Module Interface Specification for CXR

Team 27, Neuralyzers
Ayman Akhras
Nathan Luong
Patrick Zhou
Kelly Deng
Reza Jodeiri

January 17, 2025

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Symbols, Abbreviations and Acronyms

Symbol	Description
SRS	Software Requirements Specification
AI	Artificial Intelligence
CNN	Convolutional Neural Network
DICOM	Digital Imaging and Communications in Medicine
IVDDs	In Vitro Diagnostic Devices
ML	Machine Learning
PACS	Picture Archiving and Communication System
SaMD	Software as a Medical Device
ROC	Receiver Operating Characteristic Curve
SLA	Service-Level Agreement
FR	Functional Requirement
NFR	Non-Functional Requirement
FSM	Finite State Machine
CXR	Chest X-Ray Project
POC	Proof of Concept
TM	Theoretical Model
AWS	Amazon Web Services
ECS	Elastic Container Service
ECR	Elastic Container Registry

Contents

3 Introduction

The following document details the Module Interface Specifications for the application. Complementary documents include the Module Guide.
The full documentation and implementation can be found at CXR-Capstone.

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995). The mathematical notation follows standard conventions for sets, sequences, and functions. The notation includes symbols for ML model operations and medical data processing.

Data Type	**Notation**	**Description**
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)
tensor	$\mathbb{T}^n$	an n-dimensional array of numerical values used for ML computations
probability	$\mathbb{P}$	a real number in [0, 1] representing likelihood
matrix	$\mathbb{M}_{m,n}$	a 2D array of size m$\times$n containing numerical values
binary	$\mathbb{B}$	boolean values true, false

4.1 Data Types from Libraries

Data Type	**Notation**	**Description**
DICOM	DICOM	Digital Imaging and Communications in Medicine format, standard for medical imaging storage and transmission
PyTorch Tensor	torch.Tensor	Multi-dimensional matrix containing elements of a single data type, optimized for GPU operations
NumPy Array	np.ndarray	Multi-dimensional array for scientific computing and image processing
HTTP Request	HTTPRequest	Object containing HTTP request information including headers, body, and method
HTTP Response	HTTPResponse	Object containing HTTP response information including status code, headers, and body
DataFrame	pd.DataFrame	2-dimensional labeled data structure for patient and medical records
Base64	Base64	Binary-to-text encoding scheme for transmitting binary image data
YAML	YAML	Human-readable data serialization format for configuration files

Table 1: Data types from libraries

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	**Level 2**
Hardware-Hiding Module	Web Application
	HTTP Server
	Disease Prediction Server
	Disease Progression Server
Behaviour-Hiding Module	User Authentication
	Patients List View
	Patient Overview View
	Disease Progression View
	Medical Records List View
	X-Ray Report View
Software Decision Module	Disease Progression Model
	Disease Prediction Model
	Medical Record Management
	Data Persistent

Table 2: Module Hierarchy

6 Web Application Server Module

6.1 Module

Web Application Server

6.2 Other Modules the Current Module Uses

N/A

6.3 Syntax

6.3.1 Exported Constants

N/A

6.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
handleRequest	HTTP request	HTTP response	InvalidRequestException

6.4 Semantics

6.4.1 State Variables

- **sessionData**: Stores current session information.
- **activeUsers**: Keeps track of currently active users.

6.4.2 Environment Variables

- **serverPort**: The port on which the server listens for incoming connections.
- **hostAddress**: The server's host address.

6.4.3 Assumptions

- Assumes the HTTP Server Module (M2) is properly configured and running.
- Assumes valid HTTP requests are received.

6.4.4 Access Routine Semantics

handleRequest(request)

- transition: Processes the incoming HTTP request and routes it to the appropriate view module.
- output: Returns the HTTP response based on the request.

6.4.5 Local Functions

- **parseRequest(request)**: Parses the incoming HTTP request to extract necessary information.
- **generateResponse(data)**: Constructs an HTTP response based on the processed data.
- **authenticateUser(credentials)**: Verifies the user's credentials before processing the request.

7 HTTP Server Module

7.1 Other Modules the Current Module Uses

- User Authentication Module
- Disease Progression Module
- Medical Record Module

7.2 State Variables

- **CORS_setting**: holds Cross-Origin Resource Sharing settings to only allows web application server to access resources.
- **runtime.IdentityProvider**: A class that represents **User Authentication Module**
- **runtime.MedicalRecordService**: A class that represents **Medical Record Module**
- **runtime.ProgressionService**: A class that represents **Disease Progression Module**

7.3 Exported Constants and Access Programs

7.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`sign_up_user`	user_email: string, password: string, user_detail: UserDetail	user: User	`Unauthorized`, `UserAlreadyExists`, `InternalServer`
`sign_in_user`	user_email: string, password: string	bearer_token: string	`Unauthorized`, `IncorrectCredentials`, `InternalServer`
`get_self_user`	bearer_token: string	user: User	`Unauthorized`, `InternalServer`
`get_user_by_id`	uuid: string	user: User	`Unauthorized`, `UserNotFound`, `InternalServer`
`update_user_by_id`	uuid: string, updated_user_json: UserUpdateData	user: User	`Unauthorized`, `UserNotFound`, `InternalServer`

7.3.2 Exported Constants

- None

7.4 Environment Variables

- **FRONTEND_URL**: URL of front-end service, must be known at build time to enable CORS policy for front-end.

7.5 Assumptions

- None

7.6 Access Routine Semantics

7.6.1 heath_check()

- **Inputs**: None
- **Outputs**: 200 OK response or 500 InternalServerError
- **Preconditions**: The server can be at any states
- **Postconditions**: If the server returns 200 OK, all modules that this module, and it's child modules are working as expected.

7.6.2 sign_in(), sign_up(), get_user_by_id(), update_user_by_id()

- Existing routines from **User Authentication Module** that are made available for web application server to interact with via HTTP.

7.6.3 paginated_records_by_userId(), handle_record_by_id(), create_new_record(), paginated_prescriptions_by_recordId(), handle_prescription_by_id(), create_new_prescription()

- Existing routines from **Medical Record Module** that are made available for web application server to interact with via HTTP.

7.6.4 get_progression_report()

- Existing routine from **Disease Progression Module** that are made available for web application server to interact with via HTTP.

7.7 Local Functions

- **handle_exception(e)**: Return HTTP error as JSON objects when exceptions occurs, instead of default HTML page of Flask.

8 Disease Prediction Server Module

8.1 Other Modules the Current Module Uses

- M14: Data Persistence Module

8.2 State Variables

- **model**: The pre-trained model from `torchxrayvision` used for predicting lung diseases from X-ray images.
- **modelAccuracy**: Tracks the accuracy of the current model after training and validation.
- **predictionThreshold**: A constant threshold to determine the classification outcome (e.g., disease presence).
- **patientImageData**: Holds the chest X-ray image data used for prediction.

8.3 Exported Constants and Access Programs

8.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`predictDisease`	XRayImageURL	DiseasePrediction	`InvalidImageException`

8.3.2 Exported Constants

- **PREDICTION_THRESHOLD**: 0.75 (threshold for classification of disease presence)
- **MODEL_PATH**: Path to the pre-trained model (e.g., `./models/chest_xray_model.pth`)
- **MAX_PREDICTIONS**: 1000 (maximum number of predictions to handle concurrently)

8.4 Environment Variables

- **modelPath**: The path where the `torchxrayvision` pre-trained model is saved or loaded from.
- **predictionEndpoint**: The endpoint that this model is hosted on Docker host machine.

8.5 Assumptions

- Assumes the pre-trained `torchxrayvision` model is available and compatible with the data provided.
- Assumes valid X-ray image data is available for predictions.
- Assumes the Web Application Server Module (M1) and HTTP Server Module (M2) are properly configured and running.

8.6 Access Routine Semantics

8.6.1 predictDisease(XRayImageURL)

- **Transition**: Uses the loaded model to make predictions based on the provided X-ray image data.
- **Output**: Returns the disease prediction (e.g., probability of a disease being present) or throws an `InvalidImageException` if the image is invalid.

8.7 Local Functions

- **downloadXRay(imageURL)**: Download an X-ray image based on a given URL of the image.
- **loadModel()**: Loads the pre-trained model from disk or cloud storage using `torchxrayvision`'s functionality.
- **evaluateModel()**: Evaluates the model's performance with a test dataset to calculate metrics like accuracy and sensitivity.
- **preprocessImage()**: Preprocesses incoming X-ray image data to fit the model's input requirements (e.g., resizing, normalization).
- **postprocessPrediction()**: Processes the raw output from the model (e.g., probabilities) into a human-readable format (e.g., disease labels).

9 Disease Progression Server Module

9.1 Other Modules the Current Module Uses

- M14: Data Persistence Module

9.2 State Variables

- **model**: The pre-trained model used for tracking lung diseases from 2 X-ray images.
- **modelAccuracy**: Tracks the accuracy of the current model after training and validation.
- **predictionThreshold**: A constant threshold to determine the classification outcome (e.g., disease presence).
- **patientImagePair**: Pair of two x-ray images that the model use to make generate progression.

9.3 Exported Constants and Access Programs

9.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`generate_progression`	XrayImageURL1: string, XrayImageURL2: string	Progression: ProgressionData	`InternalServer`

9.3.2 Exported Constants

- None

9.4 Environment Variables

- **endpoint**: The endpoint that this model is hosted on Docker host machine.

9.5 Assumptions

- The model is pre-trained, changes in model architecture will need the server to be re-deployed.

9.6 Access Routine Semantics

9.6.1 trackProgression(XrayImageURL1, XrayImageURL2)

- **Transition**: Uses the existing model to calculate disease progression based on the provided X-ray images.
- **Output**: Returns the disease progression.

9.7 Local Functions

- **downloadXRay(imageURL)**: Download an X-ray image based on a given URL of the image.
- **evaluateModel()**: Evaluates the model's performance with a test dataset to calculate metrics like accuracy and sensitivity.
- **preprocessImage()**: Preprocesses incoming X-ray image data to fit the model's input requirements (e.g., resizing, normalization).
- **postprocessPrediction()**: Processes the raw output from the model (e.g., probabilities) into a hJSON.

10 User Authentication Module

10.1 Other Modules the Current Module Uses

- M14: Data Persistence Module

10.2 State Variables

- **cognito_idp_client**: AWS Cognito client, initialized via the AWS boto3 library.
- **user_pool_id**: Identifier of the database at which AWS Cognito stores user's information such as username, password, first name, etc.
- **client_id**: Identifier of the User Authentication Module to authenticate with AWS Cognito (via OAUTH's client credential grant).
- **client_secret**: Password of the User Authentication Module, used in conjunction with client_id to authenticate with AWS Cognito.

10.3 Exported Constants and Access Programs

10.3.1 Exported Access Programs

Name	In	Out	Exceptions
`sign_up_user`	user_email, password, user_detail	user	`Unauthorized`, `UserAlreadyExists`, `InternalServerError`
`sign_in_user`	user_email, password	bearer_token	`Unauthorized`, `IncorrectCredentials`, `InternalServerError`
`get_self_user`	bearer_token	user	`Unauthorized`, `InternalServerError`
`get_user_by_id`	uuid	user	`Unauthorized`, `UserNotFoundException`, `InternalServerError`
`update_user_by_id`	uuid, updated_user_json	user	`Unauthorized`, `UserNotFoundException`, `InternalServerError`

10.3.2 Exported Constants

This module does not export any constants

10.4 Environment Variables

- **AWS_ACCESS_KEY_ID**: Access key to authenticate backend with AWS
- **AWS_SECRET_ACCESS_KEY**: Secret key to authenticate backend with AWS
- **AWS_REGION**: AWS Region of the backend infrastructure

10.5 Assumptions

- AWS Cognito returns consistent json results with user fields, since neuralanalyzer backend expect certain fields from user object (custom:firstName, custom:organization, etc.)
- AWS Cognito always stays online, since without this service, users wont be able to login into the application.

10.6 Access Routine Semantics

10.6.1 sign_up_user(user_email, password, user_detail)

- **Inputs**:
  - **user_email**: Email address of the user.
  - **password**: Password for the user's account.
  - **user_detail**: Additional user details, such as first name and organization, formatted as Python Dictionary.
- **Outputs**: A user object containing the created user's details.
- **Preconditions**:
  - The provided email is valid and not already registered.
  - Password meets AWS Cognito's security requirements.
- **Postconditions**:
  - The user is successfully created and stored in the Cognito user pool.

10.6.2 sign_in_user(user_email, password)

- **Inputs**:
  - **user_email**: Email address of the user.
  - **password**: Password for the user's account.
- **Outputs**: A bearer token for authenticated access to other services.
- **Preconditions**:
  - The user is registered in the Cognito user pool.
  - The provided credentials are correct.
- **Postconditions**:
  - The user is successfully authenticated.
  - A bearer token is generated and returned for future authenticated requests.

10.6.3 get_self_user(bearer_token)

- **Inputs**:
  - **bearer_token**: User's Bearer token.
- **Outputs**: A user object containing the created user's details.
- **Preconditions**:
  - The bearer_token must be valid and not expired.
- **Postconditions**:
  - The user is successfully retrieved from AWS Cognito.

10.6.4 get_user_by_id(uuid)

- **Inputs**:
  - **uuid**: The unique identifier of the user (string)
- **Outputs**: A user object containing the user's details.
- **Preconditions**:
  - User's UUID must be valid, and attached to a registered user.
- **Postconditions**:
  - The user is successfully retrieved from AWS Cognito.

10.6.5 update_user_by_id(uuid, updated_user_json)

- **Inputs**:
  - **uuid**: The unique identifier of the user (string)
  - **updated_user_json**: Python Dictionary object that contains the field which the caller wants to update the user with.
- **Outputs**: A user object containing the updated user's details.
- **Preconditions**:
  - User's UUID must be valid, and attached to a registered user.
  - User's updated fields must belong to list of fields users are allowed to edit (ie: first and last name, role, organization, etc.)
- **Postconditions**:

  - User with UUID must have the fields in updated_user_json updated on AWS Cognito.

10.7 Local Functions

- **_parse_user_json_from_cognito_user**: Convert user object returned from AWS Cognito, to a more application-friendly internal user object.
- **_parse_cognito_attr_user_from_user_json**: Convert the internal user object to the user data format that AWS Cognito requires.
- **_get_username_from_email**: Converting user's email into their username. (ie: nathan@gmail.com will be converted to nathan_gmail)
- **_get_user_by_email**: Fetching user from AWS Cognito via their email.

11 Patient List View Module

11.1 Other Modules the Current Module Uses

- M1: Web Application Server Module

11.2 State Variables

- **patientList**: Stores a list of patients assigned to a doctor, including their basic information such as name, age, and medical condition.
- **searchFilters**: Stores the current filters applied to the patient list for sorting and searching purposes.

11.3 Exported Constants and Access Programs

11.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`viewPatientList`	Doctor ID, filters	List of patient details	`PatientListNotFoundException`
`applyFilter`	Filter parameters	Filtered patient list	`InvalidFilterException`
`sortList`	Sorting criteria	Sorted patient list	None

11.3.2 Exported Constants

- **MAX_PATIENTS_PER_PAGE**: 20 (maximum number of patients displayed per page in the list)

- **DEFAULT_SORT_ORDER**: "alphabetical" (default order in which patients are listed)

11.4 Environment Variables

- **patientDataPath**: Path to the data source containing patient records.
- **authenticationService**: URL or endpoint of the service used for user authentication.

11.5 Assumptions

- Assumes that the User Authentication Module (M5) is responsible for verifying that the user is a doctor with access to the list.
- Assumes the patient data is up-to-date and synchronized with the Data Persistence Module (M15).

11.6 Access Routine Semantics

11.6.1 viewPatientList(doctorID, filters)

- **Transition**: Retrieves a list of patients associated with the given doctor, applying the specified filters.
- **Output**: Returns the list of patient details or throws a `PatientListNotFoundException` if no patients are found.

11.6.2 applyFilter(filters)

- **Transition**: Applies the given filters to the current patient list.
- **Output**: Returns the filtered patient list or throws an `InvalidFilterException` if the filters are invalid.

11.6.3 sortList(criteria)

- **Transition**: Sorts the current patient list based on the provided criteria.
- **Output**: Returns the sorted patient list.

11.7 Local Functions

- **filterPatients(filters)**: Filters the patient list according to the specified parameters.
- **sortPatients(criteria)**: Sorts the patient list based on the provided criteria.

- **logListAccess(doctorID)**: Logs each time a doctor accesses the patient list for auditing purposes.

12 Patient Overview Module

12.1 Other Modules the Current Module Uses

- M6: Patient List View Module

12.2 State Variables

- **patientOverview**: Stores the summary information of a selected patient, including personal details, recent medical history, and current treatment plans.

12.3 Exported Constants and Access Programs

12.3.1 Exported Access Programs

Name	In	Out	Exceptions
`getPatient`	Patient ID	Patient overview details	`OverviewNotFoundException`

12.3.2 Exported Constants

- **OVERVIEW_REFRESH_INTERVAL**: 15 minutes (interval for refreshing the patient overview data)

12.4 Environment Variables

- **overviewDataPath**: Path to the data source containing patient overview information.
- **authenticationService**: URL or endpoint of the service used for user authentication.

12.5 Assumptions

- Assumes the User Authentication Module (M5) ensures that only authorized users can view the patient overview.
- Assumes the overview data is accurate and synchronized with the Data Persistence Module (M15).

12.6 Access Routine Semantics

12.6.1 viewPatientOverview(patientID)

- **Transition**: Retrieves the overview information of the specified patient.
- **Output**: Returns the patient overview details or throws an `OverviewNotFoundException` if the overview cannot be found.

12.7 Local Functions

- **fetchOverviewData(patientID)**: Retrieves overview data from the data source.
- **generateOverviewSummary(patientID)**: Creates a summary of the patient's current status.
- **logOverviewAccess(patientID)**: Logs each time a patient overview is accessed for auditing purposes.

13 Diseases Progression View

13.1 Other Modules the Current Module Uses

- M6: Patient List View Module

13.2 State Variables

- **Title**: fill this

13.3 Exported Constants and Access Programs

13.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`viewProgression`	Patient ID	Progression data	`ProgressionNotFoundException`

13.3.2 Exported Constants

- **PROGRESSION_UPDATE_INTERVAL**: 12 hours (time interval for updating disease progression data)
- **DEFAULT_CHART_TEMPLATE**: "default_progression_chart.html" (default template for displaying disease progression charts)

13.4 Environment Variables

- **progressionDataPath**: Path to the data source containing disease progression information.
- **chartRenderingService**: URL or endpoint of the service used for rendering progression charts.

13.5 Assumptions

- Assumes the User Authentication Module (M11) ensures that only authorized users can view the disease progression.
- Assumes the data is regularly updated and maintained in the Data Persistence Module (M15).

13.6 Access Routine Semantics

13.6.1 viewDiseaseProgression(patientID)

- **Transition**: Retrieves the disease progression details for the specified patient.
- **Output**: Returns the progression details or throws a `ProgressionNotFoundException` if the data cannot be found.

13.7 Local Functions

- **fetchProgressionData(patientID)**: Retrieves disease progression data from the data source.
- **generateProgressionChart(patientID)**: Generates a visual chart of the disease progression.
- **logProgressionAccess(patientID)**: Logs access to a patient's disease progression data for auditing purposes.

14 Medical Records List View Module

14.1 Other Modules the Current Module Uses

- M6: Patient List View Module

14.2 State Variables

- **medicalRecordsList**: Contains a list of all medical records associated with a specific patient, including dates, record types, and summaries.

14.3 Exported Constants and Access Programs

14.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`viewMedicalRecordsList`	Patient ID	List of medical records	`RecordsNotFoundException`

14.3.2 Exported Constants

- **RECORDS_REFRESH_INTERVAL**: 10 minutes (interval for refreshing the list of medical records)
- **DEFAULT_LIST_TEMPLATE**: "default_records_list_template.html" (default template for displaying the list of medical records)

14.4 Environment Variables

- **recordsDataPath**: Path to the data source containing the patient's medical records.
- **recordSummaryService**: URL or endpoint of the service used for summarizing medical records.

14.5 Assumptions

- Assumes the User Authentication Module (M5) ensures that only authorized users can view the medical records.
- Assumes the records data is accurate and up-to-date in the Data Persistence Module (M15).

14.6 Access Routine Semantics

14.6.1 viewMedicalRecordsList(patientID)

- **Transition**: Retrieves the list of medical records for the specified patient.
- **Output**: Returns the list of medical records or throws a `RecordsNotFoundException` if no records are found.

14.7 Local Functions

- **fetchMedicalRecords(patientID)**: Retrieves the list of medical records from the data source.
- **generateRecordsSummary(patientID)**: Creates summaries for each record in the list.

- **logRecordsAccess(patientID)**: Logs access to a patient's medical records list for auditing purposes.

15 X-Ray Report View

15.1 place holder

- M6: Patient List View Module

15.2 State Variables

- **place holder**: place holder

15.3 Exported Constants and Access Programs

15.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`place holder`	place holder	place holder	`place holder`

15.3.2 Exported Constants

- **place holder**: "place holder"

15.4 Environment Variables

- **place holder**: place holder

15.5 Assumptions

- place holder
- place holder

15.6 Access Routine Semantics

15.6.1 place holder(place holder)

- **place holder**: place holder

15.7 Local Functions

- **place holder(place holder)**: place holder

16 Disease Progression Module

16.1 Other Modules the Current Module Uses

- M4: Disease Progression Server Module
- M13: Medical Record Module

16.2 State Variables

None

16.3 Exported Constants and Access Programs

16.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`get_progression`	XrayImageURL1: string, XrayImageURL2: string	Progression Result	`InternalServerError`
`get_latest_progression_by_userId`	userId: string	Progression Result	`UserNotFoundException`, `InternalServerError`

16.3.2 Exported Constants

- None

16.4 Environment Variables

- **PROGRESSION_SERVER_URL**: URL of **Disease Progression Server Module**

16.5 Assumptions

- The Progression Server Module has high availability.
- The Progression Server Module expose a single endpoint of `/progression`

16.6 Access Routine Semantics

16.6.1 get_progression(XrayImageURL1, XrayImageURL2)

- **Transition**: Make an HTTP call to the **Disease Progression Server Module**
- **Output**: Returns a `ProgressionResult` containing the disease progression information (Disease gets worsen or better).

16.6.2 get_latest_progression_by_userId(userId)

- **Transition**: Gets the latest 2 records from user, via function call to **Medical Record Module**. With the two latest record, the module will extract the 2 X-Ray URls and calls `get_progression`
- **Exception**: Throws `InvalidStageException` if `stageData` is invalid.

16.7 Local Functions

- **executeHttpRequest(method, path)**: Send a generic HTTP request to a specified path

17 Disease Prediction Module

17.1 Other Modules the Current Module Uses

- M5: User Authentication Module
- M3: Disease Prediction Server Module

17.2 State Variables

- None

17.3 Exported Constants and Access Programs

17.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`predictDisease`	patientData	PredictionResult	`InternalServerError`

17.3.2 Exported Constants

- None

17.4 Environment Variables

- **PredictionServerURL**: URL of **Prediction Server Module**

17.5 Assumptions

- The Prediction Server Module has high availability.
- The Prediction Server Module expose a single endpoint of `/predict`

17.6 Access Routine Semantics

17.6.1 loadModel(modelPath)

- **Transition**: Loads the disease prediction model from the specified `modelPath`.
- **Exception**: Throws `ModelLoadException` if the model cannot be loaded.

17.6.2 predictDisease(patientData)

- **Inputs**: `patientData` is an dictionary contains user information (name, age, sex, etc.) and the url to X-ray image.
- **Output**: Returns a `PredictionResult` containing the predicted disease information.
- **Transition**: Processes `patientData` using the current prediction model to generate a prediction.
- **Exception**: Throws `InvalidInputException` if `patientData` is malformed or incomplete.

17.7 Local Functions

- **executeHttpRequest(method, path)**: Send a generic HTTP request to a specified path

18 Medical Record Module

18.1 Other Modules the Current Module Uses

- M12: Disease Prediction Module
- M14: Data Persistence Module

18.2 State Variables

- **place holder**: place holder

18.3 Exported Constants and Access Programs

18.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`get_paginated_ prescription_ by_ recordId`	recordId: string, limit: int	prescriptions: [Prescription]	`RecordNotFound`, `InternalServer`
`get_prescription_ by_id`	uuid: string	prescription: Prescription	`PrescriptionNotFound`, `InternalServer`
`create_new_ prescription`	recordId: string, prescription: Prescription	prescription: Prescription	`RecordNotFound`, `InternalServer`
`update_ prescription_ by_id`	uuid: string, prescription: Prescription	prescription: Prescription	`PrescriptionNotFound`, `InternalServer`
`get_paginated_ record_by_ userId`	userId: string, limit: int	records: [Record]	`UserNotFound`, `InternalServer`
`get_record_ by_id`	uuid: string	record: Record	`RecordNotFound`, `InternalServer`
`create_new_ record`	userId: string, record: Record	record: Record	`InternalServer`
`update_record_ by_id`	uuid: string, record: Record	record: Record	`RecordNotFound`, `InternalServer`

18.3.2 Exported Constants

- None

18.4 Environment Variables

- None

18.5 Assumptions

- The underlying Data Persistence Module persist data into storage on AWS

- The underlying Data Persistence Module stays highly available and bug prone.

18.6 Access Routine Semantics

18.6.1 get_paginated_prescription_by_recordId(recordId, limit)

- **Inputs**:
  - **recordId**: Unique identifier of the medical record (string).
  - **limit**: Maximum number of prescriptions to retrieve (integer).
- **Outputs**: A list of prescription objects associated with the provided `recordId`, limited by the `limit` parameter.

18.6.2 get_prescription_by_id(uuid)

- **Inputs**: **uuid**: Unique identifier of the prescription (string).
- **Outputs**: The prescription object corresponding to the given `uuid`.

18.6.3 create_new_prescription(recordId, prescription)

- **Inputs**:
  - **recordId**: The unique identifier of the medical record to associate with the prescription (string).
  - **prescription**: The prescription object (dictionary) containing the prescription details.
- **Outputs**: The created prescription object

18.6.4 update_prescription_by_id(uuid, prescription)

- **Inputs**:
  - **uuid**: The unique identifier of the prescription to be updated (string).
  - **prescription**: The updated prescription object (dictionary).
- **Outputs**: The updated prescription object

18.6.5 get_paginated_record_by_userId(userId, limit)

- **Inputs**:
  - **userId**: The unique identifier of the user whose records are being requested (string).
  - **limit**: The maximum number of records to retrieve (integer).
- **Outputs**: A list of records associated with the provided `userId`, limited by the `limit` parameter.

18.6.6 get_record_by_id(uuid)

- **Inputs**: **uuid**: The unique identifier of the record to retrieve (string).
- **Outputs**: The record object corresponding to the given `uuid`.

18.6.7 create_new_record(userId, record)

- **Inputs**:
  - **userId**: The unique identifier of the user associated with the record (string).
  - **record**: The record object (dictionary) containing the medical record details.
- **Outputs**: The created record object.

18.6.8 update_record_by_id(uuid, record)

- **Inputs**:
  - **uuid**: The unique identifier of the record to update (string).
  - **record**: The updated record object (dictionary).
- **Outputs**: The updated record object.

18.7 Local Functions

- None

19 Data Persistent Module

19.1 Other Modules the Current Module Uses

- None

19.2 State Variables

- **dynamodb**: AWS DynamoDB client (managed NoSQL service), initialized via aws boto3 library.
- **table**: Name of the DynamoDB table.
- **s3_client**: AWS S3 client (managed blob storage service), initialized via aws boto3 library.

19.3 Exported Constants and Access Programs

19.3.1 Exported Access Programs

Name	In	Out	Exceptions
`get_item_by_id`	uuid	item	`InternalServerError, ClientError`
`create_new_item`	item	uuid, item	`InternalServerError, ClientError`
`update_item_by_id`	uuid, item	item	`InternalServerError, ClientError`
`get_s3_upload_presigned_url`	file_name	presigned_url	`InternalServerError, ClientError`

19.3.2 Exported Constants

This module does not export any constants

19.4 Environment Variables

- **AWS_ACCESS_KEY_ID**: Access key to authenticate backend with AWS
- **AWS_SECRET_ACCESS_KEY**: Secret key to authenticate backend with AWS
- **AWS_REGION**: AWS Region of the backend infrastructure

19.5 Assumptions

- AWS DynamoDB and AWS S3 returns consistent json results and error codes.
- AWS DynamoDB and AWS S3 always stays online, without these services, users wont be able to create, view, update prescriptions and records.

19.6 Access Routine Semantics

19.6.1 get_item_by_id(uuid)

- **Inputs**:

  - **uuid**: Unique identifier of the item to retrieve.
- **Outputs**:
  - The item object corresponding to the given ‘uuid‘.
- **Preconditions**:
  - The ‘uuid‘ exists in the DynamoDB table.
- **Postconditions**:
  - The item object is successfully retrieved from the database.

19.6.2 create_new_item(item)

- **Inputs**:
  - **item**: A Python Dictionary object containing the details of the item to be created.
- **Outputs**:
  - **uuid**: The unique identifier of the created item.
  - **item**: The created item object.
- **Preconditions**:
  - The ‘item‘ dictionary is well-formed.
- **Postconditions**:
  - The new item is successfully added to the DynamoDB table.

19.6.3 update_item_by_id(uuid, item)

- **Inputs**:
  - **uuid**: The unique identifier of the item to be updated.
  - **item**: A Python Dictionary containing the updated fields for the item.
- **Outputs**:
  - The updated item object.
- **Preconditions**:
  - The ‘uuid‘ corresponds to an existing item in the DynamoDB table.
  - The ‘item‘ Dictionary contains valid fields for the update.
- **Postconditions**:
  - The item is successfully updated in the database, and returned to user.

19.6.4 get_s3_upload_presigned_url(file_name)

- **Inputs**:
  - **file_name**:The name of the file to be uploaded to S3 (string).
- **Outputs**:
  - **presigned_url**: A pre-signed URL allowing authorized upload of the specified file to AWS S3.
- **Preconditions**:
  - The 'file_name' is a valid, non-empty string
  - The S3 bucket configured in the s3_client is accessible and writable by the authenticated user.
- **Postconditions**:
  - A valid pre-signed URL for uploading the specified file is generated and returned.
  - The generated pre-signed URL is only valid for 1 minute.

19.7 Local Functions

- None

References

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach.* International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

1. What went well while writing this deliverable?

   Our existing documentation, such as the detailed list of functional and non-functional requirements and the Module Interface Specification (MIS) we created, provided a solid roadmap for how the system should behave. This clarity made it easier to develop the design blueprint and the specifications for each module because we already knew what each part needed to do and how they should interact. Another big plus was the team's early focus on a modular architecture, such as separating the machine learning (ML) services from the core backend. This approach kept our design documents organized and manageable, allowing us to develop, test, and maintain each module independently without getting overwhelmed. Additionally, having the core implementation set up during the Proof of Concept (POC) phase gave us a great direction for developing the specifications for our modules. It provided a tested foundation to build on, reducing uncertainties and making our specification development more precise and targeted.

2. What pain points did you experience during this deliverable, and how did you resolve them?

   We faced notable challenges while designing our Behavior Hiding Module, especially given our limited experience with standard healthcare user interface practices. To address this, we created an initial MVP concept UI and then conducted interviews with several healthcare professionals. Their feedback highlighted the need for straightforward interactions with patient profiles, a smooth process to upload chest X-ray scans, and a hassle-free way to create new records. In response, we iterated on our designs multiple times, adjusting layout structures, refining navigation flows, and simplifying labeling conventions to ensure the module felt intuitive and clinically relevant. Each revision brought us closer to delivering a user-centric interface that balances simplicity with the technical rigor necessary in medical settings,resulting in a solution that healthcare professionals can adopt with minimal friction.

3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?

   In our previous meeting with our supervisor, Dr. Moradi, we decided to prioritize designing a user-friendly interface specifically tailored for the use of radiologists. Based on their feedback, we focused on creating an intuitive and easy-to-navigate interface. Most of our time and research was dedicated to understanding and adhering to standard healthcare web interface guidelines to ensure the design aligns with the requirements of a healthcare setting. We decided to prioritize interpretability and transparency of AI-generated report (explainable AI) which comes from the discussions with stakeholder emphasizing trust in AI results. This would help doctors/radiologists to better understand the prediction results. As for non-client-informed decisions, we decided to implement a disease progression feature that is separated from the prediction model. It traces the patient disease progression over a certain period, allowing comparing different X-rays and reports to give a more straightforward understanding of patient condition overtime while also reviewing the effectiveness of the prescriptions given to the patient in that specifc period. By separating two models, they work independently and will still function if the other one goes down, allowing higher fault tolerance.

4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?

   In the process of drafting the design document, we identified the need to revise several aspects of our other documents, including the Software Requirements Specification (SRS) and the Hazard Analysis document, to align with newly introduced features and interface design changes. One of the key additions was the disease progression monitoring feature, implemented using a Detection Transformer (DETR). This feature required us to add new requirements in the SRS, detailing how disease progression should be visualized, the data flow for tracking multiple X-rays over time, and the integration of DETR-based models for reliable prediction. Furthermore, the design now incorporates processing DICOM metadata to extract relevant information, such as clinical history, imaging parameters, and patient demographics, which ensures that critical contextual data is seamlessly integrated into the system. This step enhances the accuracy and usability of the interface by providing doctors with a comprehensive view of patient history alongside imaging results. Additionally, we refined our interface design to primarily cater to doctors instead of only patients, shifting the focus to provide structured, clinically relevant data and user-friendly tools for interpreting complex results. This change necessitated updating user interface requirements in the SRS to include features such as advanced filtering, annotation tools, and report generation. Furthermore, the incorporation of these features introduced new safety and ethical considerations, leading us to revisit and expand the Hazard Analysis document. For example, we identified potential risks related to false positives or negatives in disease progression detection, which could impact clinical decision-making. To mitigate these risks, we documented fallback mechanisms such as manual review workflows and alerts for ambiguous cases.

5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)

   A significant limitation of our solution lies in the inherent bias and fairness concerns within our AI model. Training exclusively on the MIMIC-CXR dataset from Beth

Israel Deaconess Medical Center means our model may exhibit performance disparities across different patient demographics. The limited diversity in our training data regarding age, gender, ethnicity, and socioeconomic status could result in reduced prediction accuracy for underrepresented populations, potentially compromising our system's ability to provide equitable healthcare solutions. Furthermore, since our training data originates from a single institution's imaging equipment, the model may demonstrate reduced effectiveness when processing X-rays captured using different machines, resolutions, or configurations. This technical limitation could disproportionately affect healthcare facilities utilizing different or older equipment specifications. Given unlimited resources, we would prioritize expanding our training dataset to encompass a broader spectrum of patient populations from multiple healthcare institutions and diverse imaging equipment. We would implement comprehensive testing protocols to evaluate model performance across various demographic groups and establish robust monitoring systems to track performance metrics across different population segments and equipment configurations. While our current implementation demonstrates promising results, addressing these limitations would be essential for future iterations to ensure the system delivers consistent and equitable care across all patient populations and healthcare settings.

6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)

   We considered both MVC and a layered architecture for our medical imaging system, particularly to accommodate disease prediction and progression pipelines. While MVC separates data, business logic, and presentation, it can introduce tight coupling between controllers and views, ultimately complicating upgrades and integrations of AI components. In contrast, a layered approach naturally decouples these processes into discrete layers such as presentation, application, and data—ensuring each layer can be maintained, replaced, or updated independently. This flexibility is especially vital given our need to integrate complex machine learning and imaging algorithms that may evolve or expand over time. Additionally, layering simplifies the implementation of security, logging, and regulatory compliance by localizing those concerns within specific strata of the system. By providing a clear, top-to-bottom sequence for data flow, the layered model also reduces communication complexity, making it easier to integrate and manage our disease prediction and progression modules. Ultimately, we selected layered architecture for its modularity, maintainability, and scalability.