predicted disease probabilities for each class.
- **Key Files:** `main.py`, `models_loader.py`, `model_def.py`, `feature_preparation.py`, `config.py`.
- **Endpoints:**

- `POST /predict/`: Accepts an image file, NIH tabular data (JSON string), and sensor data (CSV string) to make a prediction.

3.8. Frontend Service (`frontend_service`)

Provides a web-based user interface for interacting with the disease prediction system.

- **Technology:** Gradio (Python), httpx (for API calls).
- **Responsibilities:**
  - Offering input fields for users to upload:
    - A chest X-ray image.
    - NIH tabular data (e.g., patient age, gender, view position - simplified for UI).
    - Sensor data (e.g., average heart rate, respiratory rate - simplified for UI).
  - Calling the `disease_prediction_service` (via the API Gateway) with the provided data.
  - Displaying the returned disease predictions (probabilities and predicted labels).
- **Key Files:** `app.py`.
- **Interface Components (based on typical Gradio usage):**
  - Image upload component.
  - Text/Number inputs for tabular and sensor data.
  - Output component to display prediction results (e.g., labels with probabilities).

4. Data Management

4.1. Data Sources

The system is designed to integrate data from multiple sources, reflecting real-world health data diversity:

1. **Medical Imagery:** Chest X-ray images (e.g., PNG format). The initial dataset is based on the NIH Chest X-ray dataset, with a sample metadata file `sampled_nih_metadata.csv`.
2. **Tabular Health History:** Metadata associated with the images, including patient demographics (age, gender), view position, and importantly, the ground truth `Finding Labels`.
3. **Sensor Data:** Simulated time-series sensor data (e.g., heart rate, respiratory rate, SpO2, temperature, blood pressure) generated by `scripts/generate_simulated_sensor_data.py` and stored in `simulated_sensor_data.csv`. This data is linked to specific studies (images).

Currently I used the NIH dataset (https://www.kaggle.com/datasets/nih-chest-xrays/sample) for the tabular data and image data, but the system is designed to be extensible to other datasets. To ensure a more diversity features and robust system, I created `scripts/generate_simulated_sensor_data.py` script to generates sensor data for each study in the NIH dataset, ensuring that the sensor data aligns with the corresponding images.

4.2. Data Ingestion Pipeline

The ingestion process is initiated through the `data_ingestion_service`:

1. A batch ingestion script (`scripts/run_batch_ingestion.py`) can be used to simulate the arrival of new patient data. This script reads image file paths, NIH metadata, and sensor data, then sends them to the `/ingest/batch/` endpoint of the Data Ingestion Service (via the API Gateway).
2. The `data_ingestion_service`: a. Uploads raw image files to the `raw-images` bucket in MinIO. b. Uploads raw NIH metadata (for the batch) and splits/uploads sensor data per study to `raw-tabular` and `raw-sensor-data-per-study` buckets respectively. c. For each study (image index): i. Creates or retrieves patient information in the `patient_data_service`. ii. Creates or updates study information in the `patient_data_service`, linking the patient and storing paths to the raw data in MinIO. iii. Asynchronously calls the `image_preprocessing_service` to process the raw image. iv. Asynchronously calls the `tabular_preprocessing_service` to process both the NIH metadata portion relevant to the study and the specific sensor data for that study.

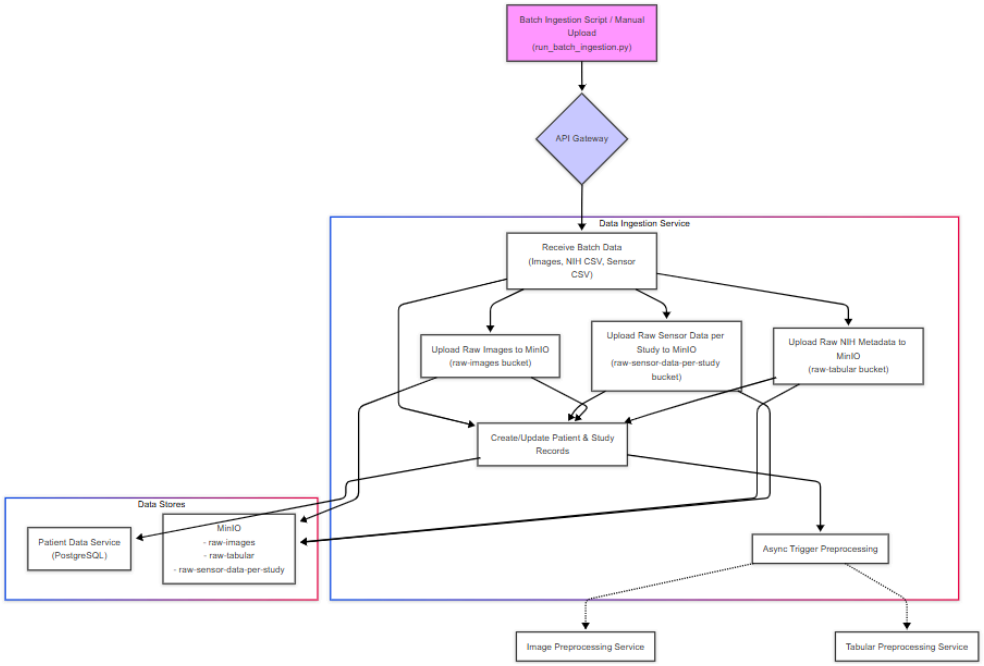

Figure 3: Data Ingestion Flow Diagram. Details the sequence from batch ingestion to storage and preprocessing triggers.

4.3. Data Preprocessing

Preprocessing is handled by specialized services:

1. **Image Preprocessing (`image_preprocessing_service`):**
   - Fetches raw image from MinIO.
   - Applies transformations: resize to a fixed size (e.g., 224x224), converts to tensor, normalizes using pre-defined mean and standard deviation values.
   - Passes the preprocessed image through a pre-trained CNN (e.g., ResNet50) to extract a feature vector.
   - Saves the feature vector (tensor) to the `processed-image-features` bucket in MinIO.
   - Updates the study record in `patient_data_service` with the MinIO path of the features.
2. **Tabular Preprocessing (`tabular_preprocessing_service`):**
   - **NIH Metadata (`nih_processor.py`):**
     - Fetches relevant study metadata from MinIO (or directly if passed).
     - Cleans data (e.g., 'Patient Age' string to integer).
     - Applies One-Hot Encoding to categorical columns (`NIH_CATEGORICAL_COLS`).
     - Applies Standard Scaling to numerical columns (`NIH_NUMERICAL_COLS`).
     - Saves the processed features (NumPy array) to `processed-tabular-nih-features` MinIO bucket.
   - **Sensor Data (`sensor_processor.py`):**
     - Fetches raw sensor data CSV for the study from MinIO.
     - Aggregates time-series data into features using operations like mean, std, min, max for defined columns (`SENSOR_COLUMNS_TO_AGGREGATE`, `SENSOR_AGGREGATIONS`).
     - Saves the processed features (NumPy array) to `processed-sensor-features` MinIO bucket.
   - Updates the study record in `patient_data_service` with MinIO paths for both types of tabular features.

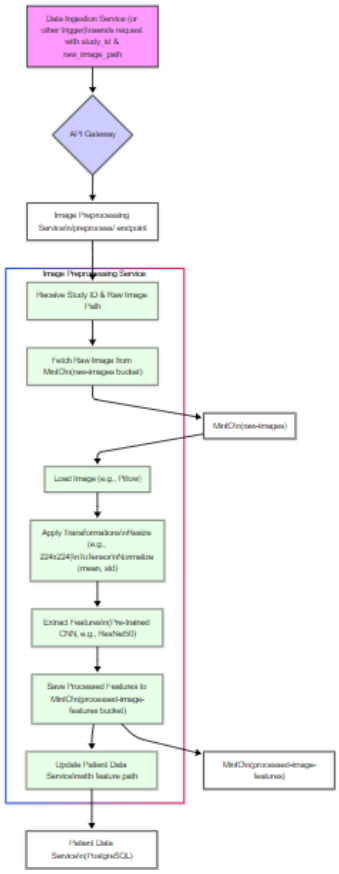

Figure 4: Detailed diagram for Image Preprocessing flow.

4.4. Data Storage

The system utilizes two primary storage solutions:

1. **MinIO:** An S3-compatible object storage service. It is used for storing:
   - Raw uploaded data: images, tabular CSVs (NIH metadata, sensor data).
   - Processed data: extracted image features, processed NIH tabular features, processed sensor features.
   - Trained machine learning models and associated artifacts (encoders, scalers, label binarizers).
   - Buckets are defined in `docker-compose.yml` and service configurations (e.g., `BUCKET_RAW_IMAGES`, `BUCKET_PROCESSED_IMAGE_FEATURES`, `BUCKET_MODELS_STORE`).
2. **PostgreSQL:** A relational database used by the `patient_data_service`.
   - Stores structured metadata about patients and their studies.
   - Includes paths/keys to the actual data objects stored in MinIO, linking the metadata with the binary data.
   - Schema is defined in `patient_data_service/app/models.py` and managed by Alembic migrations.

5. Machine Learning Pipeline

5.1. Feature Preparation (for Inference)

The `disease_prediction_service` prepares features for inference similarly to the training pipeline (`feature_preparation.py`):

- **Image Features:** Raw image is preprocessed (resized, normalized) and fed through the loaded image feature extractor model.
- **NIH Tabular Features:** Input JSON data is converted to a DataFrame, categorical columns are one-hot encoded using the loaded encoder, and numerical columns are scaled using the loaded scaler. Missing values are handled (e.g., by imputation or default values).
- **Sensor Features:** Input CSV data is aggregated (mean, std, etc.) and then potentially scaled using loaded sensor data scalers if applicable (current implementation seems to use raw aggregated values directly for prediction, but scaling would be a good practice if done during training).

5.2. Model Architecture

The core prediction model is a multi-modal fusion network, specifically an `AttentionFusionMLP` (defined in `model_training_service/scripts/model_def.py` and `disease_prediction_service/app/model_def.py`).

- It takes features from the three modalities (image, NIH tabular, sensor) as input.
- Instead of using the concatenated features, I choose `AttentionFusionMLP` model, which includes an attention mechanism to weigh the importance of different feature modalities before feeding them into subsequent MLP layers. Input features from image, NIH tabular data, and sensor data are first processed by separate linear layers. The outputs are then concatenated and passed through an attention layer. The attention weights are learned to emphasize more relevant modalities for a given prediction. The attended features are then passed through further MLP layers.
- The MLP consists of several hidden layers with activation functions (e.g., ReLU) and dropout for regularization.
- The output layer uses a sigmoid activation function for each class, suitable for multi-label disease prediction. The number of output neurons corresponds to the number of disease classes (14 diseases + "No Finding" as defined in `config_training.py`).

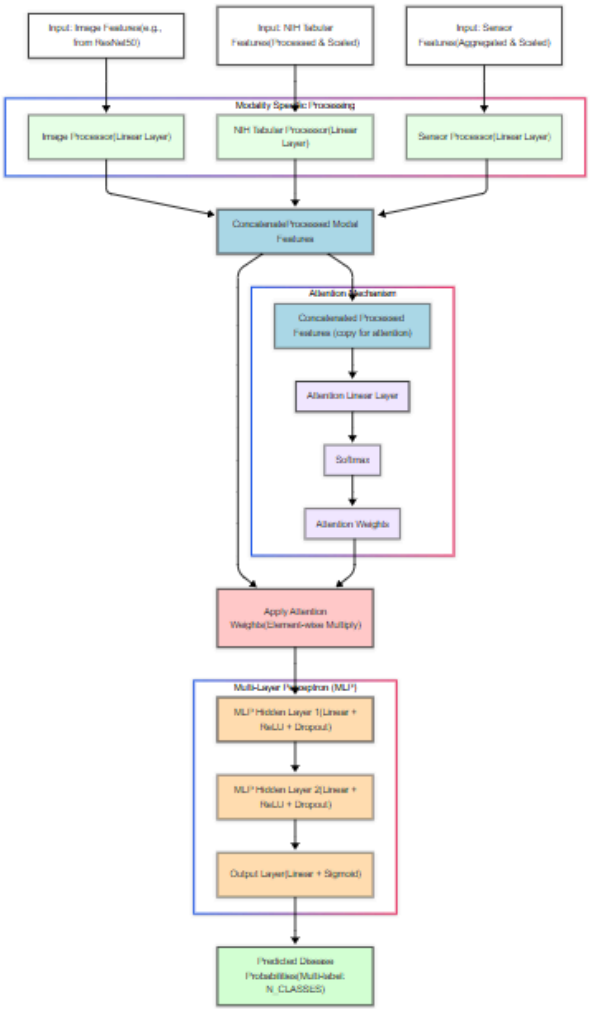

Figure 5: Diagram of the Attention Fusion Model Architecture. Shows input feature vectors, modality-specific processing, attention mechanism, MLP layers, and multi-label output.

5.2.1. Model Training (`model_training_service`)

The training process is orchestrated by the `model_training_service` using scripts:

1. **Data Loading (`data_loader.py`):**
   - Fetches study metadata (including feature paths and labels) from the `patient_data_service`.
   - Creates a custom PyTorch `Dataset` (`FusionDataset`). This dataset is responsible for:
     - Loading the processed image features, NIH tabular features, and sensor features from their respective MinIO buckets for each study.
     - Generating `nih_attention_mask` and `sensor_attention_mask`. These binary masks indicate whether the NIH tabular data or sensor data, respectively, is present for a given sample. **This is crucial for handling missing modalities**, as the `data_loader.py` checks for the existence of feature files.
   - Labels (`Finding Labels`) are multi-hot encoded using `MultiLabelBinarizer` from scikit-learn. The `ALL_DISEASE_CLASSES` list in `config_training.py` ensures consistent class ordering.
   - Uses PyTorch `DataLoader` for batching and shuffling.
2. **Training Loop:**
   - Initializes the `AttentionFusionMLP` model, loss function (BCEWithLogitsLoss), and optimizer (e.g., AdamW).
   - Iterates through epochs and batches.
   - For each batch:

        - Unpacks data including image features, NIH features, sensor features, NIH attention mask, sensor attention mask, and labels.
        - Performs forward pass by feeding all features and their corresponding attention masks to the model: `model(image_features, nih_features, sensor_features, nih_attention_mask, sensor_attention_mask)`.
        - Calculates loss.
        - Performs backward pass and optimizer step.
    - Uses AMP (Automatic Mixed Precision) for potentially faster training.
    - Validation is performed periodically to monitor performance on a hold-out set, also utilizing the attention masks.
3. **Evaluation:** Metrics such as ROC AUC (per-class and macro/micro averaged), F1-score, Precision, Recall, and Hamming Loss are calculated.
4. **Model Saving:**
    - The trained fusion model's state dictionary is saved.
    - The image feature extractor's state dictionary (if fine-tuned, though current setup seems to use it as a fixed extractor) is saved.
    - The `MultiLabelBinarizer` (MLB) instance is saved using `joblib` or `pickle` and stored in MinIO.
    - Other preprocessing artifacts like one-hot encoders and scalers for tabular data are also saved to MinIO.
    - Training metrics and configuration are logged and potentially saved as JSON to MinIO.

5.2.2. Model Deployment and Serving (`disease_prediction_service`)

- The `disease_prediction_service` loads the trained model components from MinIO upon startup (`models_loader.py`).
- This includes the main fusion model (`AttentionFusionMLP`), the image feature extractor (pre-trained ResNet50), the NIH data OneHotEncoder, and the MultiLabelBinarizer for interpreting outputs.
- The service then exposes an API endpoint (`/predict/`) to receive new patient data (image, NIH tabular, sensor data). It prepares features and corresponding attention masks for any potentially missing tabular or sensor data before feeding them to the model for prediction.

6. Prediction Workflow

1. A user interacts with the `frontend_service` (Gradio UI), providing an image file, NIH tabular data, and sensor data.
2. The `frontend_service` sends this data to the `api_gateway`.
3. The `api_gateway` routes the request to the `/predict/` endpoint of the `disease_prediction_service`.
4. The `disease_prediction_service`:
   - Receives the raw data.
   - Uses its loaded image feature extractor to get image features from the uploaded image.
   - Preprocesses the input NIH tabular data (encoding, scaling) using loaded artifacts.
   - Preprocesses the input sensor data (aggregation, scaling if applicable) using loaded artifacts or defined logic.
   - Concatenates/fuses these three sets of features.
   - Feeds the combined features into the loaded `AttentionFusionMLP` model.
   - Obtains raw output logits from the model.
   - Applies a sigmoid function to get probabilities for each disease class.
   - Optionally, applies a threshold (e.g., 0.5) to determine predicted labels.
   - Returns the list of diseases with their corresponding prediction probabilities (and/or binary predictions).
5. The `api_gateway` forwards the response back to the `frontend_service`.
6. The `frontend_service` displays the predictions to the user.

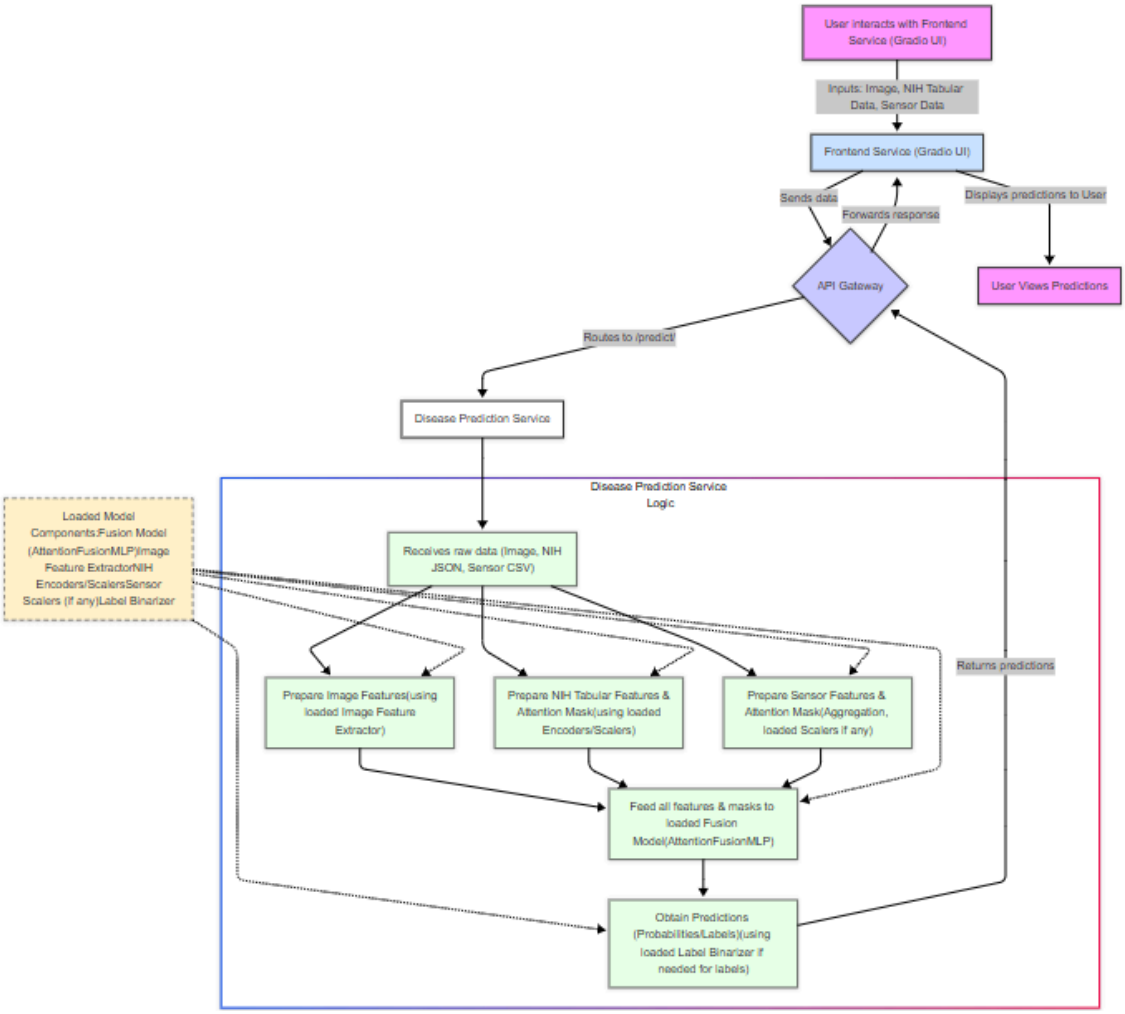

Figure 6: Prediction Workflow Diagram. Illustrates the sequence from user input in Gradio to the final prediction display.

7. Technology Stack Summary

The system leverages a modern, scalable technology stack:

- **Backend Framework:** FastAPI (Python) for all microservices requiring HTTP APIs.
- **Machine Learning Framework:** PyTorch for model definition, training, and inference. Torchvision for pre-trained image models and image transformations.
- **Data Handling & Manipulation:** Pandas, NumPy.
- **Data Preprocessing (ML):** Scikit-learn (for OneHotEncoder, StandardScaler, MultiLabelBinarizer).
- **Frontend UI:** Gradio (Python) for rapid web UI development.
- **Database (Structured Data):** PostgreSQL, accessed via SQLAlchemy ORM with Alembic for schema migrations.
- **Object Storage (Unstructured Data & Models):** MinIO.
- **Containerization & Orchestration:** Docker and Docker Compose.
- **API Gateway:** FastAPI (could be replaced by dedicated gateway solutions like Kong or Traefik in a production environment).
- **Asynchronous Task Handling:** FastAPI's `BackgroundTasks` for triggering preprocessing and model training.
- **HTTP Client:** `httpx` for inter-service communication.
- **Configuration Management:** `python-dotenv` for managing environment variables.
- **Logging:** Python's built-in `logging` module.

This stack was chosen to meet the project requirements, emphasizing Python-based solutions, robust API capabilities, efficient ML model handling, and standard operational practices with containerization.

8. Deployment

The system is designed for containerized deployment using Docker. The `docker-compose.yml` file defines and orchestrates all the microservices, including the MinIO object store and PostgreSQL database.

- Each microservice has its own Dockerfile (e.g., `api_gateway/Dockerfile`, `data_ingestion_service/Dockerfile`, etc. These are defined in their respective service directories and referenced in the `docker-compose.yml`).
- Environment variables are used extensively for configuration (e.g., service URLs, MinIO credentials, database connection strings), managed through `.env` files and the `docker-compose.yml` environments section.
- Data persistence for MinIO and PostgreSQL is handled using Docker volumes (`minio_data`, `postgres_data`) as defined in `docker-compose.yml`.
- Health checks are defined in `docker-compose.yml` for MinIO and PostgreSQL to ensure they are ready before dependent services start.

To deploy the system locally:

1. Ensure Docker and Docker Compose are installed.
2. Create a `.env` file based on `.env.example` if provided, or ensure variables are set.
3. Build the Docker images for each service using `docker-compose build`.
4. Start all services using `docker-compose up -d`.

9. Conclusion

This report has outlined the design of a microservice-based AI system for disease prediction. The system integrates data from diverse sources (images, tabular NIH data, sensor readings), preprocesses this data into suitable features, and utilizes a multi-modal fusion model with an attention mechanism for prediction. The architecture emphasizes modularity, scalability, and adherence to the specified technology stack.

The implementation demonstrates a comprehensive approach, from data ingestion and preprocessing to model training, deployment, and user interaction via a Gradio frontend. While currently set up for local Docker Compose deployment, the containerized nature of the services provides a solid foundation for scaling and migration to cloud environments.

Key strengths of the system include its multi-modal data fusion capability which leverages an attention mechanism to weigh data sources, the robust microservice architecture facilitating independent development and scaling, and the end-to-end automated pipeline covering data management, model lifecycle, and prediction serving.

Future enhancements could include more sophisticated attention mechanisms within the fusion model, integration of additional data modalities (e.g., genomic data, clinical notes), advanced techniques for handling missing data, and a more comprehensive MLOps pipeline for continuous training, deployment, and monitoring in a production setting. The system also provides a strong base for further research into explainable AI (XAI) techniques to understand the model's predictions.