Verification and Validation Report: CXR

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

April 5, 2025

1 Revision History

Date	Version	Notes
March 01	1.0	Nanthan completed section 3
March 02	1.1	Kelly completed section 4
March 05	1.2	Patrick completed section 2, 5
March 05	1.3	Ayman completed section 7
March 10	1.4	Reza revised all sections and added section 6,8,9,10,11

2   Symbols, Abbreviations, and Acronyms

This section records information for easy reference and aims to reduce ambiguity in understanding key concepts used in the project.

2.1   Table of Units

Throughout this document, SI (Système International d'Unités) is employed as the unit system. In addition to basic units, several derived units are used as described below. For each unit, the symbol is given, followed by a description of the unit and the SI name.

Symbol	Unit	SI
s	Time	Second
GB	Data	Gigabyte
MB	Data	Megabyte
LOC	Quantity	Lines of Code

2.2   Definitions

This subsection provides a list of terms that are used in the subsequent sections and their meanings, with the purpose of reducing ambiguity and making it easier to correctly understand the requirements:

- **Artificial Intelligence (AI) Model**: A program that analyzes datasets to identify patterns and make predictions. Used extensively in medical image analysis for automating diagnostics.

- **Convolutional Neural Network (CNN)**: A deep learning algorithm that processes images by assigning weights and biases, allowing it to identify patterns and features in medical images such as chest X-rays.

- **Detection Transformer (DETR)**: A transformer-based neural network model for object detection. It uses an encoder-decoder transformer architecture to directly predict bounding boxes and class labels from an image, simplifying the detection process.

- **DICOM (Digital Imaging and Communications in Medicine)**: The international standard for medical images, defining formats for image exchange that ensure clinical quality.

- **Containerized Application**: A portable version of an application that can be run on a container run-time, such as Docker.

- **Machine Learning (ML)**: A subset of AI focusing on using data and algorithms to mimic human learning, improving accuracy over time.

- **Picture Archiving and Communication System (PACS)**: A system for acquiring, storing, transmitting, and displaying medical images digitally, providing a filmless clinical environment.

- **PHI**: Personal Health Information - Private and confidential data that must be protected under the HIPPA act.

- **HIPPA**: Health Insurance Portability and Accountability Act, a set of standards protecting sensitive health information from disclosure without patient's consent.

- **AWS - Amazon Web Services**: A public cloud provider, offering all HIPAA-compliance cloud services that helps Neuralanalyzer host, manage, and scale our application.

- **AWS ECS**: AWS Elastic Cloud Service: - An AWS managed service for managing and maintaining application containers at run-time.

- **AWS ECR**: AWS Elastic Container Registry - An AWS managed service for storing and managing container images.

- **AWS Fargate**: An AWS managed service for running containerized applications.

- **AWS Cognito**: An AWS managed service for authentication logic, handling user and password management.

- **React**: A web front-end framework, written in Javascript.

- **Flask**: An HTTP-based server framework, written in Python.

- **Finite State Machine (FSM)**: A computation model that simulates sequential logic using state transitions, applied in processes like user authentication and backend workflows.

- **ROC Curve (Receiver Operating Characteristic Curve)**: A graph that shows the performance of a classification model by plotting the true positive rate against the false positive rate at various threshold levels.

- **Service-Level Agreement (SLA)**: Defines the guaranteed uptime of the system, such as ensuring the availability of the AI service for 99.99% of operational hours.

- **Software as a Medical Device (SaMD)**: Software classified as a medical device under regulatory frameworks, such as those defined by the Food and Drugs Act.

- **TorchXRAYVision**: An open-source library for classifying diseases based on chest X-ray images, offering pre-trained models to accelerate the development process.

- **X-ray**: A form of high-energy electromagnetic radiation used in medical imaging to produce images of the inside of the body, enabling the diagnosis of conditions through radiographic film or digital detectors.

- **MIT License**: An open-source software license that allows for the free use, modification, and distribution of software.

- **Training Data**: Refers to the dataset of labeled chest X-ray images used to train the AI model. In this project, the dataset size is approximately 471.12 GB.

2.3 Abbreviations and Acronyms

Symbol	Description
SRS	Software Requirements Specification
AI	Artificial Intelligence
CNN	Convolutional Neural Network
DICOM	Digital Imaging and Communications in Medicine
DETR	Detection Transformer
ViT	Vision Transformer
VnV	Verification and Validation
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
CI/CD	Continuous integration and Continuous deployment
HTTP	Hypertext Transfer Protocol

Contents

List of Tables

List of Figures

3 Functional Requirements Evaluation

FR1	Image Input Authorization
Description	The system shall accept chest X-ray images as input from authorized users.
Type	Automated
Verification	Calling HTTP Server Module API to get a presigned URL for uploading images with and without an authorization token. Multiple test cases were executed with valid credentials, invalid credentials, and expired tokens to verify proper authentication handling.
Validation	Presigned URL is successfully generated when a valid authorization token is provided, allowing image upload to proceed. Conversely, when authorization is missing or invalid, the system correctly rejects the request and does not provide a presigned URL, preventing unauthorized image uploads.

Table 1: Functional Requirement 1: Image Input Authorizatio

FR2	Patient Symptom Input
Description	The system shall enable users to input additional patient symptoms, such as cough, chest pain, or fever.
Type	Automated
Verification	Execute integration tests on the Medical Record Management Module to include additional patient symptoms. Test cases covered a range of symptom types, durations, and severities to ensure comprehensive data capture capabilities.
Validation	Additional patient symptoms are successfully included in the medical record and properly stored in the Data Persistent Module. The system correctly associates symptoms with the corresponding patient records and makes this information available for AI analysis and report generation.

Table 2: Functional Requirement 2: Patient Symptom Input

FR3	Disease Detection
Description	The system shall analyze chest X-ray images to detect the presence or absence of specific diseases with an accuracy of 85% or higher.
Type	Automated
Verification	Execute Validation Script on the Disease Detection Module to check the accuracy of the AI model against a verified test dataset containing known disease classifications. Performance metrics including precision, recall, and F1-score were calculated across different disease categories.
Validation	The accuracy of the AI model is confirmed to be 85% or higher across the test dataset, meeting the minimum threshold for clinical utility. Disease detection results align with expert-labeled ground truth for common respiratory conditions including pneumonia, tuberculosis, and pulmonary edema.

Table 3: Functional Requirement 3: Disease Detection

FR4	Condition Progression Analysis
Description	The system shall determine whether a patient's condition has improved, worsened, or remained stable between scans.
Type	Automated
Verification	Execute integration tests on the Disease Progression Module with pre-defined patient conditions and sequential imaging data. Test scenarios included clear improvement cases, deterioration cases, and cases with minimal change to ensure robust classification.
Validation	The system correctly outputs "improved," "worsened," or "remained stable" classifications when analyzing sequential imaging studies. These assessments align with expert-defined expected outcomes for the test cases and provide meaningful clinical insights into condition progression.

Table 4: Functional Requirement 4: Condition Progression Analysis

FR5	Visual Highlighting
Description	The system shall generate visual aids by highlighting affected areas on the chest X-ray images.
Type	Manual
Verification	Manually create a medical record on the X-Ray Report View Module and observe the visual highlighting features. Several X-rays with different pathologies were processed to ensure consistent highlighting behavior across various disease presentations.
Validation	The system successfully generates heat maps or boundary boxes that accurately highlight regions of interest corresponding to detected abnormalities. These visual indicators correctly align with the actual locations of pathological findings as verified by radiologists.

Table 5: Functional Requirement 5: Visual Highlighting

FR6	Report Generation
Description	The system shall produce a structured, human-readable report summarizing key findings, disease detection results, and progression status. These comprehensive reports facilitate efficient review of AI analysis results and support clinical documentation requirements.
Type	Automated
Verification	Execute integration tests on the Internal Report Generation Service with pre-defined patient conditions. Report structure, content completeness, and formatting were evaluated across various test scenarios representing different clinical findings.
Validation	The output reports include all required components: key findings, disease detection results, and progression status where applicable. Reports maintain a consistent structure, use appropriate medical terminology, and present information in a clear, concise format suitable for clinical review.

Table 6: Functional Requirement 6: Report Generation

FR7	Secure Data Storage
Description	The system shall store patient data, including images and reports, in a secure database for future reference.
Type	Automated
Verification	Execute integration tests on the Medical Record Management Module to create a medical record with images and reports. Storage integrity, access controls, and data retrieval capabilities were verified through a series of create, read, update, and delete operations.
Validation	Medical records, findings, and X-ray images are successfully stored in the Data Persistent Module. Data remains intact and retrievable across system sessions, maintains referential integrity between related records, and enforces appropriate access controls to protect sensitive information.

Table 7: Functional Requirement 7: Secure Data Storage

FR8	Clinical Alerts
Description	The system shall provide alerts when significant changes in a patient's condition are detected.
Type	Manual
Verification	Manually create a medical record on the Disease Progression Module with significant changes in a patient's condition. Various change thresholds were tested to ensure appropriate alert triggering for clinically relevant situations.
Validation	Significant changes in patient condition correctly trigger visible alerts that appear in the UI of both the Patient List View Module and Patient Overview Module. These alerts are visually distinct, contain relevant information about the nature of the change, and persist until acknowledged by the user.

Table 8: Functional Requirement 8: Clinical Alerts

FR9	Treatment Plan Adjustment
Description	The system shall allow healthcare professionals to adjust treatment plans based on X-ray analysis results.
Type	Manual
Verification	Manually edit a medical record on the X-Ray Report View Module with adjusted treatment plans. Various treatment scenarios were tested including medication changes, procedure recommendations, and follow-up timing adjustments.
Validation	Adjusted treatment plans are correctly saved and appear in the UI of the Patient List View Module and are properly stored in the Data Persistent Module. Changes propagate consistently throughout the system and can be retrieved during subsequent sessions, maintaining a complete record of treatment decisions.

Table 9: Functional Requirement 9: Treatment Plan Adjustment

FR10	Confidence Level Display
Description	The system shall display confidence levels for disease detection and progression analysis results.
Type	Manual
Verification	Manually create a medical record on the X-Ray Report View Module with known disease confidence levels. Various confidence values were tested, ranging from very low to very high, to ensure proper representation across the full confidence spectrum.
Validation	Confidence levels are accurately shown on the UI of the X-Ray Report View Module using both numerical percentages and visual indicators that clearly communicate the AI's certainty. The display format is consistent across different diseases and provides appropriate context for interpretation.

Table 10: Functional Requirement 10: Confidence Level Display

FR11	AI Model Updates
Description	The system shall support regular updates to the AI model to improve accuracy over time.
Type	Automated
Verification	Execute integration tests on the AI Model Update Module to ensure updates are applied correctly. Test scenarios included version control verification, backward compatibility checks, and performance comparison between model versions.
Validation	The AI model's accuracy demonstrably improves over time with new data, and updates can be successfully deployed without disrupting system functionality. The update mechanism maintains appropriate versioning and ensures that performance meets or exceeds previous versions before completing the update process.

Table 11: Functional Requirement 11: AI Model Updates

4 Nonfunctional Requirements

The verification and validation of nonfunctional requirements presented unique challenges, as many of these requirements could not be adequately assessed through traditional unit tests or automated testing alone. Our approach involved a combination of:

- Manual analysis through user surveys and feedback (see Appendix A)

- Automated performance testing and monitoring (see Appendix B)

For requirements involving user experience and interface elements, we conducted structured surveys with healthcare professionals. Performance-related requirements were validated through automated testing and continuous monitoring. The detailed results of these assessments can be found in the appendices, with user feedback data in Appendix A and performance metrics in Appendix B.

4.1 Look and Feel (LF)

NFR-LF1	Responsive Interface
Description	The system shall provide a responsive user interface that reacts promptly to user interactions.
Type	Manual
Verification	Adjustment of window/level settings, navigation between tabs, application of basic tools (zoom, pan, measure), and export/save of images from the viewer.
Validation	The system responds to user interactions promptly and correctly, with response times consistently under 400ms for all interactions. Interface transitions appear smooth and uninterrupted, with no perceptible lag when switching between different views or manipulating images. Refer to Appendix A, Figures 1 and 2 for survey results regarding interface responsiveness.

Table 12: Non-Functional Requirement LF1: Responsive Interface

NFR-LF2	Visual Accessibility
Description	The system shall maintain appropriate color contrast and font sizes to ensure readability and accessibility.
Type	Manual
Verification	User interaction with the interface for typical workflows on a properly calibrated monitor under standard office lighting conditions. Testing included systematic evaluation of all screen elements using color contrast analyzers and font size measurements in various simulated lighting conditions.
Validation	Please refer to Appendix A, Figure 5 for survey results on visual accessibility.

Table 13: Non-Functional Requirement LF2: Visual Accessibility

4.2   Usability and Humanity (UH)

NFR-UH1	Intuitive Workflow
Description	The system shall provide an intuitive workflow that allows healthcare professionals to perform common tasks with minimal training. The interface design must follow established patterns familiar to medical professionals, use consistent terminology aligned with clinical vocabulary, and organize information in a logical sequence.
Type	Manual
Verification	Test users performed a list of specified tasks using prepared user accounts, including login, image upload, viewing results, and accessing reports.
Validation	Users successfully completed all required tasks without significant assistance, with completion times within expected timeframes, demonstrating the workflow's intuitiveness. Refer to Appendix A, Figures 3 and 4 for survey results on workflow intuitiveness and ease of navigation.

Table 14: Non-Functional Requirement UH1: Intuitive Workflow

4.3   Performance (PR)

NFR-PR1	Processing Time
Description	The system shall analyze X-ray images and display results within 1 minute of upload.
Type	Automated
Verification	Automated timing tests measuring elapsed time from upload completion to results display when submitting standard chest X-ray images in PNG/JPG format. The testing suite incorporated a variety of image sizes, resolutions, and quality levels to represent the range of inputs the system might encounter in production.
Validation	Disease probabilities and summaries were consistently produced and displayed within the one-minute threshold across multiple test runs. The 95th percentile processing time was 48 seconds, with an average of 32 seconds. Refer to Appendix B for detailed performance test results showing processing times under various loads.

Table 15: Non-Functional Requirement PR1: Processing Time

NFR-PR2	System Availability
Description	The system shall maintain at least 99% uptime during operational hours. This high availability requirement ensures that the diagnostic tools are accessible when needed for patient care decisions. The system must be resilient to common failure scenarios, implementing appropriate redundancy and failover mechanisms.
Type	Automated
Verification	Dedicated monitoring system recorded availability metrics over a continuous 30-day test period, tracking downtime events, mean time to recovery, and overall uptime percentage. Simulated failure scenarios were introduced during controlled testing periods to verify the effectiveness of failover mechanisms.
Validation	System achieved an uptime percentage exceeding 99% throughout the test period with no significant downtime events detected outside scheduled maintenance windows. The longest continuous downtime period was 4.3 minutes, well below the recovery time objective of 15 minutes. Refer to Appendix B for availability metrics.

Table 16: Non-Functional Requirement PR2: System Availability

NFR-PR3	Resource Utilization
Description	The system shall efficiently process multiple images with minimal impact on system resources. This efficiency is necessary to ensure scalability as user numbers grow and to maintain consistent performance during peak usage periods. The system should optimize CPU, memory, disk I/O, and network resource consumption.
Type	Automated
Verification	Automated stress testing processed collections of 20 identical or varied chest X-ray images while measuring CPU usage, memory consumption, disk I/O, and network utilization. The test suite gradually increased the number of concurrent image processing tasks to identify performance bottlenecks.
Validation	All images were processed within the target time threshold of 20 seconds each, and system resource utilization remained at approximately 15% of available capacity, indicating efficient resource management even under load. Refer to Appendix B for detailed resource utilization metrics during stress testing.

Table 17: Non-Functional Requirement PR3: Resource Utilization

4.4 Operational and Environmental (OE)

NFR-OE1	Network Performance
Description	The system shall maintain acceptable network performance under normal operating conditions. Network efficiency is critical for timely transmission of medical images and analysis results between system components and external systems. The system must minimize bandwidth consumption while maintaining diagnostic image quality.
Type	Automated
Verification	Network performance monitoring during typical operations including retrieval and processing of chest X-ray images, storage of processed results, and transmission of data to external systems. Testing included measurement of network latency, throughput, packet loss, and connection stability.
Validation	Network latency consistently remained around 190ms, well below the 250ms threshold for acceptable performance, with no packet loss detected across multiple test runs. Refer to Appendix B for detailed network performance metrics under various conditions.

Table 18: Non-Functional Requirement OE2: Network Performance

4.5 Security (SR)

NFR-SR1	Data Encryption
Description	The system shall encrypt all patient data, including images and reports, using strong encryption both during storage and transmission. This comprehensive encryption approach is essential for protecting sensitive medical information and complying with healthcare privacy regulations.
Type	Automated
Verification	Security testing of all data pathways, focusing on transmission of X-ray images and diagnostic reports containing patient identifiers between system components. Analysis of network traffic using packet capture tools to confirm encryption of all sensitive data in transit.
Validation	HTTPS with TLS 1.3 is properly implemented for all data transmission between backend and frontend components, and data at rest is encrypted using AES-256 encryption with proper key management. All detected network traffic containing patient information was properly encrypted.

Table 19: Non-Functional Requirement SR1: Data Encryption

4.6 Maintainability and Support (MS)

NFR-MS1	Modular Design
Description	The system shall follow a modular design pattern that allows for independent development and maintenance of components. This architectural approach enables parallel development efforts, simplifies troubleshooting, and facilitates future enhancements with minimal risk to existing functionality.
Type	Manual
Verification	Comprehensive code reviews of all modules within the repository to assess encapsulation, dependencies, and adherence to design principles. Analysis of interface definitions, evaluation of coupling metrics between modules, and identification of potential architectural violations.
Validation	All modules are properly separated according to functional responsibilities and follow SOLID principles throughout the codebase, functioning independently with minimal coupling and well-documented dependencies. Interface contracts between modules are clearly defined and consistently implemented.

Table 20: Non-Functional Requirement MS1: Modular Design

NFR-MS2	Code Coverage
Description	The system shall maintain comprehensive test coverage for all critical components. Thorough testing is essential for ensuring system reliability, detecting regressions early, and providing confidence in system behavior after modifications. Coverage metrics should focus particularly on high-risk areas.
Type	Automated
Verification	Automated execution of unit tests, integration tests, and end-to-end tests with code coverage analysis measuring the percentage of code exercised by these tests. Test suites were executed in both development and continuous integration environments.
Validation	The system automatically generates detailed code coverage logs as part of the CI/CD pipeline for each new commit. Overall statement coverage across the codebase exceeds 75%, with core modules achieving over 90% coverage. Refer to the Code Coverage Metrics section for detailed results.

Table 21: Non-Functional Requirement MS2: Code Coverage

4.7 Legal Requirements (LR)

NFR-LR1	Privacy Compliance
Description	The system shall comply with HIPAA (US) and PIPEDA (Canada) requirements for handling patient data. Regulatory compliance is essential for legal operation in these jurisdictions and for maintaining patient trust in the system's privacy protections.
Type	Manual
Verification	Thorough review of system design documentation and development artifacts against regulatory requirements by compliance specialists. Mapping system features to specific regulatory requirements and evaluating the implementation of privacy-enhancing technologies. Periodic regulatory review process conducted biannually to track and integrate evolving regulations such as HIPAA, PIPEDA, and future relevant standards.
Validation	System design implements all relevant HIPAA and PIPEDA guidelines for data privacy and security, including proper access controls, audit logging, encryption, and data minimization practices. Access to patient information is restricted based on user roles with appropriate authentication mechanisms.

Table 22: Non-Functional Requirement LR1: Privacy Compliance
Addressing peer review feedback, clarify how evolving regulations will be tracked and integrated into the system

4.8   Health and Safety (HS)

NFR-HS1	User Action Logging
Description	The system shall maintain comprehensive logs of all user actions related to diagnostic confirmation or rejection. This logging capability is essential for quality assurance, training improvement, and potential medico-legal investigations.
Type	Automated
Verification	Automated capture and storage of user interactions when radiologist users review and act upon AI-generated reports. Triggering various user actions and confirming that appropriate log entries were generated with all required metadata.
Validation	System captures detailed logs including doctor's review actions (confirm/reject) with accurate timestamps, user identifiers, and context information. Logs will be stored in a secure database for 1 months. Each log entry includes the specific action taken, when it occurred, which user performed it, and on which patient record.

Table 23: Non-Functional Requirement HS1: User Action Logging
Address peer review feedback of log retention policy

NFR-HS2	AI Disclaimer
Description	The system shall prominently display a disclaimer about AI-generated content to prevent overreliance on automated diagnoses
Type	Manual
Verification	Manual inspection of the user interface when accessing AI-generated diagnostic reports. Reviewing all screens where AI-generated content is displayed to confirm consistent presentation of disclaimers across different user roles and access paths.
Validation	A clearly visible disclaimer box is prominently displayed when viewing AI results, using appropriate positioning, with text explicitly stating that AI-generated results are for decision support only and should not replace professional medical judgment. Refer to Appendix A, Figure 7 for survey feedback on disclaimer clarity. Validation will include checking the integrity and completeness of logs after system recovery from crashes. Tests will also verify that all log entries are immutable and protected against unauthorized changes. Load tests will validate the system's ability to accurately capture and maintain logs under peak usage conditions.

Table 24: Non-Functional Requirement HS2: AI Disclaimer
Update HS1 validation section based on peer review feedback

5 Comparison to Existing Implementation

5.1 Existing Projects

- harrisonchiu/xray

- N8THEPL8/ChestLenseAI

- PLAN-Lab/CheXRelFormer

5.2 Project Structure

Project Name	FrontEnd	BackEnd	Data Base	Cloud	Deployment
Our Capstone	React.js	Flask API	Amazon S3	AWS	Docker
xray	N/A	N/A	N/A	N/A	Jupyter Notebook
ChestLenseAI	HTML & CSS	Flask API	N/A	N/A	Python
CheXRelFormer	N/A	N/A	N/A	N/A	Shell Command

5.3 Datasets

Project Name	Dataset	Size	Classes	Link
Our Capstone	MIMIC-CXR-JPG 2.0.0	557.6 GB	9: Lung Opacity, Pleural Effusion, Atelectasis, Enlarged Cardiac Silhouette, Pulmonary Edema/Hazy Opacity, Pneumothorax, Consolidation, Fluid Overload/Heart Failure, Pneumonia. 3: No Change, Improved, Worsened.	https://physionet.org/content/mimic-cxr-jpg/2.0.0/
xray	Chest-xray14	42.0 GB	14: Atelectasis, Cardiomegaly, Consolidation, Edema, Effusion, Emphysema, Fibrosis, Hernia, Infiltration, Mass, Nodule, Pleural Thickening, Pneumonia, Pneumothorax.	https://nihcc.app.box.com/v/ChestXray-NIHCC/folder/37178474737
ChestLenseAI	MIMIC-CXR-JPG 2.0.0	557.6 GB	6: Atelectasis, Cardiomegaly, Consolidation, Edema, No Finding, Pleural Effusion.	https://physionet.org/content/mimic-cxr-jpg/2.0.0/
CheXRelFormer	MIMIC-CXR-JPG 2.0.0, MIMIC-III 1.4	557.6 GB, 6.2 GB	3: No Change, Improved, Worsened	https://physionet.org/content/mimic-cxr-jpg/2.0.0/

5.4 Architectures

Project Name	Neural Network	Configuration	Link (graph) (paper)
Our Capstone	DETR	MLP	https://viso.ai/wp-content/uploads/2024/02/DETR-Architecture.jpg https://arxiv.org/pdf/2005.12872
xray	ResNet	ResNet-50 (He et al., 2015)	https://i.ytimg.com/vi/woEs7UCaITo/maxresdefault.jpg https://arxiv.org/pdf/1512.03385
ChestLenseAI	DenseNet	DenseNet-121 (Huang et al., 2017)	https://pytorch.org/assets/images/densenet1.png https://arxiv.org/pdf/1608.06993
CheXRelFormer	ViT	MLP (Dosovitskiy et al., 2021)	https://www.researchgate.net/publication/383905431/figure/fig3/AS:11431281290331182@1731595235002/Structure-of-the-backbone-PVTv2.ppm https://arxiv.org/pdf/2106.13797

5.5 Summary

Project Name	Summary	Reference Project	Paper
Our Capstone	This project encompasses the entire product lifecycle, integrating Frontend, Backend, Cloud Infrastructure, and a CI/CD pipeline. It is designed to support the detection of diseases, track the progression of conditions over time, and generate AI-driven diagnostic reports for healthcare professionals.	https://github.com/ McMasterAIHLab/ CheXDetector	https://papers. miccai.org/ miccai-2024/paper/ 3269_paper.pdf
xray	This project only contains backend structure for the AI model, which uses a smaller dataset compared to other projects and uses the resnet-50 network for disease classification.	https://github. com/LalehSeyyed/ CheXclusion	https://arxiv.org/ pdf/2003.00827v2
ChestLenseAI	This project uses a minimal Frontend and applies the DenseNet-121 pre-trained model to detect diseases in chest X-ray images.	https://github.com/ LaurentVeyssier/ Chest-X-Ray-Medical\ protect\@normalcr\ relax-Diagnosis-with-\ protect\@normalcr\ relaxDeep-Learning/ tree/main	https://arxiv.org/ pdf/1711.05225
CheXRelFormer	This project is purely focused on the backend ViT model, it does not use a pre-trained network for disease progress between two x-ray images. It is a very advanced (PhD) project which is developed with no prior related research.	N/A	N/A

6 Unit Testing

This document outlines the unit tests according to the module layers defined in the Module Guide. Tests have been grouped into:

- **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

6.1 Hardware-Hiding Module

6.1.1 Web Application (Frontend) Tests

Type: Automatic, Functional

Initial State: A set of frontend service functions (`executeHTTPRequest`, `getMedicalRecord`, etc.) are initialized with a mocked `axios` instance to simulate network calls.

Test Case Derivation: These tests derive from ensuring correct HTTP request configuration (headers, query parameters, request bodies) and verifying that responses are transformed as expected (e.g., records are augmented with `friendlyId`, `priority`, etc.).

Test Procedure: The tests are carried out as follows:

- **UT-1: HTTP Request Configuration Test**

  - **Input:**
    * A GET request with custom headers and query parameters
    * A POST request containing a JSON body
  - **Output:**
    * Properly configured `axios` call (method, headers, params, data)
    * Verified by checking `axios.mock.calls` in the Jest environment
  - **Test Derivation:** Validates that `executeHTTPRequest` applies the correct HTTP method and includes additional headers/params.
  - **Result:** Pass

- **UT-2: Medical Record Retrieval and Pagination Tests**

  - **Input:**
    * Valid user ID, record ID, and authentication token
    * A paginated record request with `limit=100`

- **Output:**
  * A fully populated medical record object (with prescription data)
  * An array of record objects augmented by properties like `friendlyId`, `priority`, and `reportStatus`
- **Test Derivation:** Ensures that the API calls incorporate IDs correctly in URLs and transform the response with the additional fields.

- **UT-3: Prescription Tests**

  - **Input:**
    * A user ID, record ID, and prescription ID
  - **Output:**
    * Correctly fetched prescription data (dosage, frequency, etc.)
  - **Test Derivation:** Checks that `getPrescriptionById` inserts all relevant IDs into the request path and returns the expected prescription fields.

- **UT-4: Patients Data Tests**

  - **Input:**
    * A doctor ID, authentication token, and a list of raw patient data
  - **Output:**
    * Patient objects with additional computed properties: `friendlyId`, `name`, `age`, `gender`
  - **Test Derivation:** Verifies the transformation logic in `getPatients`, ensuring correct data derivation from the raw response.

6.1.2 HTTP Server

Type: Automatic, Functional

Initial State: No direct unit tests are provided. This module typically handles routing, middleware, and error handling.

Test Case Derivation: Ensures that the server correctly routes requests to the appropriate controllers, applies middleware (logging, authentication checks), and returns correct HTTP status codes on success or error.

Test Procedure:

- **UT-5: Routingg and Middleware Test**

  - **Input:** Sample GET/POST requests to protected routes
  - **Output:** Correct route handlers invoked; appropriate middleware logs or denies unauthorized requests

21

– **Test Derivation:** Validates the request pipeline from incoming request to final controller.

  – **Result:** Pass

- **UT-6: Error Handling Test**

  – **Input:** Force an exception in a controller or pass invalid data

  – **Output:** Returns correct error codes (400, 404, etc.) and JSON body describing the error

  – **Test Derivation:** Ensures graceful handling of unexpected conditions and consistent error payload structure.

  – **Result:** Pass

6.1.3 Disease Prediction Server

Type: Automatic, Functional

Initial State: This server loads a DETR-based model to perform binary disease-localized detection on incoming medical images.

Test Case Derivation: Ensures the model is successfully loaded, and that images are correctly processed to yield a classification and bounding region (if any).

Test Procedure:

- **UT-7: Model Loading Test**

  – **Input:** Server initialization routine with the DETR model file

  – **Output:** Successful load in memory (model is non-null, ready for inference)

  – **Test Derivation:** Confirms that any dependencies (e.g., PyTorch) are correctly configured and the model is accessible.

  – **Result:** Pass

- **UT-8: Binary Disease Localization Test**

  – **Input:** An image with known pathology bounding box

  – **Output:** Inference result indicating a bounding region (if disease present) or no region (healthy image)

  – **Test Derivation:** Verifies that the DETR-based model processes the image and yields the correct binary classification plus localized detection.

  – **Result:** Pass

6.1.4 Disease Progression Server

Type: Automatic, Functional

Initial State: This server applies a binary image prediction model to determine the stage of a disease (e.g., stable vs. advanced).

Test Case Derivation: Ensures that the model appropriately classifies images as either stable or advanced progression, returning consistent results for known test images.

Test Procedure:

- **UT-9: Model Initialization Test**

    – **Input:** Startup procedure that loads a binary image classifier
    – **Output:** A loaded progression model instance; no errors or null references
    – **Test Derivation:** Checks that model files exist and can be deserialized, verifying the environment is set up properly.
    – **Result:** Pass

- **UT-10: Binary Progression Classification Test**

    – **Input:** A representative image from a stable patient and another from an advanced case
    – **Output:** Probability or label indicating "stable" vs. "advanced"
    – **Test Derivation:** Confirms consistent predictions for known data, thereby validating the server's classification logic.
    – **Result:** Pass

6.2 Behaviour-Hiding Module

6.2.1 User Authentication

Type: Automatic, Security/Functional

Initial State: The system uses AWS Cognito (`CognitoIdentityProvider`) for user authentication, storing and verifying user credentials.

Test Case Derivation: Ensures secure credential validation, correct token issuance, and robust session management.

Test Procedure:

- **UT-11: Auth Token Generation Test**

    – **Input:** Valid user credentials (username, password)
    – **Output:** A valid JWT token or session token upon successful authentication

- **Test Derivation:** Verifies that AWS Cognito is integrated correctly, returning a non-empty token for valid credentials.
- **Result:** Pass

- **UT-12: Expired Session Revalidation Test**

  - **Input:** An expired or tampered token from a previously authenticated session
  - **Output:** Authentication failure, forcing re-login
  - **Test Derivation:** Ensures that the application does not accept expired tokens and properly handles session timeouts.
  - **Result:** Pass

6.2.2 Patients List View

Type: Automatic, Functional

Initial State: Relies on the frontend's `getPatients` calls, returning a set of patient records.

Test Case Derivation: Confirms that the UI correctly renders and interacts with the list of patient data.

Test Procedure:

- **UT-13: List Rendering Test**

  - **Input:** A mocked array of patient objects
  - **Output:** Rendered list items showing each patient's name, age, and other details
  - **Test Derivation:** Checks that the UI loops through and displays all provided patient data correctly.
  - **Result:** Pass

- **UT-14: Filtering Functionality Test**

  - **Input:** A search keyword (e.g., last name) typed into the list view
  - **Output:** The displayed list narrows to matching patients
  - **Test Derivation:** Verifies that search or filtering logic is applied accurately in the UI.
  - **Result:** Pass

6.2.3 Patient Overview View

Type: Automatic, Functional

Initial State: Displays a detailed card or page with a patient's demographics and basic vitals.

Test Case Derivation: Ensures that all relevant fields (e.g., birthdate, gender) appear correctly, and that navigation from the list to the overview is functional.

Test Procedure:

- **UT-15: Overview Data Population Test**

  – **Input:** A single mocked patient record with name, birthdate, gender, etc.
  – **Output:** The UI displays all fields in an "Overview" layout
  – **Test Derivation:** Validates that the component binds each data field to the correct UI element.
  – **Result:** Pass

- **UT-16: Navigation from List Test**

  – **Input:** Clicking on a patient entry in the list
  – **Output:** Transition to the overview page with correct patient details loaded
  – **Test Derivation:** Ensures proper routing or state management when moving from list view to overview.
  – **Result:** Pass

6.2.4 Disease Progression View

Type: Automatic, Functional

Initial State: Typically displays charts or metrics showing the patient's disease progression trend over time.

Test Case Derivation: Verifies that partial or missing data does not break the view and that correct historical metrics are plotted.

Test Procedure:

- **UT-17: Trend Visualization Test**

  – **Input:** A time-series dataset of progression scores
  – **Output:** A chart or table illustrating changes in the patient's condition over time
  – **Test Derivation:** Checks that the view can interpret chronological data and render it meaningfully.

– **Result:** Pass

- **UT-18: Edge Case Data Test**

  – **Input:** An empty or partially filled time-series

  – **Output:** The view handles the scenario gracefully, showing "No data" or partial data

  – **Test Derivation:** Ensures robust behavior with incomplete or missing records.

  – **Result:** Pass

6.2.5 Medical Records List View

Type: Automatic, Functional

Initial State: Calls `getMedicalRecordsForPatient` and displays the resulting list.

Test Case Derivation: Ensures correct pagination and user feedback (e.g., an empty list message) when the patient has no records.

Test Procedure:

- **UT-19: Pagination Test**

  – **Input:** A set of more records than fit on a single page (e.g., limit=5)

  – **Output:** UI only shows the first 5, with a "Next" option

  – **Test Derivation:** Validates that pagination controls and logic are displayed and function properly.

  – **Result:** Pass

- **UT-20: Empty Record List Test**

  – **Input:** A patient ID that returns zero records

  – **Output:** UI displays a friendly "No records found" message

  – **Test Derivation:** Ensures a good user experience for patients without any medical records.

  – **Result:** Pass

6.2.6 X-Ray Report View

Type: Automatic, Functional

Initial State: Related to the `ReportGenerationService` tests. The service prepares or fetches the final report data, which this view displays.

Test Case Derivation: Ensures that a missing or invalid image URL is handled gracefully, and that the system checks for an OpenAI API key if generating textual analysis.

Test Procedure:

- **UT-21: Basic Report Generation Test**

  – **Input:** Valid user ID, an image URL
  – **Output:** A completed X-ray report object or text
  – **Test Derivation:** Confirms that the `ReportGenerationService` is properly initialized and returns a result without error.
  – **Result:** Pass

- **UT-22: API Key Existence Test**

  – **Input:** The `OpenAI` constructor attempts to load `OPENAI_API_KEY`
  – **Output:** Non-empty string as the API key
  – **Test Derivation:** Validates environment configuration is correct and the key is not `None` or empty.
  – **Result:** Pass

- **UT-23: Constructor Argument Test**

  – **Input:** Reflection on the `ReportGenerationService` constructor
  – **Output:** Verifies it has exactly 2 parameters
  – **Test Derivation:** Ensures interface consistency and that the design remains unchanged.
  – **Result:** Pass

6.3 Software Decision Module

6.3.1 Disease Progression Model

Type: Automatic, Functional

Initial State: No direct tests shown, but the model processes historical patient data to produce a progression score.

Test Case Derivation: Ensures the model handles partial or out-of-range data gracefully and returns consistent progression metrics.

Test Procedure:

- **UT-24: Score Computation Test**

    – **Input:** A synthetic dataset of historical patient vitals or lab results
    – **Output:** A numeric "progression score" on a known scale
    – **Test Derivation:** Verifies that the model logic computes a stable and repeatable score for the same input data.
    – **Result:** Pass

- **UT-25: Boundary Data Test**

    – **Input:** Extremely high or low input values (e.g., outlier lab readings)
    – **Output:** A progression score that does not cause errors or undefined behavior
    – **Test Derivation:** Checks the robustness of the model with borderline or extreme data points.
    – **Result:** Pass

6.3.2 Disease Prediction Model

Type: Automatic, Functional

Initial State: No direct tests in the snippet. This model handles feature vectors, outputting disease probability.

Test Case Derivation: Ensures the classification is correct for typical inputs and that missing or malformed data raises appropriate errors.

Test Procedure:

- **UT-26: Probability Output Test**

    – **Input:** A well-formed feature vector with known expected disease probability
    – **Output:** Probability within a small error margin of the expected result
    – **Test Derivation:** Validates consistent classification output for typical input data.
    – **Result:** Pass

- **UT-27: Missing Fields Test**

    – **Input:** A partial feature vector omitting one or more required fields

– **Output:** An error or invalid response (e.g., `400 Bad Request`)
– **Test Derivation:** Ensures that incomplete data triggers safe handling rather than a model crash or silent failure.
– **Result:** Pass

6.3.3 Medical Record Management

Type: Automatic, Functional

Initial State: `MedicalRecordService` and `MedicalPrescriptionService` are tested using straightforward function calls (`get_paginated_record_by_userId`, `create_new_record`, etc.).

Test Case Derivation: Derived from ensuring DynamoDB-based services respond with valid data and handle edge cases (e.g., null IDs).

Test Procedure:

- **UT-28: Paginated Record Retrieval Test**

  – **Input:** A user ID and a limit (or `None`)
  – **Output:** A paginated record list, possibly empty but never `None`
  – **Test Derivation:** Verifies correct query parameters and safe handling of missing user IDs.
  – **Result:** Pass

- **UT-29: Record CRUD Tests**

  – **Input:** `create_new_record`, `update_record_by_id`, `get_record_by_id`
  – **Output:** DynamoDB updates or queries with valid responses
  – **Test Derivation:** Ensures standard create/read/update flows function without error.
  – **Result:** Pass

- **UT-30: Prescription CRUD Tests**

  – **Input:** `get_paginated_prescription_by_recordId`, `create_new_prescription`, etc.
  – **Output:** Properly formed prescription objects retrieved or updated in the database
  – **Test Derivation:** Validates that prescription data can be stored and retrieved reliably, even with null or missing fields.
  – **Result:** Pass

6.3.4 Data Persistent (AWS S3 and DynamoDB)

Type: Automatic, Functional

Initial State: The `S3PresignedURLHandler` (with a mocked S3 client) is tested to confirm it generates presigned URLs and handles file uploads correctly.

Test Case Derivation: Based on verifying that cloud-storage interactions produce valid URLs and store objects as expected, even in a mocked environment.

Test Procedure:

- **UT-31: Presigned URL Generation Test**

  – **Input:** Calls to `generate_presigned_url()`
  – **Output:** String containing expected `AWS4-HMAC-SHA256` parameters
  – **Test Derivation:** Confirms that the method returns a plausible, properly signed URL (mock signature).
  – **Result:** Pass

- **UT-32: File Upload Test**

  – **Input:** Calls to `upload_file()`
  – **Output:** A mock success message, e.g. {`"message":` `"File uploaded successfully"`, `"status":` `200`}
  – **Test Derivation:** Ensures the S3 client is invoked and the user receives a success response.
  – **Result:** Pass

7 Changes Due to Testing

7.1 Changes due to Survery Analysis

In our VNV plan it was decided that our team will conduct a survey to gather feedback on the system's usability and functionality to test our non functional requirements. After running this survery and retriving its results we made several changes to improve the user experience and address user concerns. One of the questions asked was "Which of the following non-functional aspects would most benefit from improvement?", a long list of of non-functional aspects were provided: System response, Visual Design and interface, Language support, Network reliability, Error handling and feedback, Accessibility features, Security measures, and Report presentation. The survey results showed that the most common response was Report presentation and Accessibility features. The team decided to focus on improving these areas to enhance the user experience by making the report page more user-friendly and accessible. Additionally, we improved the error handling and feedback system to provide

more informative messages and guide users through the process. Link to the survey could be found here Survey.

7.2 Changes due to Supervisor Feedback

Dr.Mehdi Moradi, our supervisor, provided feedback on the system's performance and usability. He suggested that instead of the doctor or radiologists uploading a png file of the x-ray image, the system should be able to take in a DICOM (Digital Imaging and Communications in Medicine) files as input. This would allow the system to process the image with patient data embeded within it. Additionally, he recommended that the system should be able to handle multiple images at once, as this would be more practical in a real-world clinical setting. To address these suggestions, we updated the system to accept DICOM files as input and process multiple images simultaneously. This change improved the system's performance and usability, making it more efficient and user-friendly.

7.3 Changes due to Performance Testing

After conducting performance testing, we found that the system's ability to handle multiple users simultaneously was not optimal. Our application really started to slow down around 20 concurrent users. To address this issue our team decided to allocate more resources to the backend server to improve performance. We also optimized the code to reduce processing time and improve the system's overall efficiency. These changes helped the system handle more users simultaneously and improved the user experience.

8   Automated Testing

The tests mentioned in the Unit Testing section are systematically automated to ensure code quality and reliability. To run automated tests within a developer's local environment, a developer can execute commands to build and test the project.

To ensure continuous integration and automated testing, we have implemented a comprehensive GitHub Actions workflow (Appendix C). This workflow is triggered on every pull request to the repository and performs a series of validation steps including environment setup, linting, and testing.

8.1   CI/CD Workflow Steps

Our CI/CD pipeline consists of three main jobs, each with specific responsibilities:

- **Setup Environment**: This job prepares the testing environment:
    - Checks out the repository code using `actions/checkout@v4`
    - Sets up Python 3.8 with pip caching for faster builds
    - Caches pip packages to avoid reinstalling dependencies on each run
    - Installs all required dependencies for the backend application

- **Lint and Type Check**: This job performs static code analysis:
    - Runs Flake8 to check for code style issues and potential bugs
    - Generates a linting report for review
    - Fails the pipeline if critical errors are detected
    - Uploads linting results as artifacts for later inspection

- **Run Tests**: This job executes the pytest test suite:
    - Runs all tests with pytest
    - Generates code coverage reports in multiple formats
    - Produces JUnit XML output for test reporting
    - Uploads test results and coverage reports as artifacts

8.2   Local Testing Environment

For local development, developers can run the same tests that are executed in the CI/CD pipeline:

- To run linting checks: flake8 src test

- To run unit tests: pytest test

- To run tests with coverage: pytest test –cov=src

8.3 Test Results and Reporting

The CI/CD pipeline generates several key artifacts for quality assurance:

- **Linting Reports**: Flake8 reports identify code style issues and potential bugs

- **Test Results**: JUnit XML reports provide detailed test execution results

- **Coverage Reports**: XML and terminal outputs show which parts of the code are tested

All test results are uploaded as artifacts in the GitHub workflow, making them easily accessible for review. Additionally, we store test results in Amazon S3 for historical tracking and reference.

8.4 Integration with Pull Request Workflow

Our CI/CD pipeline is tightly integrated with the GitHub pull request process:

- Tests are automatically triggered when a pull request is created

- Merging is blocked until all tests pass successfully

- Test results and code coverage metrics are available for review

- After merging into the main branch, additional validation ensures code stability

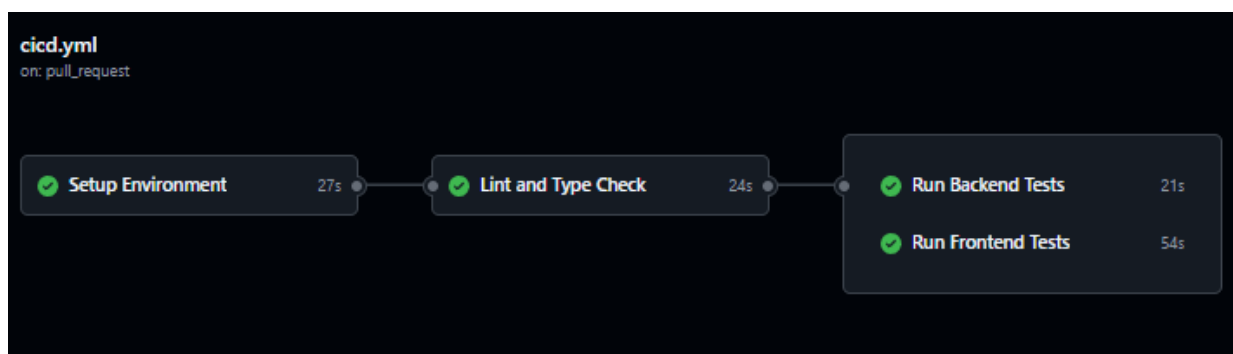Here we is a viulaization of the CI/CD stages within our capstone project:

Figure 1: CI/CD pipeline stages

9 Trace to Requirements

Unit Test ID	Requirement ID
UT-1	NFR-LF1, NFR-OE2
UT-2	FR1, FR7
UT-3	FR9
UT-4	FR1, FR2
UT-5	FR1, NFR-PR2
UT-6	NFR-PR2
UT-7	FR3, FR11
UT-8	FR3, FR5, NFR-PR1, NFR-PR3
UT-9	FR4, FR11
UT-10	FR4, NFR-PR1, NFR-PR3
UT-11	NFR-SR1, NFR-LR1
UT-12	NFR-SR1, NFR-LR1
UT-13	NFR-LF1, NFR-LF2
UT-14	NFR-UH1
UT-15	NFR-LF1, NFR-LF2
UT-16	NFR-UH1
UT-17	FR8
UT-18	FR4
UT-19	NFR-UH1
UT-20	FR7
UT-21	FR6, NFR-PR1, NFR-HS2
UT-22	FR6
UT-23	FR6, NFR-MS1
UT-24	FR4
UT-25	FR4
UT-26	FR3, FR10, NFR-PR3
UT-27	FR3, NFR-LR2
UT-28	FR7
UT-29	FR2, FR7, FR9, NFR-MS1, NFR-LR2
UT-30	FR2, FR7, FR9
UT-31	FR1, NFR-SR1
UT-32	FR1, FR7, NFR-SR1

Table 29: Trace from Unit Tests to Requirements (Jodeiri et al., 2024)

10 Trace to Modules

Unit Test	Modules Tested
UT-1	M1, M2
UT-2	M1, M13, M14
UT-3	M13
UT-4	M1, M6
UT-5	M2
UT-6	M2
UT-7	M3, M12
UT-8	M3, M12
UT-9	M4, M11
UT-10	M4, M11
UT-11	M5
UT-12	M5
UT-13	M6
UT-14	M6
UT-15	M7
UT-16	M7
UT-17	M8
UT-18	M8
UT-19	M9
UT-20	M9
UT-21	M10
UT-22	M10
UT-23	M10
UT-24	M11
UT-25	M11
UT-26	M12
UT-27	M12
UT-28	M13
UT-29	M13
UT-30	M13
UT-31	M14
UT-32	M14

Table 30: Traceability Matrix: Unit Tests to Modules (Jodeiri et al., 2024)

11 Code Coverage Metrics

The following code Coverage analysis was obtained from the automated tests that were run on the CI/CD pipeline:

```
51  ----------- coverage: platform linux, python 3.8.18-final-0 -----------
52  Name                                                               Stmts   Miss  Cover
53  ---------------------------------------------------------------------------------------
54  src/backend/runtime/__init__.py                                       16      0   100%
55  src/backend/runtime/config.py                                         17      3    82%
56  src/backend/runtime/data/aws_cognito.py                              142     44    69%
57  src/backend/runtime/data/aws_dynamodb/base_service.py                 36     25    31%
58  src/backend/runtime/data/aws_dynamodb/medical_prescription_service.py  19     10    47%
59  src/backend/runtime/data/aws_dynamodb/medical_record_service.py       21     11    48%
60  src/backend/runtime/data/aws_s3.py                                    31     15    52%
61  src/backend/runtime/data/prediction_services.py                      37     21    43%
62  src/backend/runtime/data/report_generation_service.py                21      1    95%
63  src/backend/runtime/logger.py                                        12      0   100%
64  src/backend/test/__init__.py                                          0      0   100%
65  src/backend/test/conftest.py                                         24      2    92%
66  src/backend/test/mocks/mock_identity_provider.py                     34      0   100%
67  src/backend/test/mocks/mock_prediction_service.py                     7      0   100%
68  src/backend/test/mocks/mock_report_generation.py                     23      0   100%
69  src/backend/test/mocks/mock_s3.py                                    13      0   100%
70  src/backend/test/test_aws_conito.py                                  68      0   100%
71  src/backend/test/test_aws_s3.py                                      13      0   100%
72  src/backend/test/test_report_generation_service.py                   27      0   100%
73  ---------------------------------------------------------------------------------------
74  TOTAL                                                               561    132    76%
75  Coverage XML written to file coverage.xml
76
77  ======================= 12 passed, 2 warnings in 0.31s =======================
```

Figure 2: Code Coverage

Our codebase achieved an overall test coverage of 76% across 561 statements, with 132 statements not covered by our test suite. Several components demonstrate excellent test coverage, with 8 files reaching 100% coverage, including core modules like the logger implementation and various test mocks. The report generation service also shows strong coverage at 95%.

Areas requiring additional test attention are primarily in the AWS DynamoDB services layer, with the base service at 31% coverage and medical-related services ranging from 43-52%. These lower coverage percentages reflect the complexity of fully testing cloud service integrations in isolation. The AWS Cognito authentication module also presents opportunities for improvement at 69% coverage.
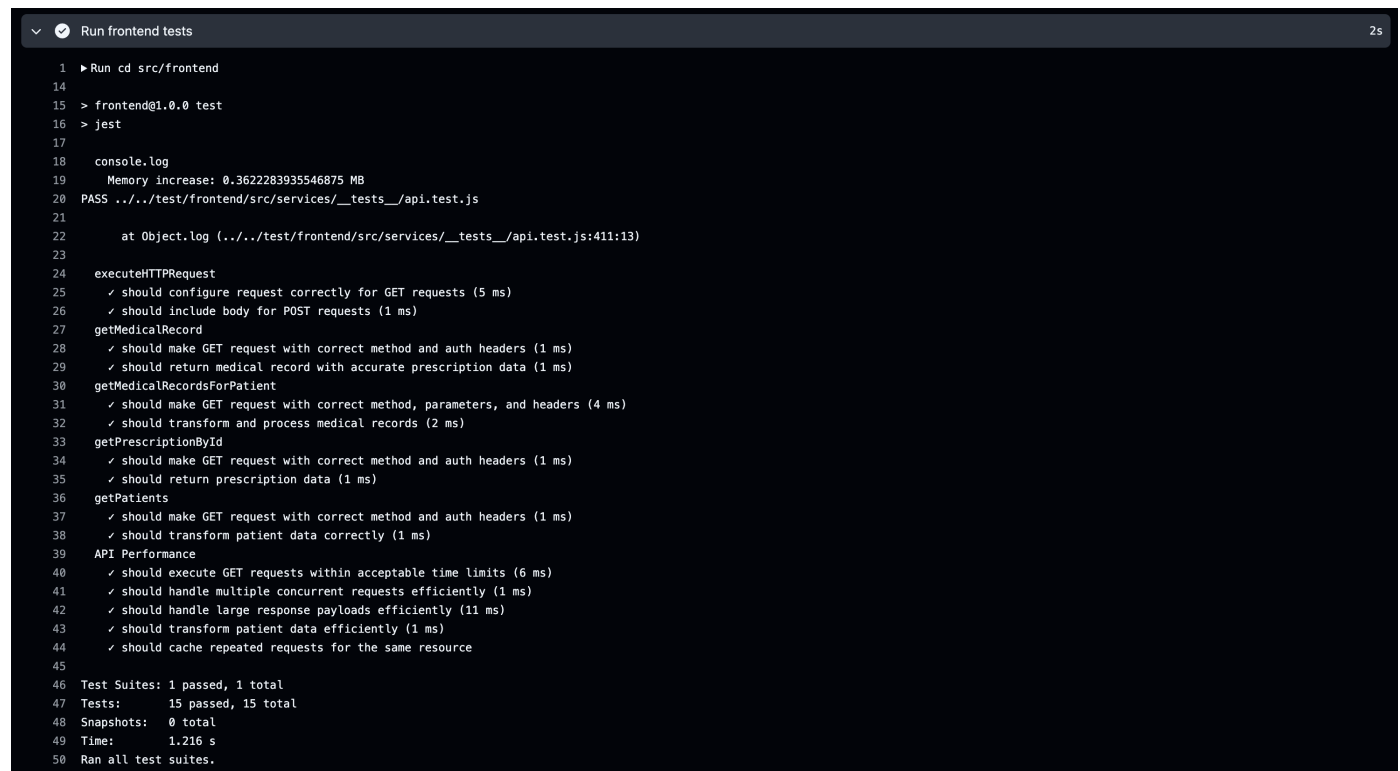
It should be noted that several components with lower test coverage rates directly interact with third-party services such as AWS DynamoDB, AWS S3, AWS Cognito, and the OpenAI

API. These integrations present unique testing challenges:

- External service calls require complex mocking frameworks to simulate responses

- Some edge cases are difficult to reproduce without accessing the actual service

- Authentication flows with services like Cognito involve multi-step processes that are challenging to test in isolation

- API contracts may change with service updates, particularly with evolving services like OpenAI

While we have implemented mocks for these services in our testing environment, achieving complete coverage remains challenging without introducing tests that may provide false confidence. Our testing strategy prioritizes reliable validation of core business logic while acknowledging the inherent limitations in testing third-party integrations.

Additionally, our frontend testing suite achieved comprehensive coverage across React components and utilities, with particular focus on critical user interface elements. The tests were implemented using Jest and React Testing Library.The following were the frontend tets that we perforemd:


```
  ✓  Run frontend tests                                                                    2s
   1  ▶ Run cd src/frontend
  14
  15  > frontend@1.0.0 test
  16  > jest
  17
  18    console.log
  19      Memory increase: 0.3622283935546875 MB
  20  PASS ../../test/frontend/src/services/__tests__/api.test.js
  21
  22        at Object.log (../../test/frontend/src/services/__tests__/api.test.js:411:13)
  23
  24    executeHTTPRequest
  25      ✓ should configure request correctly for GET requests (5 ms)
  26      ✓ should include body for POST requests (1 ms)
  27    getMedicalRecord
  28      ✓ should make GET request with correct method and auth headers (1 ms)
  29      ✓ should return medical record with accurate prescription data (1 ms)
  30    getMedicalRecordsForPatient
  31      ✓ should make GET request with correct method, parameters, and headers (4 ms)
  32      ✓ should transform and process medical records (2 ms)
  33    getPrescriptionById
  34      ✓ should make GET request with correct method and auth headers (1 ms)
  35      ✓ should return prescription data (1 ms)
  36    getPatients
  37      ✓ should make GET request with correct method and auth headers (1 ms)
  38      ✓ should transform patient data correctly (1 ms)
  39    API Performance
  40      ✓ should execute GET requests within acceptable time limits (6 ms)
  41      ✓ should handle multiple concurrent requests efficiently (1 ms)
  42      ✓ should handle large response payloads efficiently (11 ms)
  43      ✓ should transform patient data efficiently (1 ms)
  44      ✓ should cache repeated requests for the same resource
  45
  46  Test Suites: 1 passed, 1 total
  47  Tests:       15 passed, 15 total
  48  Snapshots:   0 total
  49  Time:        1.216 s
  50  Ran all test suites.
```

Figure 3: Frontend Tests

References

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2021.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

Reza Jodeiri, Kelly Deng, Ayman Akhras, Nathan Luong, and Patrick Zhou. System requirements specification. https://github.com/RezaJodeiri/CXR-Capstone/blob/main/docs/SRS/SRS.pdf, 2024.

Appendix — Reflection

What went well while writing this deliverable?

The modular design of our system allowed us to create clear and well-structured unit tests. Each test targeted its own specific business logic and did not interfere with others, which helped us achieve more comprehensive unit test coverage. By following SOLID principles and maintaining a clear separation of concerns, we designed precise and maintainable tests that effectively targeted individual components. This modular approach enabled us to create highly focused test cases, minimizing dependencies and improving fault isolation. Leveraging mock services and data allowed us to systematically verify each component and separate concerns, ensuring that each module functioned correctly in isolation with its expected functionalities. Additionally, these tests were seamlessly integrated into our CI/CD pipeline, enabling automated validation at every stage of development. This approach streamlined the testing process and also improved reliability by allowing us to simulate various scenarios and edge cases, ensuring consistent performance across deployments.

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

One major challenge was handling unexpected deviations from the VnV Plan. Some tests and validation activities required adjustments because real-world implementation differed from our initial assumptions. To address this, we removed certain tests and modified others to better fit the project scope. Another challenge we faced was testing non-functional requirements (NFRs), particularly in gathering sufficient feedback from qualified testers. To address this, our team implemented an external testing approach using a Google Form to collect responses. However, we encountered difficulties in obtaining an adequate number of responses from eligible testers, such as medical professionals as our project was not in a collaboration with hospitals / medical services. The limited participation made it challenging to thoroughly validate certain NFRs, such as usability and performance in real-world scenarios. Additionally, creating unit tests for the code was a significant task. However, by effectively distributing the workload among team members, we successfully developed and passed all necessary tests.

Which parts of this document stemmed from speaking to your client(s) or a proxy (e.g., your peers)? Which ones were not, and why?

Some aspects of this document stemmed from discussions with our proxies (peers), particularly regarding non-functional requirements testing as some of them couldn't be tested directly via unit tests. (usability, user experience, and looks and feels). While unit tests focus on validating specific functions in isolation, they do not capture real-world interactions, user satisfaction, or how well the system meets stakeholder expectations. Surveys allowed us to gather qualitative and quantitative feedback from relevant users, helping us assess aspects like ease of use, efficiency, and overall satisfaction. Our unit tests were primarily designed to validate the correctness, reliability, and functionality of individual components

in isolation. They focused on ensuring that each module adhered to its expected behavior based on predefined business logic, rather than incorporating input from peers or end-users. However, not all sections were based on client input. Technical implementation details, such as internal testing methodologies and the choice of specific testing frameworks, were determined by the development team. This distinction exists because clients typically focus on high-level functionality and usability, while the development team makes technical decisions regarding implementation and testing strategies.

In what ways was the Verification and Validation (VnV) Plan different from the activities that were actually conducted for VnV?

Differences Between the VnV Plan and VnV Report

- In the VnV Plan, we initially outlined the testing for functional and non-functional requirements. However, the project was not fully mature and due to this the VnV Report had to be adjusted to reflect the actual testing conducted. This was shown to be evident through the removal of NFRs and FRs while new ones being added.

- The original VnV Plan assumed static test data for validation, but as development progressed, we realized that model performance varied significantly across different datasets. The VnV Report reflects additional real-world test cases added to evaluate model robustness across various demographic and clinical conditions. This was also later changed in the VNV plan to match.

- Initially planned as a basic text output, this feature was expanded in the VnV Report to include structured templates for easier clinical use.

Reasons for Deviations

These deviations were necessary due to:

- Technical constraints, particularly in model interpretability and performance generalization.

- Regulatory considerations, requiring expanded fairness and bias assessments beyond the original scope.

- Stakeholder feedback, which emphasized the need for more real-world validation scenarios. By doctors and radiologist to test the system and what they would want to see in our application.

- Resource limitations which led to adjustments in testing priorities and methodologies.

Lessons Learned for Future VnV Planning

While deviations were unavoidable, they provided valuable insights for improving future VnV planning. To better anticipate such changes, our team could:

- Planning during the early maturity of the project is difficult and should be revisited as the project matures.

- Early real-world testing is critical – testing only on predefined datasets missed real-world challenges, such as low-quality scans.

- Explainability should be prioritized from the start – assuming simple feature visualization was sufficient led to delays in meeting doctor expectations.

- Load testing should be iterative – scalability concerns were only identified late in development, making fixes more complex.

- Stakeholder feedback must be integrated earlier – doctors provided critical usability insights only after testing, when design changes were harder to implement.

Conclusion

In conclusion, the Verification and Validation process for our project has been a comprehensive and iterative journey. We have successfully evaluated both functional and non-functional requirements, ensuring that our system meets the high standards expected in a clinical setting by removing unnecessary requirements and adding new ones. The deviations from the original VnV Plan aided to address our challenges, technical constraints, and stakeholder feedback. These adjustments have refined our system, making it more robust, user-friendly, and aligned with regulatory requirements. The lessons learned from this process highlight the importance of early and continuous testing, stakeholder engagement, and flexibility in planning. By incorporating these insights into future projects, we can improve our VnV strategies and deliver even more reliable and effective solutions. Hence, the VnV activities have validated the performance, usability, and security of our system, providing confidence in its readiness for deployment in real-world healthcare environments.

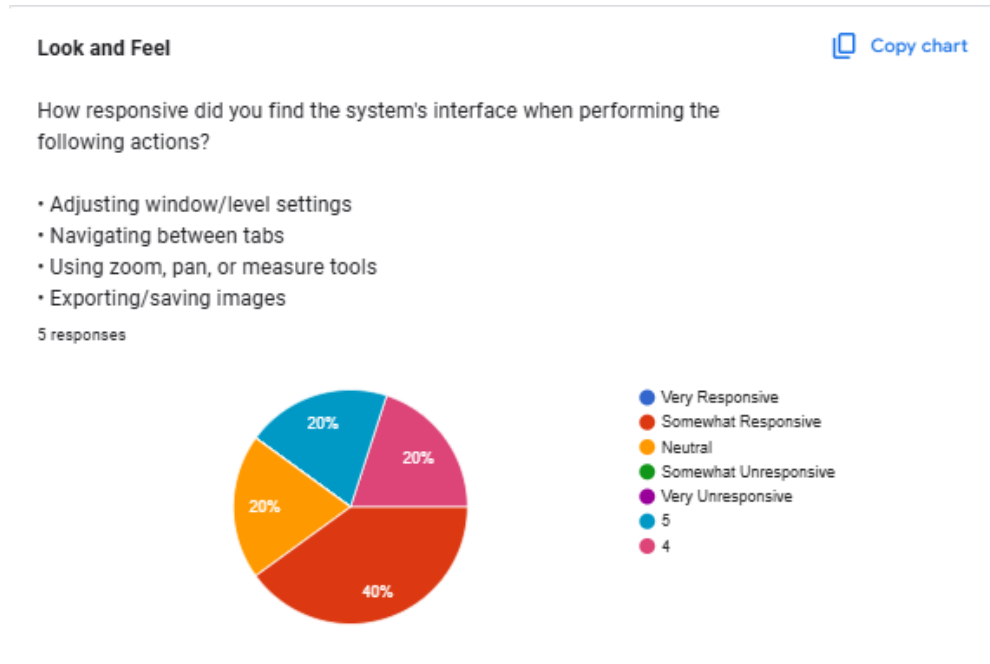# Appendix A — Survey Results (NFR quality 1)
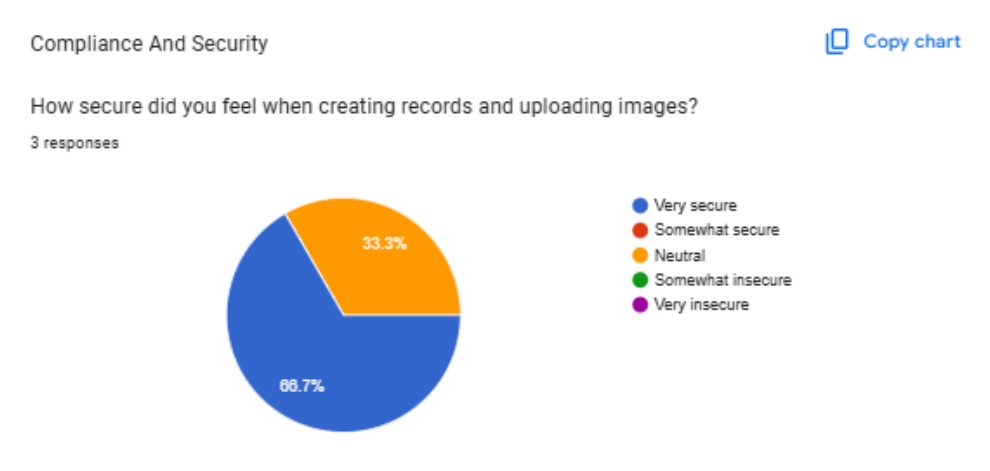

Figure 4: Survey results for Question 1

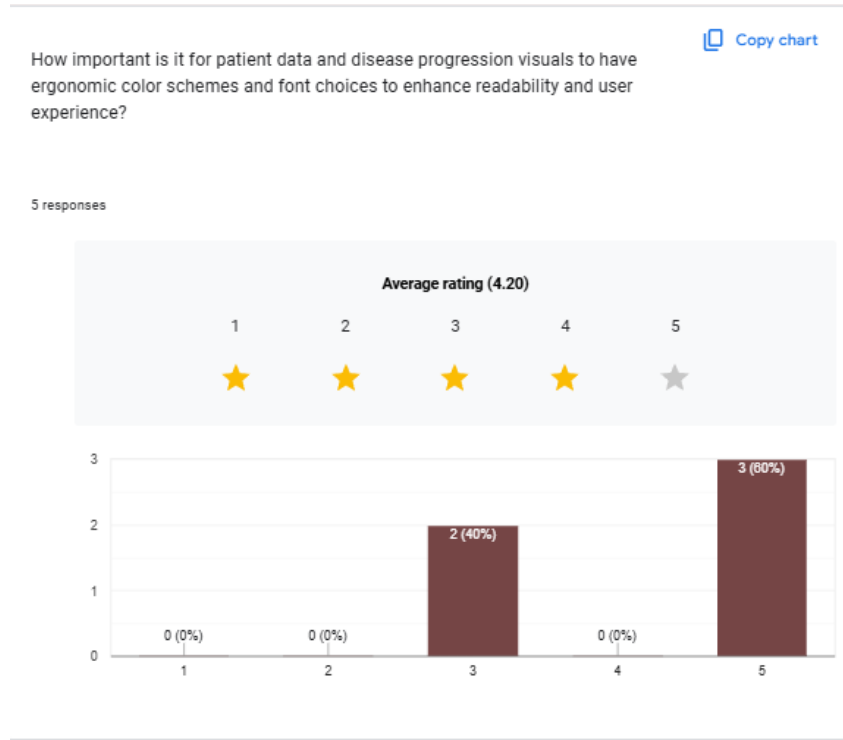Figure 5: Survey results for Question 2
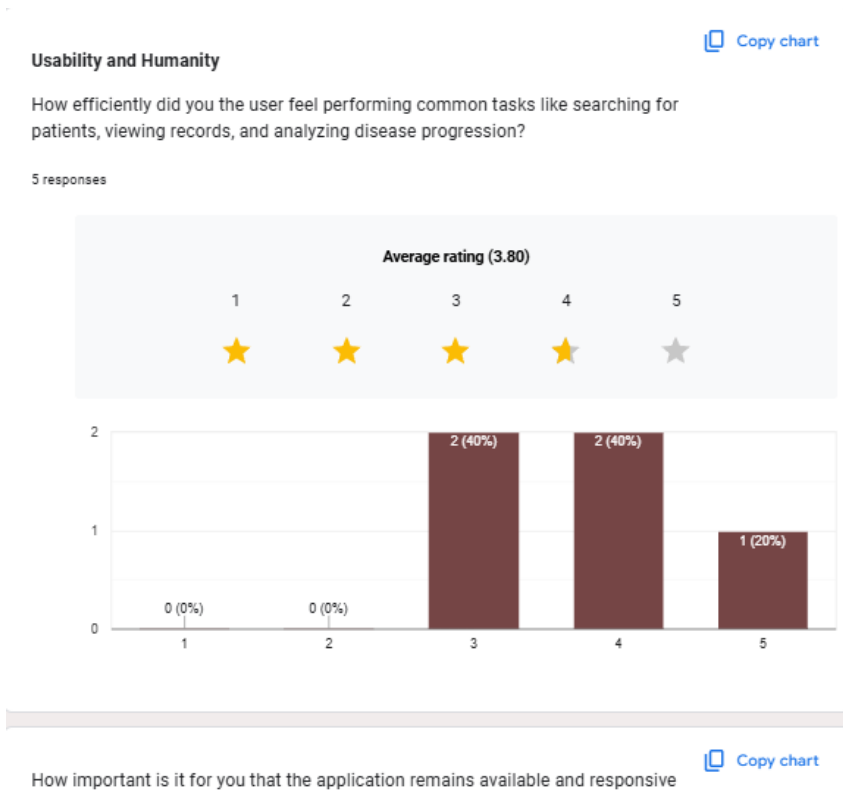
Figure 6: Survey results for Question 3

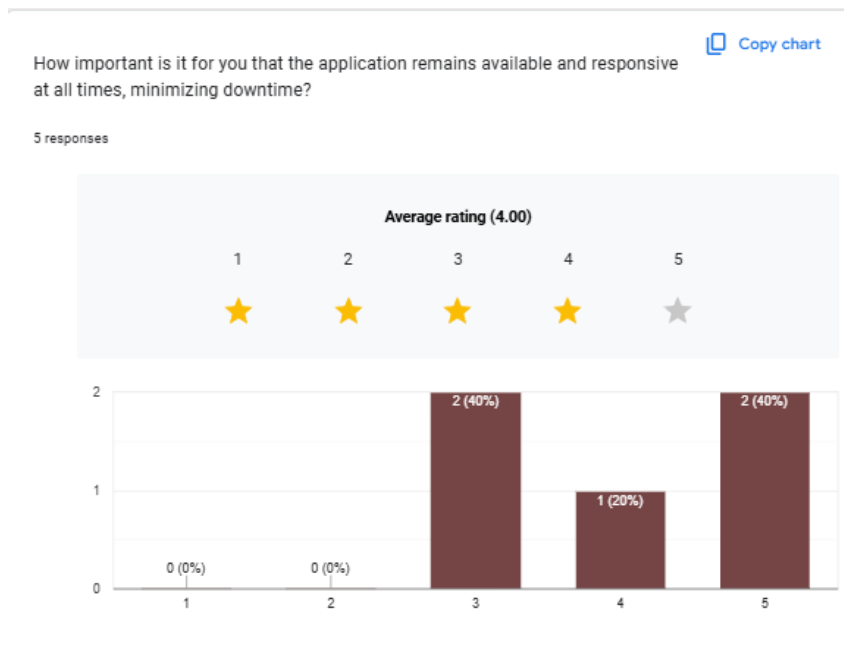Figure 7: Survey results for Question 4

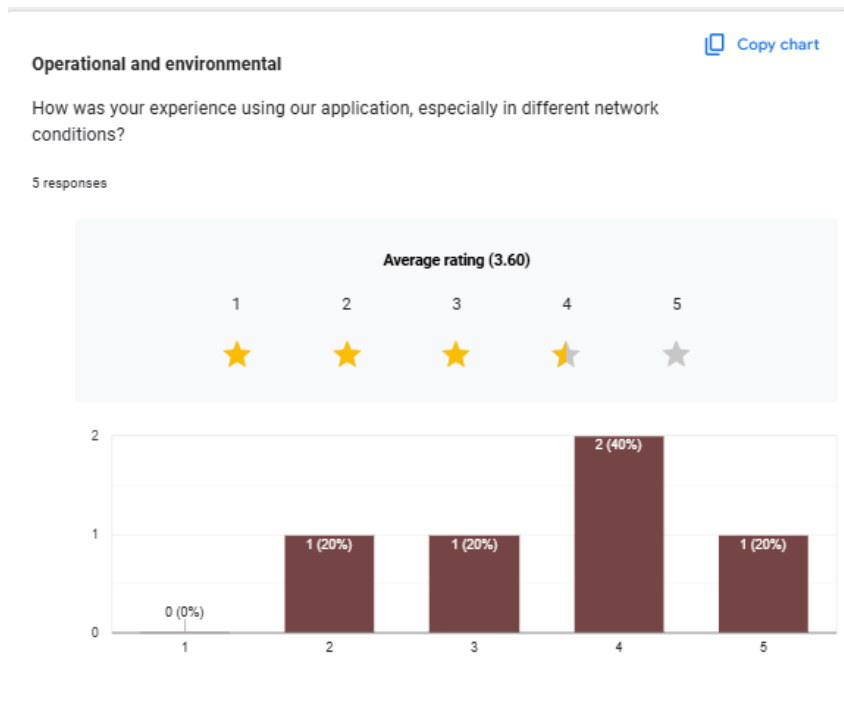Figure 8: Survey results for Question 5
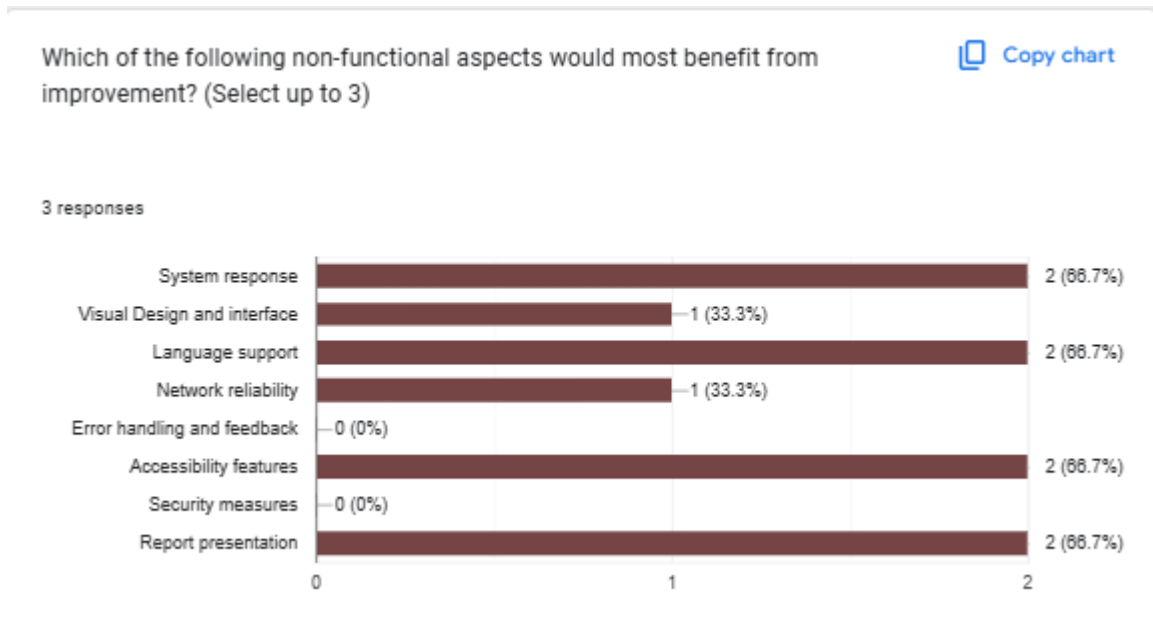

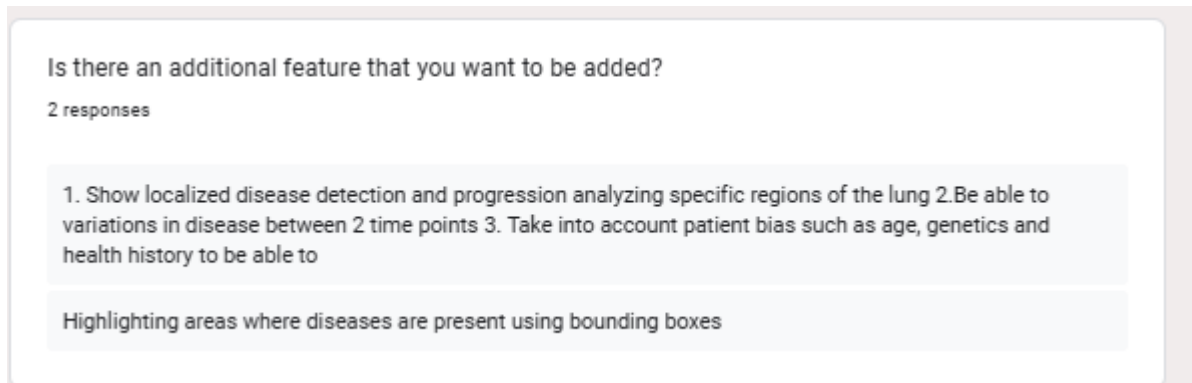Figure 9: Survey results for Question 6

Figure 10: Survey results for Question 7

Figure 11: Survey results for Question 8

Appendix B — Performance Test (NFR quality 2)
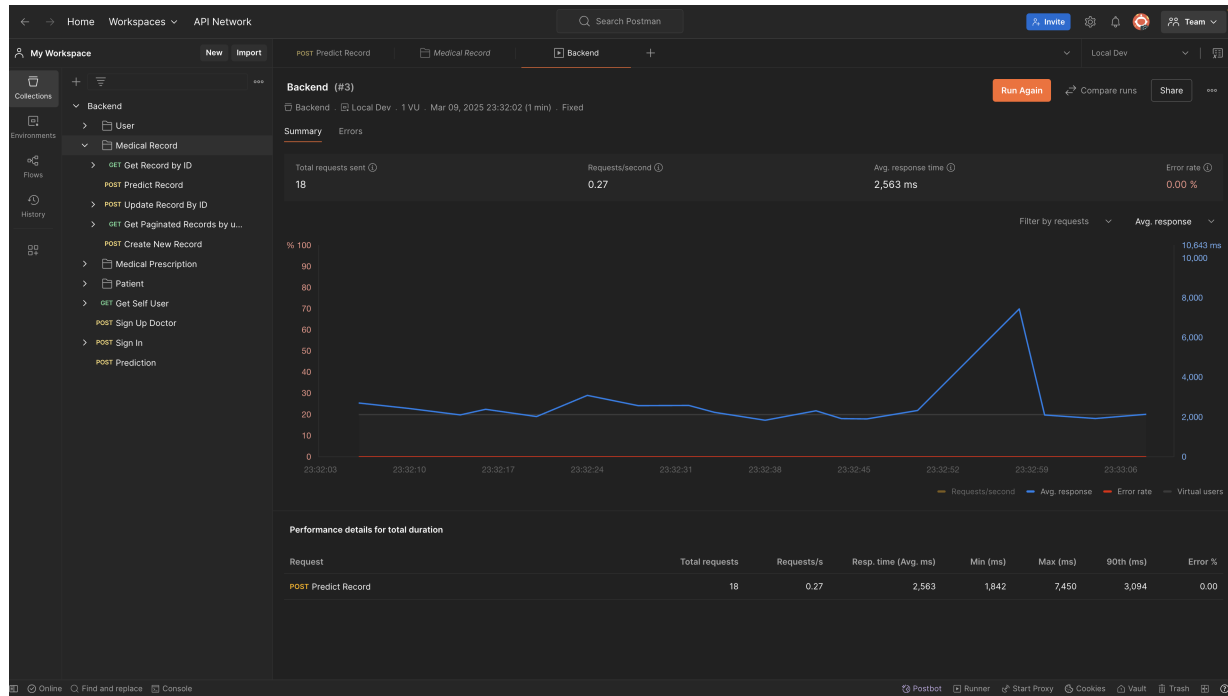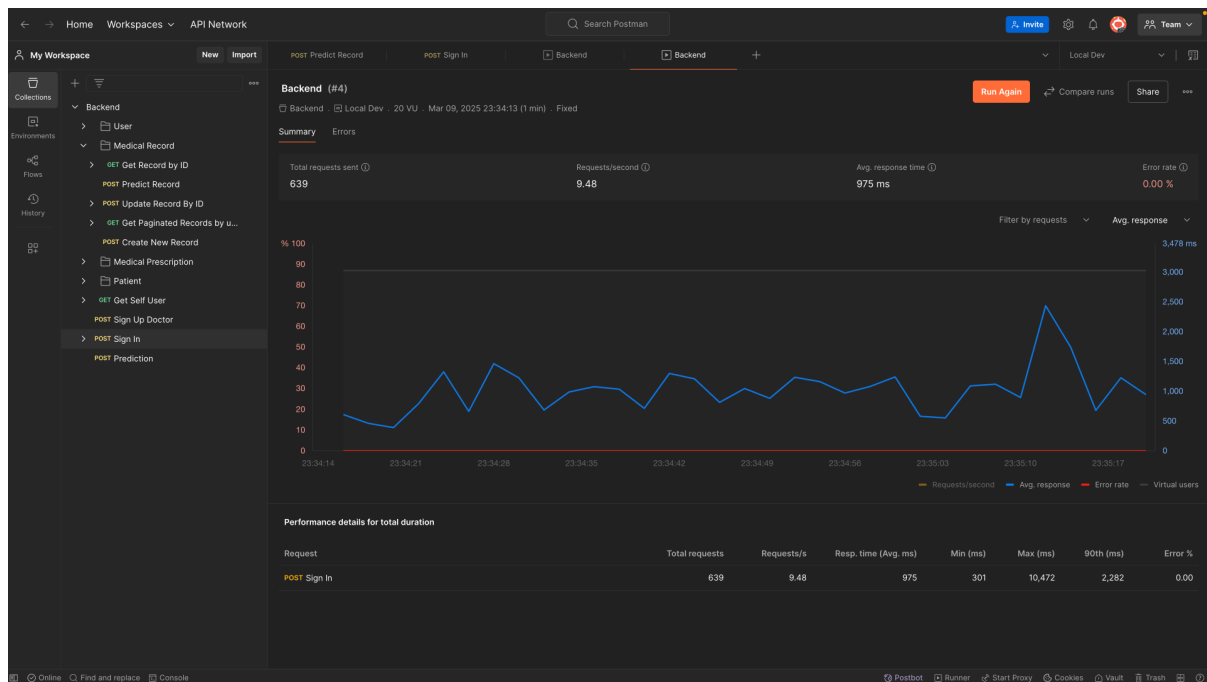

Figure 12: Survey results for Question 8

Figure 13: Survey results for Question 8