Module Interface Specification for Scanalyze AI

Team 16, Ace
Harrison Chiu
Hamza Issa
Ahmad Hamadi
Jared Paul
Gurnoor Bal

April 4, 2025

1 Revision History

Date	Version	Notes
01/07/2024	0.0	Initial Document.
01/08/2024	0.1	Added MIS structure and formatting; Started adding module definitions.
01/10/2024	0.2	Included Table from Module Guide (MG) into the Module Decomposition section.
01/12/2024	0.3	Expanded Symbols, Abbreviations, and Acronyms section; refined MIS formatting.
01/14/2024	0.4	Completed Introduction and Notation sections; refined function signatures in MIS.
01/16/2024	0.5	Updated MIS for HardwareAcceleration and DatasetHandler modules based on feedback and finished all of the other modules.
2025/03/28	0.6	Updated MIS to reflect our change in project from Diffusion Models to CNN Multi-disease detection model

Table 1: Revision History

2 Summary of Changes Made to the MIS

- The Introduction section was rewritten to reflect the updated project focus, shifting from a generative diffusion model to a CNN-based multi-disease classification model.
- The Module Decomposition and Module Descriptions were revised to accurately align with the current structure and logic found in the codebase.
- The Reflection Questions were updated to reflect the latest design decisions, module responsibilities, and overall development work carried out for the MIS.

3 Symbols, Abbreviations and Acronyms

This section records the symbols, abbreviations, and acronyms information for easy reference for terms used in this document.

For information on most of the symbols, abbreviations, and acronyms referenced in this document, see the SRS Documentation at the following link:

GitHub SRS Documentation

The information on the rest of the symbols, abbreviations, and acronyms referenced in this document are shown in the table below.

symbol	**description**
AI/ML	Artificial Intelligence/Machine Learning
DICOM	Digital Imaging and Communications in Medicine; technical standard for digital storage/transmission of medical images and related information
GUI	Graphical User Interface
JPEG/JPG	Joint Photographic Experts Group; digital image compression standard, image format
M	Module
MG	Module Guide
MVC	Model-View-Controller Software Architecture
NLP	Natural Language Processing
SRS	Software Requirements Specification
Scanalyze AI	The Process of Designing and Developing Software; a reference to the software application described in this document

Contents

4 Introduction

This document outlines the Module Interface Specifications for Scanalyze, a web-based chest X-ray classification application. The software leverages a custom-tuned multi-class classification machine learning (ML) algorithm to analyze chest X-ray (CXR) images and produce a probability distribution over a range of potential chest-related diseases. By identifying the most likely conditions present in a given CXR image, the system supports radiologists, researchers, and medical institutions in streamlining diagnostic workflows and reducing report backlogs.

The application is built on an MVC-backend architecture, integrating components such as user credential management, repository communication, encryption protocols, and a dedicated model backend. This modular structure ensures security, scalability, and maintainability while supporting seamless integration of classification and user-facing features.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at:

GitHub Repository.

5 Notation

The structure of the MIS for modules comes from [2], with the addition that template modules have been adapted from [1]. The mathematical notation comes from Chapter 3 of [2]. For instance, the symbol := is used for a multiple assignment statement, and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by Chest Scan.

Data Type	**Notation**	**Description**
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in $(-\infty, \infty)$
natural number	$\mathbb{N}$	a number without a fractional component in $[1, \infty)$
real number	$\mathbb{R}$	any number in $(-\infty, \infty)$

The specification of Scanalyze uses some derived data types, such as sequences, strings, and tuples. Sequences are lists containing elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types.

Additionally, Scanalyze defines functions, where inputs and outputs are described by their data types. Local functions are defined using type signatures followed by their specifications.

6 Module Decomposition

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

Level 1	**Level 2**
Hardware-Hiding	N/A
Behaviour-Hiding	ModelInterface AuthClient DataRetrieval DataPreparation Authorization
Software Decision	Config MLBackend ModelArchitecture Training

Table 2: Module Hierarchy

7 MIS of ModelInterface

7.1 Module

ModelInterface
This module provides the frontend interface for uploading medical images (e.g., chest X-rays), sending them to a backend model, and visualizing the results. It facilitates zoomable image previews, prediction display, confidence-based filtering, and result export as a PDF.

7.2 Uses

- Enables users to interact with machine learning models via image upload, analysis, and result exploration.
- Provides an intuitive interface using modern UI features (React).
- Integrates with backend APIs for real-time inference and result retrieval.

7.3 Syntax

7.3.1 Exported Constants

None.

7.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
uploadImage	imageFile	imagePreview	FileTypeError
analyzeImage	uploadedImage	predictionResults	APIError
downloadReport	analysisResults	pdfFile	PDFGenerationError

7.4 Semantics

7.4.1 State Variables

- **selectedFile**: Stores the uploaded image file.
- **result**: Object holding prediction results and generated report.
- **confidenceThreshold**: Numeric slider value for filtering predictions.

7.4.2 Environment Variables

- None.

7.4.3 Assumptions

- Image files are in a supported format (e.g., JPEG, PNG).
- Backend API returns a valid JSON response.
- Internet access is available for cloud-based predictions.

7.4.4 Access Routine Semantics

uploadImage(file: Image):

- **Transition:** Accepts an image file, verifies its file type, and updates state for preview.
- **Output:** An image preview rendered in the interface.
- **Exceptions:** Throws **FileTypeError** if rendering fails.

analyzeImage(image: File):

- **Transition:** Sends the image to the backend using a POST request. Receives prediction labels and probabilities.
- **Output:** Renders a diagnostic report that contains prediction results and deeper insights.
- **Exceptions:** Throws **APIError** if the response is malformed or request fails.

downloadReport(analysis: object):

- **Transition:** Converts visible diagnostic report results in the UI into a downloadable PDF.
- **Output:** A PDF file downloaded via browser.
- **Exceptions:** Throws **PDFGenerationError** if rendering or download fails.

8 MIS of AuthClient

8.1 Module

AuthClient
This frontend module handles user authentication flows, including registration and login UI logic, form state management, and API interaction. It is responsible for managing input fields, UI feedback (success or error messages), and page navigation after successful login or registration. While it communicates with backend endpoints, **the core responsibility of this module is limited to the frontend layer**.

8.2 Uses

- Renders registration and login forms for users to input credentials.
- Manages local input state: username, email, password, and feedback messages.
- Handles form submission and calls backend API via a reusable API service.
- Displays real-time feedback for success and error states using frontend validation and API responses.
- Performs routing/navigation after successful login or registration using React Router.
- Provides a secure and intuitive interface for authentication-related user flows.

8.3 Syntax

8.3.1 Exported Constants

None.

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
registerUser	username, email, password	successMessage	RegistrationError
loginUser	username, password	successMessage	AuthenticationError
redirectOnLogin	success	homepage	NavigationError

8.4 Semantics

8.4.1 State Variables

- **username**: Bound to registration form input.
- **email**: Used in registration forms.
- **password**: Captures user password securely.

8.4.2 Environment Variables

- None.

8.4.3 Assumptions

- Form fields are validated on the frontend prior to submission.
- Network is reachable when API calls are triggered.

8.4.4 Access Routine Semantics

registerUser(username: string, email: string, password: string):

- **Transition:** Collects and validates form data, submits it to the API service. On success, updates UI with a success message and redirects to login.
- **Output:** Success message: "Registration successful! Please log in."
- **Exceptions:** Displays a frontend **RegistrationError** if the backend returns a conflict.

loginUser(username: string, password: string):

- **Transition:** Sends login requests through the API service. On success, clears form state and navigates the user to the homepage or dashboard.
- **Output:** Success message: "Login Successful."
- **Exceptions:** Displays **AuthenticationError** if credentials are invalid.

redirectOnLogin(success: boolean):

- **Transition:** If login succeeds, it triggers navigation to the homepage.
- **Output:** Homepage is rendered on the UI.
- **Exceptions:** Throws **NavigationError** (404 error) if routing is misconfigured or fails.

9 MIS of DataRetrieval

9.1 Module

DataRetrieval
Handles downloading and extracting NIH Chest X-ray image archives. Ensures local storage of required .png files for downstream training and validation.

9.2 Uses

- Automates retrieval of 12 archive files from publicly hosted NIH Box links.
- Extracts and stores images in a structured local directory (`./data/images`).
- Prevents redundant downloads by checking file existence.

9.3 Syntax

9.3.1 Exported Constants

None.

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
prepareData	None	imageFiles in /data/images	NetworkError, FileIOError

9.4 Semantics

9.4.1 State Variables

- **downloadDir**: Stores .tar.gz files.
- **imagesDir**: Directory for extracted .png files.

9.4.2 Environment Variables

- Internet connection.
- Write permissions to local file system.
- NIH archive links (hardcoded).

9.4.3 Assumptions

- Links are valid and publicly accessible.
- Directory structure is consistent.

9.4.4 Access Routine Semantics

prepareData():

- **Transition:** Creates download and extraction directories if they do not exist. Downloads NIH image archives unless they are already cached. Extracts .png files into the target image directory.
- **Output:** All NIH .png chest X-ray images are saved into `./data/images`.
- **Exceptions:** Raises **NetworkError** if internet access fails, and **FileIOError** if extraction paths are inaccessible or write fails.

10 MIS of DataPreparation

10.1 Module

DataPreparation

This module handles all logic required to prepare the dataset for training. This includes preprocessing and filtering the data, label binarization, stratified splitting, image augmentation, and the calculation of class-wise statistics used during loss balancing and regularization.

10.2 Uses

- Filters rows based on label conditions (e.g. excludes "No Finding" and "Hernia").
- Converts multi-label strings into binary disease columns.
- Splits data to create training and validation sets.
- Constructs image transformation pipelines (base + augmented).
- Computes class-wise weights for BCE loss and average label count per sample.

10.3 Syntax

10.3.1 Exported Constants

None.

10.3.2 Exported Access Programs

Name	**In**	**Out**
preprocessDataframe	dataframe	filteredDataframe
binarizeLabels	dataframe	binary-labelDataframe
splitData	dataframe	trainDataframe, valDataframe
applyTransforms	trainDataframe, valDataframe	augmentedTrainDataframe, augmentedValData
computeTrainingStats	trainDataframe	posWeight: Tensor, avgPositive: float

10.4 Semantics

10.4.1 State Variables

- `data`: Raw or filtered DataFrame.
- `trainDataframe, valDataframe`: Split datasets used for training and validation.
- `transform`: Torchvision transform pipeline for images.

- `posWeight`: Tensor for loss class balancing.
- `avgPositive`: Float indicating average number of labels per sample.

10.4.2 Environment Variables

None.

10.4.3 Assumptions

- Labels in the CSV are pipe-separated.
- All image inputs are grayscale and resized to 128x128.
- Class distribution is imbalanced and requires correction.
- Augmentations apply only to training data.

10.4.4 Access Routine Semantics

preprocessDataframe(data: DataFrame, trim: bool):

- **Transition:** Removes image data that contain hernia or no finding. Filters rows to retain meaningful diagnostic labels.
- **Output:** Cleaned DataFrame suitable for label encoding.
- **Exceptions:** Raises `DataParsingError` if structure is invalid or columns are missing.

binarizeLabels(data: DataFrame):

- **Transition:** Parses the Finding Labels string field into 13 binary columns representing the disease classes in `allLabels`.
- **Output:** DataFrame with additional binary columns for each disease.
- **Exceptions:** Raises `LabelProcessingError` if the label field is missing or improperly formatted.

splitData(data: DataFrame):

- **Transition:** Performs an 80/20 train-validation split with stratification based on a partial hash of the Finding Labels.
- **Output:** `trainDataframe`: Training subset, `valDataframe`: Validation subset.
- **Exceptions:** Raises `SplitError` if stratification fails or sample size is too small.

createTransforms(augment: bool):

- **Transition:** Constructs a Torchvision transformation pipeline. Applies randomized augmentations like horizontal flip, affine transformation, and brightness/contrast jitter to the training dataset. Applies only base preprocessing (resize, normalize) to the validation dataset.
- **Output:** Augmented validation and training datasets.
- **Exceptions:** Raises `TransformConfigError` if the transformation composition fails due to internal misconfiguration.

computeTrainingStatistics(train_df: DataFrame):

- **Transition:** Calculates two statistics from the training set:
  - `posWeight`: Ratio of negative to positive samples per class (for use in BCEWith-LogitsLoss).
  - `avgPositive`: Average number of positive labels per training sample (used in sparsity regularization).
- **Output:**
  - `posWeight`: Tensor of shape (13,).
  - `avgPositive`: Float scalar.
- **Exceptions:** Raises `ValueError` if data is malformed or if class counts include zeros.

11 MIS of Authorization Module

11.1 Module

Authorization
This module is responsible for handling user authentication and registration. It enables secure access to protected endpoints using JWT tokens. It manages login, signup, token generation, validation, and user roles for access control.

11.2 Uses

- Enables user login and registration via REST APIs.
- Generates and validates JWT tokens for session-less authentication.
- Filters incoming requests and injects authenticated users into the Spring Security context.
- Defines and persists users and roles using JPA and MySQL.

11.3 Syntax

11.3.1 Exported Constants

None.

11.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
login	loginDto	JWTAuthResponse	UsernameNotFoundException
register	RegisterDto	Success Message	APIException
generateToken	Authentication	JWT String	APIException
validateToken	String (JWT)	Boolean	APIException
getUsername	String (JWT)	Username	APIException

11.4 Semantics

11.4.1 State Variables

- **UserRepository:** Handles retrieval and persistence of user data.
- **RoleRepository:** Handles retrieval and persistence of roles.
- **JwtTokenProvider:** Responsible for JWT operations like generation and validation.

- **CustomUserDetailsService:** Loads user credentials and authorities for Spring Security.

11.4.2 Environment Variables

- **app.jwt-secret:** Secret used to sign JWT tokens.
- **app.jwt-expiration-milliseconds:** Expiry duration for the JWT token.

11.4.3 Assumptions

- Usernames and emails are unique and validated at registration.
- Valid roles (`ROLE_USER`, `ROLE_ADMIN`, etc.) are seeded or pre-created in the system.
- Spring Security is enabled and properly configured (refer to `SecurityConfig` module).
- The secret key and expiration time are defined in the application properties.

11.4.4 Access Routine Semantics

login(LoginDto)

- **Transition:** Validates credentials, loads the user, and generates a JWT token.
- **Output:** `JwtAuthResponse` containing the access token and type.
- **Exceptions:** Throws `UsernameNotFoundException` if user is not found; may throw `APIException` on JWT issues.

register(RegisterDto)

- **Transition:** Registers a new user by saving to the database with encrypted password and assigned roles.
- **Output:** `String` message indicating success.
- **Exceptions:** Throws `APIException` if username or email already exists.

generateToken(Authentication)

- **Transition:** Generates a JWT token using username and configured expiration duration.
- **Output:** JWT token as a `String`.
- **Exceptions:** None explicitly thrown.

validateToken(token)

- **Transition:** Parses the token and validates its signature, expiration, and format.
- **Output:** Boolean indicating whether the token is valid.
- **Exceptions:** Throws `APIException` with relevant HTTP status if token is malformed, expired, unsupported, or null.

getUsername(token)

- **Transition:** Extracts the username (subject) from the decoded JWT claims.
- **Output:** Username as a `String`.
- **Exceptions:** Throws `APIException` if token is invalid.

doFilterInternal(request, response, filterChain) (JWTAuthenticationFilter)

- **Transition:** Extracts token from header, validates it, loads user, and sets security context.
- **Output:** None (side-effect: injects user into `SecurityContextHolder`).
- **Exceptions:** Continues filter chain even on invalid tokens (handled globally).

11.4.5 Local Functions

getTokenFromRequest(request): Extracts Bearer token from the request header.

12 MIS of Config Module

12.1 Module

Config

This module is responsible for configuring key aspects of the Spring Boot backend application. It includes application-wide configurations such as REST template instantiation, CORS settings, security filter chains, and network connectivity testing between Java and the Flask ML backend.

12.2 Uses

- Enables CORS policies to allow frontend-backend communication.
- Provides `RestTemplate` bean for HTTP communication.
- Secures API endpoints using `JWT` authentication and Spring Security.
- Verifies network connectivity to the Flask API used for ML inference.

12.3 Syntax

12.3.1 Exported Constants

None.

12.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
corsConfigurer	None	WebMvcConfigurer	-
authenticationManager	AuthenticationConfig	AuthenticationManager	Exception
passwordEncoder	None	PasswordEncoder	Exception
securityFilterChain	HttpSecurity	SecurityFilterChain	Exception

Table 3: Exported Access Programs for `Config` Module

12.4 Semantics

12.4.1 State Variables

- `authenticationEntryPoint`: Handles unauthorized access attempts.
- `userDetailsService`: Loads user-specific data for authentication.
- `authenticationFilter`: Filters incoming requests and validates JWT tokens.

12.4.2 Environment Variables

- Requires access to GPU resources for computationally intensive image generation.
- File system access for saving synthetic datasets.

12.4.3 Assumptions

- Input configurations are valid and specify all necessary parameters.
- The environment has the required computational resources to execute the process.

12.4.4 Access Routine Semantics

initializeGenerator(ConfigParams):

- **Transition:** Sets up the image generator based on configuration parameters. Updates `generatorConfig`.
- **Output:** Generator instance ready for image generation.
- **Exceptions:** Throws `InvalidConfigError` if the configuration parameters are invalid.

generateImages(GenInstance, ImageParams):

- **Transition:** Uses the generator instance to create synthetic images based on the specified parameters.
- **Output:** A dataset of synthetic chest X-ray images.

saveGeneratedImages(SyntheticDataset, OutputPath):

- **Transition:** Saves the synthetic dataset to the specified file/folder.
- **Output:** Receives confirmation that the dataset was successfully saved.
- **Exceptions:** Throws `FileWriteError` if the save operation fails (e.g., non-existing file path).

13 MIS of MLBackend

13.1 Module

MLBackend

This module provides the backend logic for receiving medical images, performing deep learning-based analysis using a ResNet50 model, and returning structured diagnostic predictions. It uses Flask as the web framework and PyTorch for inference.

13.2 Uses

- Serves as the API layer between the frontend and the trained chest X-ray classification model.

13.3 Syntax

13.3.1 Exported Constants

None.

13.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
test_connection	None	API Reachability Message	-
predict_api	Image File	Predictions & Probabilities	PredictionError
load_model	Model Path, Device	Loaded Model	LoadError
predict	Image Bytes	Diagnoses, Class Probabilities	PredictionError

Table 4: Exported Access Programs for `MLBackend` Module

13.4 Semantics

13.4.1 State Variables

- `model`: An instance of `OptimizedChestXRayResNet` loaded from disk.
- `device`: Either `cuda` (GPU) or `cpu`, selected at runtime.
- `class_labels`: List of 14 disease classes used to interpret model output.

13.4.2 Environment Variables

- The model file `resnet50model.pth` is present in the backend directory.
- All dependent libraries are installed.
- Model was trained on images with 14 disease classes.
- Incoming images are valid and conform to expected input types.

13.4.3 Assumptions

- The model file `resnet50model.pth` is present in the backend directory.
- All dependent libraries required for inference are installed (PyTorch, Flask, etc.).
- Model was trained using images with 14 disease classes.
- Incoming image files are valid (correct format and size) and conform to expected input types.

13.4.4 Access Routine Semantics

predict_api()

- **Transition:** Accepts an image file via POST request. Runs prediction pipeline and returns disease list and class probabilities.
- **Output:** `"predictions": {[diagnoses], [class_probabilities]}`
- **Exceptions:**
  - Returns status 400 if no file is provided.
  - Returns status 500 if inference or transformation fails (`PredictionError`).

load_model(model_path, device)

- **Transition:** Loads and prepares a PyTorch model for evaluation.
- **Output:** Instantiated, ready-to-use `OptimizedChestXRayResNet`.
- **Exceptions:** Throws `LoadError` if file is missing or incompatible.

predict(image_bytes)

- **Transition:** Applies transformation pipeline and runs inference.
- **Output:**
  - Diagnoses (list of detected classes with probability ¿ 0.5).
  - `class_probs` (dictionary of all 14 class probabilities).
- **Exceptions:** Returns a dictionary with an `error` key if prediction fails.

14 MIS of ModelArchitecture

14.1 Module

ModelArchitecture

This module defines the convolutional neural network (CNN) architecture used for multi-label disease classification of chest X-ray images. It builds upon a pretrained MobileNetV2 backbone and introduces a customized classifier head for 13-class output. The architecture is designed for transfer learning, allowing the base to remain frozen while training the classifier layers.

14.2 Uses

- Loads MobileNetV2 with pretrained ImageNet weights.
- Freezes base convolutional layers to retain learned features.
- Defines a custom classifier head for 13-class multi-label prediction.
- Assembles the final CNN for use in training and inference.
- Outputs raw logits suitable for BCEWithLogitsLoss.

14.3 Syntax

14.3.1 Exported Constants

None.

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
initializeBaseModel	None	pretrainedBaseModel	ModelInitializationError
freezeBaseLayer	baseModel	baseModelWithFrozenLayers	LayerFreezingError
assembleFullModel	frozenLayerBaseModel	finalCNNModel	ModelAssemblyError

Table 5: Exported Access Programs for `ModelArchitecture` Module

14.4 Semantics

14.4.1 State Variables

- `baseModel`: Pretrained MobileNetV2 model.

- `classifier`: Internally constructed multi-layer classification head.
- `fullModel`: Final nn.Module combining base and classifier.

14.4.2 Environment Variables

None.

14.4.3 Assumptions

- The classifier head is suitable for multi-label problems (e.g., 13 classes).
- The loss function used will be BCEWithLogitsLoss.

14.4.4 Access Routine Semantics

initializeBaseModel()

- **Transition:** Loads the pretrained MobileNetV2 model from Torchvision.
- **Output:** `BaseModel` loaded.
- **Exceptions:** Raises `ModelInitializationError` if model loading fails or required packages are missing.

freezeBaseLayers(base_model: nn.Module)

- **Transition:** Disables gradient updates for all parameters in the base model.
- **Output:** Frozen base model with `requiresGrad = False` on all layers.
- **Exceptions:** Raises `LayerFreezingError` if the model structure is incompatible with this operation.

assembleFullModel(base_model: nn.Module, num_classes: int)

- **Transition:** Constructs a classification head composed of adaptive average pooling, flattening, batch normalization, dropout, and fully connected layers ending in a final layer with `num_classes` outputs. Then attaches this head to the base model to form the full model.
- **Output:** The full model which is a PyTorch `nn.Module` that outputs raw logits for multi-label classification.
- **Exceptions:** Raises `ModelAssemblyError` if the classifier cannot be constructed or integrated into the base model.

15 MIS of Training

15.1 Module

Training

This module manages the full training process of the model. It controls the training loop, applies loss functions and regularization, tracks performance metrics, and saves the best model checkpoint. It includes support for early stopping, custom regularization techniques (e.g., margin and sparsity), and consistent evaluation using multi-label classification metrics.

15.2 Uses

- Trains the model on labeled data using backpropagation and optimization.
- Applies BCEWithLogitsLoss for multi-label prediction.
- Supports additional loss terms: margin regularization and sparsity loss.
- Tracks epoch-wise accuracy and validation performance.
- Applies early stopping if validation does not improve.
- Saves the best-performing model.

15.3 Syntax

15.3.1 Exported Constants

None.

15.3.2 Exported Access Programs

Name	**In**	**Out**	**Excep**
trainModel	Model, trainLoader, valLoader, config	Path to saved model file	Trainin
evaluateModelPerformance	logits, labels	performanceMetrics	Metric
calculateMarginLoss	logits, labels	marginRegularizationLoss	Margin

Table 6: Exported Access Programs for `Training` Module

15.4 Semantics

15.4.1 State Variables

- `trainLoader, validLoader`: Input DataLoaders.
- `model`: Neural network model to train.
- `config`: Dictionary of optimizer, loss function, device, training options.
- `bestValLoss`: Lowest validation loss.
- `earlyStopCount`: Early stopping counter.

15.4.2 Environment Variables

None.

15.4.3 Assumptions

- Config contains valid settings for optimizer, criterion, etc.
- Model is constructed to output logits for BCEWithLogitsLoss.
- Model checkpoint directory exists.

15.4.4 Access Routine Semantics

trainModel(model, trainLoader, validLoader, config: dict)

- **Transition:** Uses components from config (e.g., optimizer, loss function, device, margin/sparsity flags) to train the model over multiple epochs. Tracks validation loss and metrics. Saves the best model checkpoint based on validation performance.
- **Output:** Model checkpoint file: `./models/model_<timestamp>.pth`.
- **Exceptions:** Raises `TrainingRuntimeError` for failures like memory issues, config errors, or unexpected runtime exceptions.

evaluateModelPerformance(logits: Tensor, labels: Tensor)

- **Transition:** Takes the model's raw predictions (`logits`) and compares them to the correct labels to measure how well the model is performing.
- **Output:** Calculates and outputs the multilabel accuracy (how often each disease label is predicted correctly) and subset accuracy (how often the entire prediction for an image is exactly right).

calculateMarginLoss(logits: Tensor, labels: Tensor)

- **Transition:** Calculates margin loss by penalizing low-confidence predictions near zero.
- **Output:** Float value representing additional margin penalty to be added to total loss.
- **Exceptions:** Raises `MarginLossError` if logits or labels are malformed or misaligned.

References

[1] Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2003). *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall, Upper Saddle River, NJ.

[2] Hoffman, D. M., & Strooper, P. A. (1995). *Software Design, Automated Testing, and Maintenance: A Practical Approach.* International Thomson Computer Press, New York, NY. URL: http://citeseer.ist.psu.edu/428727.html

16 Appendix

16.1 Reflection Questions

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

The purpose of reflection questions is to assess individual and team learning experiences and identify areas for future improvement. Reflection is an essential component of the software development process, ensuring continuous improvement and better decision-making.

16.1.1 1. What went well while writing this deliverable?

- Tasks were effectively divided among team members, ensuring steady progress.
- Using the `Module Guide` (MG) and `SRS` as references helped maintain consistency across documents.
- Regular team meetings and discussions clarified misunderstandings, ensuring each module was well-defined and fit into the overall architecture.
- The design process was clearly outlined, allowing for efficient documentation of our diffusion model approach.
- Our prior research and understanding of CNN models and various training strategies helped us articulate the technical aspects clearly.

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

- Ensuring consistency across documents was challenging, as some module definitions overlapped or lacked clarity.
  - **Resolution:** We refined definitions in the MIS to remove ambiguities and iteratively reviewed sections for alignment.
- Formatting in LaTeX was another hurdle, especially when working with tables and cross-references.
  - **Resolution:** Using Overleaf streamlined collaboration and debugging.

16.1.3 3. Which design decisions stemmed from clients/peers, and which came from research?

Several design decisions stemmed from discussions with peers and client feedback, including the need for a modular architecture and the separation of frontend and backend responsibilities. Suggestions like grouping metric calculations and simplifying transform logic were peer-driven for better clarity and usability.

- **On the other hand,** decisions such as using a pretrained MobileNetV2, implementing multi-label classification with `BCEWithLogitsLoss`, and incorporating regularization (margin/sparsity) were based on research into best practices for medical image classification.

16.1.4 4. What parts of other documents needed changes, and why?

- **SRS Functional Requirements:** Clarified the difference between `DatasetHandler` and `DataPreprocessing` to avoid redundancy.
- **Traceability Matrix:** Adjusted anticipated changes (AC5) to include `IntegrationModule` for handling data exports.
- **Hazard Analysis:** Expanded to include risks related to false positives and false negatives in AI-based diagnostics.
- **Mission Goals (MG):** Revised to better reflect user workflows.
- **User Interaction Model:** Refined to ensure radiologists could override AI suggestions when necessary.

16.1.5 5. What are the limitations of your solution?

- **Hardware constraints:** Dependence on GPUs/TPUs may limit accessibility for smaller institutions.
- **Data availability:** Limited access to real medical datasets impacts training quality.
- **Regulatory compliance:** More effort is required to meet HIPAA/GDPR standards for handling medical data.
- **AI explainability:** Additional interpretability tools could provide clearer reasoning behind outputs.
- **Computational efficiency:** Training models require significant GPU resources, limiting real-time applications.

16.1.6 6. What alternative designs were considered, and why was this one chosen?

We considered several design approaches before choosing our final architecture. A monolithic design was simpler and easier to build, but it would have made future changes harder due to its tightly connected components. Microservices were another option that would allow us to separate different parts of the system (like the model, preprocessing, and UI), but this added unnecessary complexity for our current needs.

We also looked at different types of models. Vision Transformers were appealing because they can capture patterns across the entire image, but they require more computing power and are not yet widely used in medical imaging. U-Net was considered for its ability to show where abnormalities appear in the image, but it requires detailed annotations, which we didn't have. Diffusion models were also explored for their strengths in uncertainty estimation and image generation, but they were slower and not focused on classification tasks.

In the end, we chose a modular CNN-based classifier because it was fast, reliable, and well-supported in medical AI research. It allowed us to build a clean pipeline that can be maintained and expanded easily while keeping the model efficient and practical for our dataset and use case.