



Master Test Plan

Version 1.0

Group 14 (Project ID : 2)

210329E - LAKSARA K.Y.

210333K - LAKSHAN P.D.

210343P – LIYANAGE I.V.S.

In21-S5-CS3501 Data Science and Engineering Project

Department of Computer Science and Engineering University of Moratuwa

Table of Contents

1. Evaluation Mission and Test Motivation.....	4
2. Target Test Items.....	5
2.1 User Interfaces:.....	5
2.2 Data Models and Components:.....	5
2.3 Prediction Models:.....	6
2.4 Security and Authentication:.....	6
2.5 Performance and Load Testing:.....	6
2.6 External APIs and Integrations:.....	7
3. Test Approach.....	7
3.1 Testing Techniques and Types.....	7
3.1.1 Data and Database Integrity Testing.....	7
3.1.2 Function Testing.....	9
3.1.3 User Interface Testing.....	12
3.1.4 Performance Profiling.....	13
3.1.5 Load Testing.....	15
3.1.6 Security and Access Control Testing.....	18
3.1.7 Failover and Recovery Testing.....	20
3.1.8 Configuration Testing.....	23
3.1.9 ML/DS Testing.....	24
• Melanoma Detection Model.....	24
• Skin Disease Detection Model.....	26
4. Deliverables.....	28
4.1 Deliverable: Functional Testing (Integration Testing for the Backend).....	28
4.1.1 User Update Tests:.....	29
4.1.2 Google Login Tests:.....	29
4.1.3 Doctor Retrieval Tests:.....	29
4.1.4 Report Management Tests:.....	29
4.1.5 Signup Tests:.....	30
4.2 Database Testing Deliverables.....	30
4.2.1 Test 1: Find a Unique Report.....	30
4.2.2 Test 2: Find All Reports.....	31
4.2.3 Test 3: Find the Newest Report.....	31
4.2.4 Test 4: Find All Users.....	31
4.3 Frontend Testing Deliverables.....	34
4.3.1 Integration Testing:.....	34
4.3.2 End-to-End (E2E) Testing:.....	34
4.4 Deliverables for ML/DS Models.....	37

4.4.1 Deliverables for Melanoma Detection Model.....	37
• Train and Validation Accuracy Plot.....	37
• Train and Validation Loss Plot.....	37
• Evaluation Metrics.....	38
• Classification Report.....	38
• ROC Curve and AUC Score.....	38
• Confusion Matrix.....	38
• Grad-CAM and Grad-CAM++ Output.....	39
4.4.2 Deliverables for Skin Disease Detection Model.....	39
• Model Training and Validation Metrics.....	39
• Evaluation Metrics.....	40
• Confusion Matrix.....	40
• Grad-CAM and Grad-CAM++ Visualizations.....	40
5. Risks, Dependencies, Assumptions, and Constraints.....	41
6. References.....	42

1. Evaluation Mission and Test Motivation

This Master Test Plan focuses on the test approach for the **HealthBot+** AI-powered web application, designed for the early detection and management of skin diseases. The mission of the system is to provide users and healthcare professionals with a user-friendly platform that integrates advanced image analysis, AI models, and chatbot functionalities for accurate diagnostic support. To ensure the highest quality and user experience, the system must undergo thorough testing across all aspects.

HealthBot+ handles sensitive medical information, and any bug or security loophole could have severe consequences for users. As a result, both the user interface and system logic must meet stringent quality standards to deliver reliable and secure diagnostics. The testing process must be well-organized and cover every layer of the system, ensuring it delivers what is expected and complies with healthcare regulations. The application is tested on unit testing, integration testing and end-to-end testing.

The main objectives of the testing process include:

- **Bug Detection:** The goal is to identify and fix as many bugs as possible before deploying HealthBot+ to users. Special attention will be given to identifying bugs that affect system confidentiality and integrity, as the platform deals with sensitive medical data such as patient history and diagnosis reports.
- **Design and Implementation Flaws:** Testing will aim to find any problems in the system's design or implementation that could impact quality or fail to meet user requirements. These will be corrected before final deployment.
- **Risk Identification:** Potential risks associated with the project will be identified, tested, and addressed before they materialize. Risks such as incorrect diagnoses or breaches of data security will be forecasted and mitigated early in the process.
- **Quality and Performance Standards:** The system will be tested to ensure it meets both functional and non-functional requirements, as well as the quality standards expected of healthcare applications. Performance testing and user acceptance testing will ensure that HealthBot+ delivers accurate and timely results.
- **Stakeholder Satisfaction:** A user acceptance test will be conducted to confirm that the system meets the expectations of stakeholders, ensuring the desired level of quality is present in the final product.
- **Compliance with Legal and Process Mandates:** The system must adhere to legal requirements related to data privacy and healthcare regulations. A thorough legal compliance test will be carried out to resolve any issues before deployment.
- **Beta User Feedback:** During user acceptance testing, stakeholders will be informed that the system is a Beta version, and their feedback will be collected to make necessary adjustments before full release.

By carrying out this comprehensive testing plan, HealthBot+ will be prepared to deliver accurate, reliable, and secure diagnostic tools to users and healthcare professionals.

2. Target Test Items

Since this application is a **skin disease detection web application**, it is crucial to ensure high accuracy, reliability, security, and performance. The system will handle sensitive user data and medical reports, so robust data processing, security features, and clear user interfaces are required. The listing below identifies those test items: software, hardware, and supporting product elements that have been identified as targets for testing.

2.1 User Interfaces:

These are the views of the system that users will interact with to upload images, view predictions, and receive feedback. The user interface should be tested for responsiveness, ease of navigation, and compatibility across devices and browsers. Specific testing will focus on:

- **Page load times:** Ensure the interface loads quickly, even when large image files are uploaded.
- **Cross-browser compatibility:** Validate that the UI works across different browsers (Chrome, Firefox, Safari, etc.).
- **Accessibility:** Test that the interface is accessible for users with disabilities, complying with web accessibility standards.

Testing tool: **Selenium** will be used for automated UI testing to verify usability, page load times, and general interface consistency across browsers and devices.

2.2 Data Models and Components:

This is the core of the system as it handles sensitive medical data and patient information. The system must store image data and patient data securely in the database, as well as ensure the models are updated for accurate predictions. Key aspects include:

- **Database performance:** Test for efficient data retrieval and storage, ensuring minimal delay when querying data.
- **Data integrity and durability:** Ensure no data loss and the ability to recover data from backups.
- **Scalability:** Test that the system can handle an increasing number of patients and large datasets.

Testing tool: **MySQL Workbench** and **Postman** will be used to test the API endpoints and database performance.

2.3 Prediction Models:

These are the machine learning models responsible for skin disease detection based on images. It is vital that these models are rigorously tested to ensure accuracy and reliability. Key testing items include:

- **Model performance:** Validate the accuracy, precision, recall, and F1-score of the predictions.
- **Stress testing:** Test the model's performance when handling large-scale datasets and images of different resolutions.
- **Model explainability:** Ensure that model predictions are accompanied by explanations, using tools like LIME and Grad-CAM to validate the model's interpretability.

Testing tool: **TensorFlow/Keras** testing suite, **Jupyter Notebooks**, and custom scripts for accuracy validation.

2.4 Security and Authentication:

Since the system deals with patient data, security must be paramount. The security testing will ensure that patient information is protected, and only authorized users have access to specific parts of the application. Focus areas include:

- **User authentication and authorization:** Validate the security of login and access control features.
- **Data encryption:** Ensure that sensitive data (such as patient details and reports) are securely encrypted.
- **Penetration testing:** Simulate attacks to test the robustness of the application against vulnerabilities.

Testing tool: **OWASP ZAP** for penetration testing and **Postman** for API security checks.

2.5 Performance and Load Testing:

The system must be able to perform optimally under heavy load, as it may be used by multiple users simultaneously. This includes testing the application's performance under high traffic and ensuring that response times are within acceptable limits.

- **Concurrent user load:** Test how the system behaves with multiple users accessing the application simultaneously.
- **Response times:** Validate the response time for predictions and report generation under different workloads.

Testing tool: **Apache JMeter** will be used for performance testing, simulating different loads and analyzing the system's behavior.

2.6 External APIs and Integrations:

The system relies on external APIs (e.g., Firebase for image storage and authentication services). These APIs must be tested to ensure they function correctly and handle errors gracefully.

- **Firebase storage:** Test the image upload, retrieval, and deletion processes.
- **External model API:** Ensure proper communication between the application and external model services.

Testing tool: **Postman** for API integration testing, along with **Firebase emulator** for local testing of Firebase services.

3. Test Approach

The test approach presents the recommended strategy for designing and implementing the required tests for HealthBot+, an AI-powered skin disease diagnosis and management system. The approach includes testing the Flask backend, React frontend, MongoDB database, and Firebase integration to ensure data integrity, functionality, performance, and security.

3.1 Testing Techniques and Types

3.1.1 Data and Database Integrity Testing

The databases and their processes should be tested as an independent subsystem. In this case, MongoDB was used as the primary database system, and testing was conducted without the target-of-test's user interface. Apache JMeter (version 5.6.3) was employed for database performance testing, focusing on metrics such as query performance, concurrent threads, and system load. Multiple listeners were utilized to analyze and monitor the database behavior under various conditions.

Technique Objective	Test MongoDB's performance and behavior under various workloads, focusing on identifying incorrect target behavior, slow queries, or data corruption during concurrent access.
Technique	<p>Database Access: Each MongoDB query was tested by running multiple test cases, including valid and invalid data retrieval and insertion.</p> <p>Performance Tests: Using Apache JMeter, tests were configured to simulate multiple threads (concurrent requests), ramp-up</p>

	<p>periods, and loop counts to measure how well the database handles load.</p> <p>Data Inspection: The data was inspected after each test to ensure that it was populated or retrieved correctly. Listeners like Result Tree, Summary Report, Aggregate Report, and Response Time Graph were used to validate the system's behavior under load.</p> <p>Seeding the Database: Different data inputs, both valid and invalid, were seeded to the MongoDB database, testing how well it handled requests, insertions, and data validation across different user requests</p>
Oracles	<p>Performance Monitoring: Listeners in Apache JMeter provided metrics for determining whether the database behaved as expected under various conditions.</p> <p>Data Integrity Check: Validation of retrieved data was performed using the Result Tree listener, ensuring that correct records were retrieved based on the queries made.</p> <p>Concurrency Handling: Monitoring the database's ability to handle concurrent access (multiple threads) was achieved using Aggregate Report and Graph Results listeners.</p> <p>Error Detection: By running JMeter tests with invalid data, the system's response to errors, such as invalid queries or malformed requests, was tested to ensure proper error handling.</p>
Required Tools	<p>Apache JMeter (5.6.3): To simulate database loads, perform concurrency tests, and measure performance.</p> <p>JMeter Listeners: Result Tree, Summary Report, Aggregate Report, Graph Results, Response Time Graph.</p> <p>MongoDB Database Utilities: For database management and direct data inspection.</p> <p>Custom Groovy Scripts: For MongoDB queries and updates within the JMeter environment.</p>
Success Criteria	<p>All MongoDB queries are correctly executed, with no data corruption or retrieval issues observed across all concurrent users.</p> <p>Testing covers various scenarios like multiple concurrent threads, varying ramp-up periods, and loop counts.</p>

	<p>Apache JMeter listeners show that the system remains performant and stable under different load levels.</p> <p>Data integrity is maintained across all database operations, ensuring correct data retrieval and insertion.</p>
Special Considerations	<p>The MongoDB cluster used for testing was connected to a cloud environment (MongoDB Atlas). Network conditions were considered during testing.</p> <p>Testing involved using small datasets for better visibility of performance bottlenecks.</p> <p>Manual inspection of MongoDB data was done through monitoring tools to validate any test failures detected by JMeter.</p>

3.1.2 Function Testing

Function testing of the skin disease web application focuses on any requirements that can be traced directly to its use cases, business functions, and business rules. The goals of these tests are to verify the proper acceptance, processing, and retrieval of data, as well as the appropriate implementation of business rules. This type of testing follows black-box techniques, which means that internal processes are verified by interacting with the system through the Graphical User Interface (GUI) and analyzing the output or results. The following table outlines the recommended testing approach for each functionality in the application.

Technique Objective	<p>Exercise the core functionality of the skin disease web app, including navigation, data entry, processing, and retrieval, to observe and log the system's behavior.</p> <p>Each functionality is tested based on its business logic to ensure that it meets the functional requirements such as:</p> <p>User authentication: Validating login/logout for patients and doctors.</p> <p>Report generation and management: Doctors handling patient reports, adding comments, and updating report status.</p> <p>Profile management: Users updating personal details and uploading profile pictures.</p>
---------------------	---

	<p>Skin disease prediction: Patients submitting images for analysis, receiving results, and doctors reviewing model predictions.</p>
Technique	<p>Execute the following for each use case scenario's individual flows, functions, and features:</p> <p>Valid Data Testing: Ensure that the expected results occur when valid data is used (e.g., successful login with correct credentials, correct disease prediction based on submitted image).</p> <p>Invalid Data Testing: Verify that appropriate error or warning messages are displayed when invalid data is used (e.g., incorrect passwords, incomplete report submissions, or unsupported file types for image uploads).</p> <p>Business Rule Enforcement: Confirm that the system applies all relevant business rules correctly (e.g., doctors are the only ones authorized to update reports, patients can only access their own reports).</p> <p>Form Validation: Ensure that data entered in forms (e.g., profile updates, report details) is properly validated before being written to the MongoDB database.</p> <p>Role-Based Access Control: Verify that only authorized users (doctors, patients) can view or modify specific data, based on their roles.</p> <p>Image Upload Testing: Test that images are successfully uploaded to Firebase storage, and URLs are correctly generated and stored in the database.</p>
Oracles	<p>Automated testing is done using the unittest framework in Python. The strategy involves:</p> <p>Using mocks to simulate user interactions with the database and external services like Firebase.</p> <p>Comparing actual outcomes against expected results using assertions.</p> <p>Ensuring that login, report generation, and profile updates behave as expected based on both valid and invalid input.</p> <p>Manually verifying critical GUI elements for appropriate feedback during interactions (e.g., login success/failure, form validation messages).</p>

Required Tools	<p>Unittest Framework: For unit testing Flask functions.</p> <p>Mock Library: To mock MongoDB operations and Firebase interactions.</p> <p>Flask Test Client: To simulate requests and responses in testing scenarios.</p> <p>Postman (Optional): For manual API testing and validation of key endpoints.</p> <p>GitHub: For code backup and version control.</p> <p>Firebase Console: For monitoring image uploads and validation of URL generation.</p> <p>PyTest (Optional): For higher-level testing across all system functions</p>
Success Criteria	<p>Testing will be considered successful if:</p> <p>All key use-case scenarios (user authentication, report management, disease prediction) are tested without errors.</p> <p>All business rules (e.g., role-based access, validation, report status updates) are applied correctly.</p> <p>All major features of the system (e.g., image uploads, profile updates) work as expected under normal and edge-case scenarios.</p> <p>The system successfully processes and retrieves correct data with appropriate feedback for invalid actions.</p>
Special Considerations	<p>Security: Focus should be placed on testing secure password handling and JWT token generation for user authentication.</p> <p>Performance: MongoDB performance should be monitored during load testing, especially when retrieving large reports or handling concurrent users.</p> <p>Image Quality: Ensure that the system correctly handles image uploads, avoiding loss of quality when uploaded to Firebase.</p> <p>Role Access: Make sure only authorized users (doctors) can access and modify sensitive reports, while patients can only view their own results.</p>

3.1.3 User Interface Testing

The goal of UI testing is to ensure that the user interface provides appropriate access and smooth navigation through the various features of the target software. The aim is to ensure that the objects within the UI function as intended and comply with both corporate and industry standards.

Technique Objective	<p>Test navigation throughout the system, ensuring the flow from screen to screen, field to field, and different access methods (tab keys, mouse movements, shortcuts) functions as required.</p> <p>Check if window objects and their characteristics (menus, size, position, state, and focus) behave as expected.</p> <p>Validate the rendering and interaction with elements such as forms, buttons, menus, navigation bars, and information cards across different application windows.</p>
Technique	<p>Implement integration tests using Jest and React Testing Library for components such as navbar_button, navbar, patient_card, and stat_card.</p> <p>Perform End to end tests for the following components to ensure smooth interaction between them: home, login_signup, login, patient, report, signup, doctor, diagnose, and doctorReview.</p> <p>Conduct end-to-end testing to ensure the entire frontend operates as expected when combined with the backend.</p> <p>Structured test files are stored as follows:</p> <p>Integration testing files are located under components/_tests/*.</p> <p>End to end testing files are stored under the cypress folder.</p> <p>Integration tests can be executed via the terminal using the command npm test.</p> <p>End to end tests can be executed via the terminal using the command npx cypress open.</p>
Oracles	<p>Utilize automated test scripts to observe whether navigation and form submissions behave as expected without routing or UI errors.</p> <p>Unit testing ensures individual components render and operate</p>

	<p>correctly, while integration testing verifies that the full functionality of the app is cohesive.</p> <p>For automated verification, ensure form submissions are captured, and UI errors such as misspelled labels, incorrect routes, and broken links are highlighted.</p>
Required Tools	<p>All application windows and components pass unit, integration, and end-to-end tests.</p> <p>There are no routing issues, UI bugs, or user-facing errors.</p> <p>The interface should meet user-friendly standards, with easy-to-understand navigation and an intuitive layout.</p>
Success Criteria	<p>All application windows and components pass unit, integration, and end-to-end tests.</p> <p>There are no routing issues, UI bugs, or user-facing errors.</p> <p>The interface should meet user-friendly standards, with easy-to-understand navigation and an intuitive layout.</p>
Special Considerations	<p>Not all custom third-party objects may be fully accessible for automated testing</p>

3.1.4 Performance Profiling

Performance profiling is a performance test focused on measuring response times, transaction rates, and other time-sensitive metrics. The aim is to verify that the system meets performance requirements under various conditions, such as different workloads or configurations.

For our MongoDB performance testing, we profiled key transactions and analyzed their performance based on anticipated and worst-case scenarios.

Technique Objective	<p>The objective of performance profiling is to evaluate how well the MongoDB-based system performs under different load conditions. This includes measuring response times and throughput under both normal workloads (expected traffic) and worst-case workloads (heavy traffic).</p>
---------------------	---

Technique	<p>We used JMeter to simulate real-world workloads for MongoDB queries, focusing on specific functional transactions such as:</p> <ul style="list-style-type: none"> Finding a unique report Retrieving all reports Fetching the newest report Finding all users <p>Scripts were designed to simulate both single-user interactions and multiple concurrent users to evaluate the system under different conditions.</p> <p>The test included increasing the number of queries and iterating the operations to stress the system, gathering data from different load scenarios.</p>
Oracles	<p>We assessed success by analyzing the following key performance indicators (KPIs):</p> <p>Response Times: How fast MongoDB returned results for each query under different loads.</p> <p>Transaction Rates: The number of successful operations completed per second.</p> <p>Throughput: The total volume of data processed by MongoDB during the test.</p> <p>Error Rate: Ensuring there were no database connection failures or timeouts under high load conditions.</p>
Required Tools	<p>For performance profiling, the following tools were used:</p> <p>Apache JMeter (5.6.3): For simulating load and generating performance metrics.</p>

	<p>JMeter Listeners: Including Summary Report, Aggregate Report, Graph Results, and Response Time Graph to monitor system behavior.</p> <p>Java Driver for MongoDB: To interact with MongoDB during the tests.</p> <p>Custom Groovy Scripts: For running MongoDB queries within JMeter.</p>
Success Criteria	<p>Single Transaction/User: Each transaction (e.g., finding a report) must complete without errors, and response times must be within acceptable limits.</p> <p>Multiple Transactions/Users: MongoDB should handle multiple concurrent transactions without failures, while maintaining stable response times and throughput.</p>
Special Considerations	<p>Testing was performed under both normal and heavy load conditions. Multiple virtual users were simulated in JMeter to represent hundreds of concurrent clients querying the MongoDB database.</p> <p>Tests were run in controlled environments to avoid interference from other processes, ensuring accurate measurements.</p> <p>The database used for testing was either scaled to the expected production size or adjusted proportionally to test specific scenarios</p>

3.1.5 Load Testing

Load testing was performed on the MongoDB database used for the skin disease web application to evaluate how the system behaves under different workload conditions, including concurrent requests and high loads. The goal of this testing was to measure the system's ability to handle typical and peak loads while maintaining optimal performance. MongoDB's performance was tested using Apache JMeter, focusing on specific database operations to identify performance bottlenecks and ensure stability under stress.

Technique Objective	The objective of load testing was to simulate typical and peak workload conditions by executing various MongoDB queries that reflect actual usage scenarios within the web application. The system's response times, throughput, and error rates were measured to ensure the database could handle the expected loads without degradation in performance.
Technique	<p>Transaction Simulation: JMeter was configured to simulate multiple transactions that represent key operations within the MongoDB database. This includes:</p> <p>Find a unique report: Simulates a user retrieving a single report by ID.</p> <p>Find all reports: Represents bulk retrieval of reports to test the performance of large dataset queries.</p> <p>Find the newest report: Tests the system's ability to retrieve the most recent report efficiently.</p> <p>Find all users: Simulates retrieving all user profiles from the database.</p> <p>Varying Workloads: Different workload levels were simulated using a combination of:</p> <p>Number of Threads: Varying the number of concurrent threads (e.g., 50 threads for testing).</p> <p>Ramp-up Period: Increasing the ramp-up period (e.g., 25 seconds) to simulate gradual load increases.</p> <p>Loop Count: Repeating the tests (e.g., 2 loops) to simulate sustained usage over time.</p> <p>Workload Representation:</p> <p>Peak loads were simulated by rapidly increasing the number of concurrent threads.</p> <p>Average workloads were simulated with moderate thread counts to represent typical daily use.</p>

	<p>Instantaneous peaks were simulated with a high number of threads for a short burst period.</p> <p>Sustained peaks were tested by keeping the high load running for longer durations.</p> <p>Environment Configuration: All tests were executed using JMeter's Apache JMeter (5.6.3) tool, with MongoDB running in a cloud environment via MongoDB Atlas.</p>
Oracles	<p>The following strategies were used to verify the success or failure of each test:</p> <p>Response Time: The system was expected to respond to each query within acceptable time limits for typical and peak loads. Acceptable response times were defined based on application requirements, with thresholds for queries such as retrieving a single report or all user profiles.</p> <p>Throughput: The number of successful transactions per second (TPS) was monitored, with higher values indicating better performance under load.</p> <p>Error Rate: Any test that produced significant errors (e.g., failed MongoDB queries or timeouts) was flagged for review. The goal was to maintain low or no errors during load testing.</p> <p>Resource Utilization: Resource monitoring tools were used to observe CPU and memory usage during tests. Increased load should not cause resource exhaustion that would lead to system crashes.</p>
Required Tools	<p>The following tools were employed during load testing:</p> <p>Apache JMeter (5.6.3): To simulate database loads, perform concurrency tests, and measure performance metrics such as response time, throughput, and error rates.</p> <p>JMeter Listeners: These tools were used to capture test data and generate reports:</p> <ul style="list-style-type: none"> View Result Tree Summary Report Aggregate Report Graph Results

	<p>Response Time Graph</p> <p>MongoDB Utilities: MongoDB's built-in utilities and management tools were used to monitor the database directly, ensuring that queries were executed as expected and data integrity was maintained.</p> <p>Custom Groovy Scripts: Custom scripts were written to execute MongoDB queries, including retrieving reports and user data, ensuring real-world scenarios were tested.</p>
Success Criteria	<p>The load testing was deemed successful if:</p> <p>The system maintained acceptable response times and throughput under varying load conditions.</p> <p>The error rate was low, and the system did not fail during peak load periods.</p> <p>The database queries executed efficiently without significant delays or timeouts.</p> <p>Resource utilization remained within acceptable limits, preventing system crashes or degradation in performance.</p>
Special Considerations	<p>Dedicated Testing Environment: Load testing was performed in a dedicated cloud environment, ensuring that other processes did not interfere with performance metrics.</p> <p>Database Scaling: The MongoDB database was populated with a representative number of records for both reports and users to simulate real-world conditions.</p> <p>Thread Configuration: Multiple thread configurations were tested to simulate various user loads, with threads ranging from small (10 threads) to large (50 threads) loads.</p>

3.1.6 Security and Access Control Testing

Security and Access Control Testing focuses on ensuring that users have appropriate access levels within the MongoDB database and the application itself. This includes testing at both the application level (access to data and functions) and the system level (system access and permissions).

Technique Objective	<p>Application-level Security: Ensure that users can only access the data and functions they are permitted to based on their assigned roles within the system.</p> <p>System-level Security: Verify that only authorized users can access the system, and they can only interact with the database and services as their access levels allow.</p>
Technique:	<p>Application-level Security:</p> <p>Identify user roles: Different user types, such as patients, doctors, and admins, were defined within the system, each with specific permissions.</p> <p>Create test cases for each user type: Patients can create and view reports but cannot modify or delete them.</p> <p>Doctors can view and modify patient reports but cannot delete them.</p> <p>Admins have full access, including the ability to delete reports or users.</p> <p>Test permissions: We performed testing by running specific transactions (e.g., querying the reports, modifying user details) for each user role and verifying access.</p> <p>Modify roles: For additional verification, we temporarily elevated or downgraded user permissions and re-ran the tests to ensure proper access restrictions or permissions were applied.</p> <p>System-level Security:</p>

	<p>Access verification: We tested the MongoDB system to ensure only authorized users could access the database using valid credentials and through the appropriate gateways (i.e., via secure connections like MongoDB Atlas).</p> <p>Connection tests: We verified that unauthorized users (i.e., invalid IPs or credentials) were denied access to MongoDB.</p> <p>Remote access testing: Remote access to the system was tested through VPN and SSH protocols, ensuring secure and authorized access.</p>
Oracles	<p>We determined the success of security tests by:</p> <p>Ensuring correct access control: Users only had access to data or functions based on their role.</p> <p>Testing the security breach points: No unauthorized access to the database, and correct user restrictions were enforced. Observing the system behavior for anomalies or unauthorized actions during security tests.</p>
Required Tools	<p>For security and access control testing, the following tools were used:</p> <p>Apache JMeter (5.6.3): Simulated different users accessing the MongoDB system to verify permissions.</p> <p>MongoDB Access Controls: MongoDB role-based access control (RBAC) to define specific permissions for each user.</p> <p>Custom Groovy Scripts: Scripts to verify access and data visibility for different user types.</p> <p>Security Probing Tools: While probing tools like “hacker” tools were not used extensively in this case, system-level access was tested for basic vulnerabilities.</p>
Success Criteria	<p>Application-level security: Each user type was able to access only the designated data or functions. Unauthorized access was denied, and role changes reflected accurate permissions.</p> <p>System-level security: Only authorized users were able to access the system and database. Remote access was secured, and unauthorized attempts were blocked.</p>
Special Considerations	<p>Access Control: We worked closely with the system administrator to ensure that MongoDB’s built-in security controls</p>

	<p>were configured properly.</p> <p>Environment: Load tests and security checks were performed on a dedicated testing environment to avoid interference with live systems.</p>
--	---

3.1.7 Failover and Recovery Testing

Failover and recovery testing ensures that the HealthBot+ web application can successfully failover and recover from hardware, software, or network malfunctions without losing data or compromising data integrity.

For systems requiring continuous operation, failover testing verifies that when a failover condition occurs, the system's recovery processes restore the application to a functional state.

Recovery testing is an adversarial test process in which the system is subjected to extreme or simulated conditions to induce failures, such as database failures, power outages, or network interruptions. During these tests, recovery processes are triggered, and the system is monitored to verify that it correctly recovers both functionality and data.

Technique Objective:	<p>Simulate failure conditions and invoke recovery processes (both manual and automated) to restore the Flask backend, React frontend, and MongoDB database to a stable and known state. Failures may arise from various sources. The following conditions will be tested to observe the system's behavior post-recovery:</p> <p>Power interruption to the client (React frontend) or server (Flask backend).</p> <p>Network communication failures that disrupt API calls between the frontend and backend or between the backend and MongoDB database.</p> <p>Corrupted or invalid data within the MongoDB database due to a system crash or hardware failure.</p> <p>High system workload leading to performance bottlenecks or system crashes.</p> <p>Interrupted processes, such as incomplete HTTP requests or image</p>
----------------------	--

	uploads, caused by power outages or network disruptions.
Technique	<p>Test scenarios already created for functional and business logic testing will serve as a foundation for generating a series of transactions to support failover and recovery testing. Key actions include:</p> <p>Simulating power loss on client-side machines (React frontend) or server-side (Flask backend) to observe system response.</p> <p>Simulating or initiating network interruptions to observe how communication between components is impacted and recovered.</p> <p>Rolling back incomplete transactions if an interruption occurs mid-process, such as during a medical report generation or image upload.</p> <p>Injecting corrupted data into MongoDB to observe how the system reacts and if it can recover from data integrity issues.</p> <p>Simulating a high workload (e.g., during report generation or API request spikes) and verifying the system's resilience under load.</p> <p>Testing data restoration by regularly backing up the MongoDB database and reloading it after failures.</p> <p>Additional recovery tests should be executed once the above conditions are triggered, with manual and automated recovery procedures invoked to return the system to a stable state.</p>
Tools Required:	<p>MongoDB Atlas backup and recovery tools</p> <p>Firebase storage backup for image data</p> <p>Monitoring tools such as Flask's error logs and network performance monitors</p> <p>Uninterruptible Power Supply (UPS) for server-side power backup</p> <p>External storage for database backup and recovery</p>
Success Criteria:	<p>Successful recovery of data after a power or network failure.</p> <p>Successful rollback of ongoing transactions when interrupted mid-process (e.g., medical report creation).</p> <p>Proper backup and recovery of MongoDB database and Firebase</p>

	<p>storage, ensuring that data can be restored to a stable state after a failure.</p> <p>Minimal downtime, with the system returning to normal operations promptly after failure.</p>
Special Considerations:	<p>Recovery testing may require simulating power loss or network failure, which can be intrusive and may need to be conducted in isolated environments.</p> <p>Alternative simulation methods using diagnostic tools may be necessary in environments where physical disconnection is not feasible.</p> <p>Backup and recovery processes should be automated where possible, and the backup frequency must be carefully managed to minimize data loss in case of a system failure.</p> <p>These tests should ideally be conducted during off-peak hours or on a dedicated test environment to minimize impact on live systems.</p>

3.1.8 Configuration Testing

Configuration testing ensures that the HealthBot+ web application operates optimally on various hardware and software configurations. Since this project includes a Flask backend, React frontend, MongoDB database, and Firebase storage, it is essential to verify that the application functions smoothly across different client environments, ensuring consistent performance and data integrity.

In most production environments, client devices and network conditions vary, with different hardware specifications, software configurations, and network speeds. This testing ensures that the application can adapt to various configurations without affecting performance or user experience.

Technique Objective	<p>Exercise the HealthBot+ system on a variety of hardware (client machines, mobile devices, etc.) and software configurations (different browser versions, operating systems, network types) to monitor and log its performance. The goal is to identify changes in behavior based on configuration states and ensure optimal performance under varying conditions.</p>
---------------------	--

	<p>For this project, it's important to configure the system for minimal memory usage and data consumption, focusing on high performance during diagnosis processes and medical report generation, without requiring high-end hardware or graphics processing.</p>
Technique	<p>Use Function Test scripts to verify application functionality, database connections, and queries to MongoDB.</p> <p>Test the software's performance while running non-target-of-test applications (e.g., other browser tabs, productivity tools like Google Docs) to simulate typical user environments.</p> <p>Execute key transactions (such as image uploads, report generation, and chatbot interactions) while varying the system's available memory and CPU resources.</p> <p>Test on various configurations, including different operating systems (Windows, macOS, Linux) and web browsers (Chrome, Firefox, Safari).</p> <p>Simulate different network speeds (Wi-Fi, 4G, 5G) and latencies to observe how network interruptions or slowdowns impact the system.</p>
Required Tools	<p>Flask's built-in debugger and performance monitoring tools</p> <p>Browser developer consoles (Chrome DevTools, Firefox Developer Tools) for monitoring memory usage, network activity, and performance</p> <p>MongoDB Atlas tools to track database performance under different client configurations</p> <p>Firebase monitoring tools for analyzing image upload and storage behavior across varying network speeds</p>
Success Criteria	<p>The system operates smoothly under different software and hardware configurations, meeting user requirements for both performance and responsiveness.</p> <p>The application is user-friendly and provides a seamless experience with reasonable data usage and memory consumption.</p> <p>Minimal configuration changes by users ensure system stability and performance in diverse client environments.</p>

3.1.9 ML/DS Testing

- Melanoma Detection Model

DS/ML (Data Science and Machine Learning) testing for skin disease detection models focuses on evaluating the performance, accuracy, and reliability of models designed to classify skin lesions as benign or malignant. The objective of this testing is to ensure the model generalizes well to unseen data, produces reliable predictions, and operates optimally under various testing conditions. Key aspects of this testing include measuring performance using metrics such as accuracy, precision, recall, and F1-score, identifying the optimal classification threshold through ROC curve analysis, and ensuring interpretability through explainable AI techniques like Grad-CAM. Thorough testing and validation are critical to ensure the model meets performance standards, especially in medical applications where accurate and trustworthy predictions are vital.

Technique Objective	The objective of testing the melanoma detection model is to evaluate the model's performance on unseen data, validate its ability to generalize, and assess its precision, recall, accuracy, and F1-score. Additionally, the goal is to detect the optimal threshold for binary classification and evaluate the model's overall ability to differentiate between benign and malignant cases.
Technique	<p>Train-Validation Split: The dataset was split into training and validation sets using an 80/20 ratio, ensuring the same data distribution in both sets using stratified sampling.</p> <p>Training Monitoring: The model was monitored for accuracy and loss during training using EarlyStopping to halt training once validation performance plateaued.</p> <p>Performance Metrics: Accuracy, precision, recall, F1-score, and AUC were used to assess the model's effectiveness.</p> <p>ROC Curve: Receiver Operating Characteristic (ROC) curve was used to evaluate model sensitivity (true positive rate) and specificity (false positive rate), helping to find the optimal threshold for binary classification.</p> <p>Confusion Matrix: A confusion matrix was generated to visualize correct and incorrect predictions for both classes (benign and malignant).</p> <p>Threshold Tuning: Based on the ROC curve, an optimal classification threshold was determined to minimize false positives and false negatives.</p>

	<p>Grad-CAM: Used Grad-CAM and Grad-CAM++ techniques to visualize the regions of the input images that the model focused on for making predictions, ensuring interpretability.</p>
Oracles	<p>Ground Truth Labels: The ground truth labels from the SIIM-ISIC melanoma dataset were used to evaluate the correctness of the model's predictions.</p> <p>Performance Metrics: Accuracy, precision, recall, F1 score, and AUC were used as quantitative oracles to assess the model's success in classifying images.</p> <p>ROC Curve Analysis: The ROC curve and AUC value served as an indicator of how well the model could differentiate between classes across different threshold settings.</p> <p>Confusion Matrix: The confusion matrix highlighted misclassifications and was used to fine-tune the classification threshold for optimal performance.</p>
Required Tools	<p>TensorFlow/Keras: Used for building, training, and evaluating the deep learning model.</p> <p>Scikit-learn: For calculating performance metrics (accuracy, precision, recall, F1 score) and generating confusion matrices and ROC curves.</p> <p>Matplotlib: For plotting training and validation accuracy, loss, confusion matrix, and ROC curve.</p> <p>Grad-CAM Implementation: TensorFlow was used to generate Grad-CAM and Grad-CAM++ visualizations to understand the model's focus during inference.</p>
Success Criteria	<p>The model should achieve high accuracy and AUC values on the validation set, demonstrating its ability to differentiate between benign and malignant lesions.</p> <p>Balanced precision and recall values are desired, with minimal overfitting, as shown by the consistency between training and validation accuracy.</p> <p>Grad-CAM outputs should show that the model is focusing on relevant areas of the images (e.g., lesion areas) for its predictions.</p> <p>The confusion matrix should reflect a minimal number of false positives and false negatives.</p>

Special Considerations	<p>Imbalanced Data: The dataset contains significantly fewer malignant cases compared to benign ones, which was addressed using RandomOverSampler to balance the training set.</p> <p>Missing Data: Missing values in important features such as sex, age, and anatomical site were imputed using the mode of the dataset.</p> <p>Early Stopping: EarlyStopping was employed to prevent overfitting, ensuring that training stopped once validation performance plateaued.</p> <p>Threshold Optimization: The optimal threshold was determined based on the ROC curve to minimize false positives and negatives in a medical context where misclassification can have serious consequences.</p>
------------------------	---

- **Skin Disease Detection Model**

Technique Objective:	The goal of this testing is to evaluate the performance of the skin disease detection model, ensuring its ability to accurately classify images into seven categories of skin diseases. The objective is to measure the effectiveness of the model using key performance metrics like accuracy, precision, recall, and F1-score. Additionally, the evaluation will include plotting training curves, confusion matrices, and testing the model's ability to predict single images.
Technique:	<p>Training and Validation Evaluation: After training the model, validation accuracy and loss are monitored using plots to evaluate the training effectiveness and identify potential overfitting or underfitting.</p> <p>Test Set Evaluation: The model is tested on a held-out test set where predictions are compared to true labels. Key metrics like test accuracy, precision, recall, F1-score, and confusion matrices are computed.</p> <p>Prediction Analysis: For further evaluation, individual image predictions are tested, and the correctness of the model's prediction is verified by comparing the predicted labels with the ground truth.</p>

Oracles:	<p>Accuracy and Loss Curves: These are plotted for both the training and validation datasets. A good match between training and validation performance indicates that the model generalizes well.</p> <p>Confusion Matrix: This is used to visualize how well the model predicts each class, helping to identify where the model may be confusing certain diseases.</p> <p>Classification Report: Precision, recall, and F1-score are used as oracles to evaluate how well the model handles imbalanced classes and how well it distinguishes between different types of skin diseases.</p>
Required Tools:	<p>TensorFlow/Keras: Used to build, train, and evaluate the model.</p> <p>Matplotlib, Plotly: For plotting accuracy/loss curves and confusion matrices.</p> <p>Scikit-learn (Sklearn): For generating the confusion matrix and classification reports.</p> <p>Seaborn: For enhanced visualization of confusion matrices.</p>
Success Criteria:	<p>Accuracy: The model should achieve a reasonable level of accuracy on the test set, ensuring it can generalize well to unseen data.</p> <p>Confusion Matrix: The matrix should indicate minimal misclassifications between the disease categories, particularly for critical diseases.</p> <p>Precision and Recall: Balanced precision and recall for each disease class (especially for rarer classes) to ensure the model is not biased toward more frequent classes.</p> <p>Training Curves: The training and validation curves should show steady improvement and convergence without significant divergence, indicating balanced model training.</p>
Special Considerations:	<p>Class Imbalance: Given the imbalance in the number of samples across disease categories, the evaluation must account for potential biases, which are addressed through data augmentation and oversampling during training.</p>

	<p>Individual Predictions: In addition to overall evaluation, single-image predictions are tested to check the correctness of predictions for random individual images.</p> <p>Stopping Criteria: The model is evaluated at each epoch, with early stopping and learning rate reduction applied to prevent overfitting and optimize performance.</p>
--	--

4. Deliverables

4.1 Deliverable: Functional Testing (Integration Testing for the Backend)

The functional testing for the backend focused on ensuring the correct implementation and interaction of the Flask-based API endpoints. These endpoints handle critical operations such as user authentication (login, signup, Google login), report management, and doctor information retrieval. The primary objective was to verify that each endpoint behaves as expected under various conditions, including success, failure, and error scenarios. Unit tests were written for each key functionality using Python's **unittest** framework and **unittest.mock** for mocking database operations.

test_auth.py

```
PS C:\Users\Isara Liyanage\Documents\Github\HealthBot_Plus> & "C:/Users/Isara Liyanage/AppData/Local/Programs/Python/Python312/python.exe" "C:/Users/Isara Liyanage/Documents/Github/HealthBot_Plus/HealthBot_Plus App/backend/test_auth.py"
{'doctor_id': '678d62740eea45f9c09918d9'}
{'doctor_id': '678d62740eea45f9c09918db'}
c:\Users\Isara Liyanage\Documents\Github\HealthBot_Plus\HealthBot_Plus App\backend\login.py:129: DeprecationWarning: datetime.datetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
    'exp': datetime.utcnow() + timedelta(hours=1)
c:\Users\Isara Liyanage\Documents\Github\HealthBot_Plus\HealthBot_Plus App\backend\login.py:145: DeprecationWarning: datetime.datetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
    expires=datetime.utcnow() + timedelta(hours=1),
...c:\Users\Isara Liyanage\Documents\Github\HealthBot_Plus\HealthBot_Plus App\backend\login.py:30: DeprecationWarning: datetime.datetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
    'exp': datetime.utcnow() + timedelta(hours=1)
c:\Users\Isara Liyanage\Documents\Github\HealthBot_Plus\HealthBot_Plus App\backend\login.py:46: DeprecationWarning: datetime.datetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
    expires=datetime.utcnow() + timedelta(hours=1),
..
Ran 7 tests in 16.868s

OK

-----
Total Tests: 7
Success: 7
Failures: 0
Errors: 0
Skipped: 0
```

test_report.py

```
PS C:\Users\Isara Liyanage\Documents\Github\HealthBot_Plus> & "C:/Users/Isara Liyanage/AppData/Local/Programs/Python/Python312/python.exe" "C:/Users/Isara Liyanage/Documents/Github/HealthBot_Plus/HealthBot_Plus App/backend/test_report.py"
..612e4f653d141f5aeedf92e7
....
Ran 6 tests in 13.590s

OK

-----
Total Tests: 6
Success: 6
Failures: 0
Errors: 0
Skipped: 0
```

test_signup.py

```
PS C:\Users\Isara Liyanage\Documents\GitHub\HealthBot_Plus> & "C:/Users/Isara Liyanage/AppData/Local/Programs/Python/Python312/python.exe" "C:/Users/Isara Liyanage/Documents/GitHub/HealthBot_Plus/App/backend/test_signup.py"
<MagicMock name='db._getitem_().insert_one()' id='2431194088800'>
Document inserted with ID: None
<MagicMock name='db._getitem_().insert_one()' id='2431194278384'>
Document inserted with ID: 67b0641c95851f95bf33ed76
..
-----
Ran 3 tests in 15.157s

OK

-----
Total Tests: 3
Success: 3
Failures: 0
Errors: 0
Skipped: 0
```

4.1.1 User Update Tests:

Introduction:

The user update tests verify the functionality that allows users to update their personal information. These tests ensure that the correct details are saved in the database, and the response confirms a successful update. By testing various update scenarios, we ensure that the system accurately modifies user profiles when requested.

4.1.2 Google Login Tests:

Introduction:

Google login functionality provides users with the option to authenticate using their Google account. These tests simulate the process of Google-based login, ensuring that valid users can successfully access the system and invalid attempts return proper error responses.

4.1.3 Doctor Retrieval Tests:

Introduction:

The doctor retrieval tests focus on ensuring that the system can correctly retrieve and return doctor details based on the provided doctor ID. By testing both valid and invalid IDs, the system is checked for proper retrieval functionality and appropriate error handling when no data is found.

4.1.4 Report Management Tests:

Introduction:

The report management tests cover functionalities related to retrieving, updating, and modifying medical reports. These tests ensure that users can successfully access reports, update statuses (e.g., mark as "Reviewed"), and add comments or update model accuracy. This is crucial for the accuracy and timeliness of the report-handling system.

4.1.5 Signup Tests:

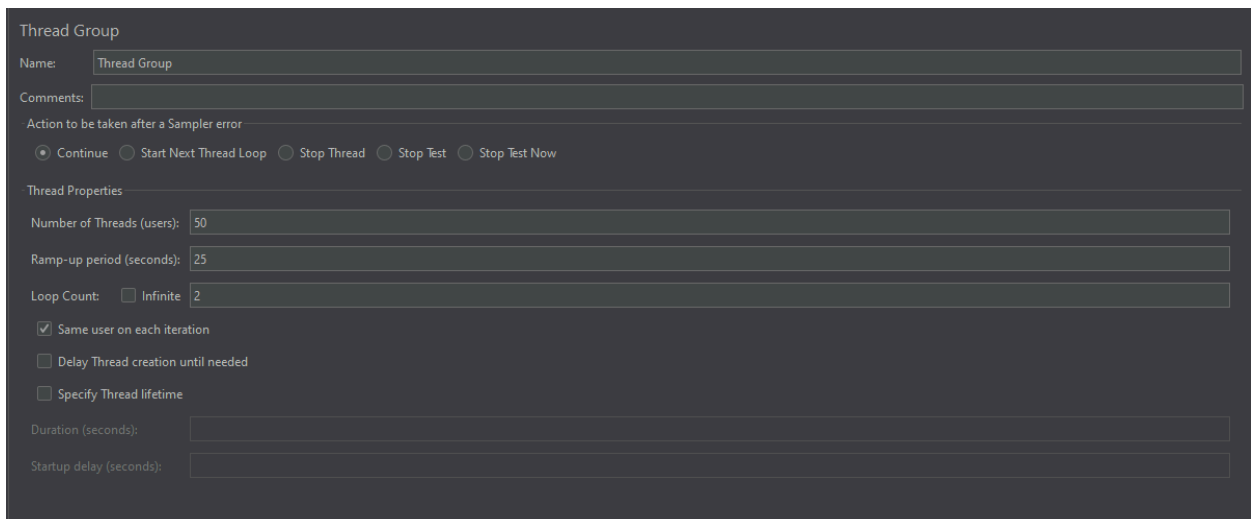
Introduction:

The signup tests validate the user registration process, ensuring that new users can be successfully added to the system while preventing duplicate registrations. These tests cover scenarios of successful user creation, handling cases where the email is already in use, and simulating database failures.

4.2 Database Testing Deliverables

This section summarizes the MongoDB performance testing, focusing on evaluating the database's efficiency in handling key operations under load. The tested operations included retrieving specific reports, fetching all reports, identifying the newest report, and listing all users. Apache JMeter (version 5.6.3) was used, configured with 50 threads, a 25-second ramp-up, and 2 loops.

The objective was to capture performance metrics such as response times and throughput across varying workloads. JMeter listeners, including Summary Report, Aggregate Report, and Graph Results, were used to analyze system behavior and identify optimization areas.



The screenshot shows the 'Thread Group' configuration window in Apache JMeter. The 'Name' field is set to 'Thread Group'. The 'Comments' field is empty. Under 'Action to be taken after a Sampler error', the 'Continue' radio button is selected. The 'Thread Properties' section shows 'Number of Threads (users)' set to 50, 'Ramp-up period (seconds)' set to 25, and 'Loop Count' set to 2 with the 'Infinite' checkbox unchecked. The 'Same user on each iteration' checkbox is checked, while 'Delay Thread creation until needed' and 'Specify Thread lifetime' are unchecked. The 'Duration (seconds)' and 'Startup delay (seconds)' fields are empty.

4.2.1 Test 1: Find a Unique Report

This test measured MongoDB's performance in retrieving a unique report by user ID. Metrics captured include response times and system behavior under load, providing insights into query efficiency.

4.2.2 Test 2: Find All Reports

This test evaluated MongoDB's ability to handle bulk report retrievals. Performance metrics showed how the database managed multiple simultaneous requests and large datasets.

4.2.3 Test 3: Find the Newest Report

The test focused on MongoDB's efficiency in retrieving the latest report. Results highlighted how well the database handled concurrent requests while maintaining low response times.

4.2.4 Test 4: Find All Users

This test analyzed MongoDB's performance in fetching all users. Metrics showed how MongoDB scaled to handle multiple requests and large datasets efficiently without significant performance drops.

1) Find a unique report

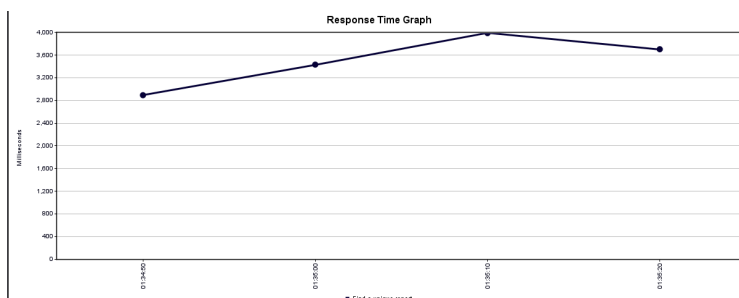
Summary report

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Find a unique re...	300	3555	2613	6260	683.71	0.00%	1.3/sec	0.00	0.00	.0
TOTAL	300	3555	2613	6260	683.71	0.00%	1.3/sec	0.00	0.00	.0

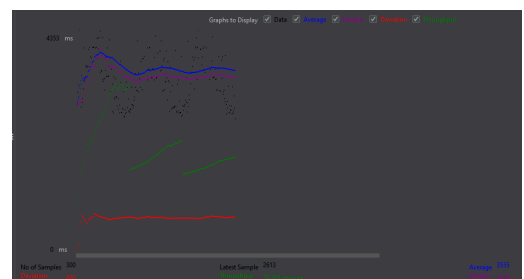
Aggregate report

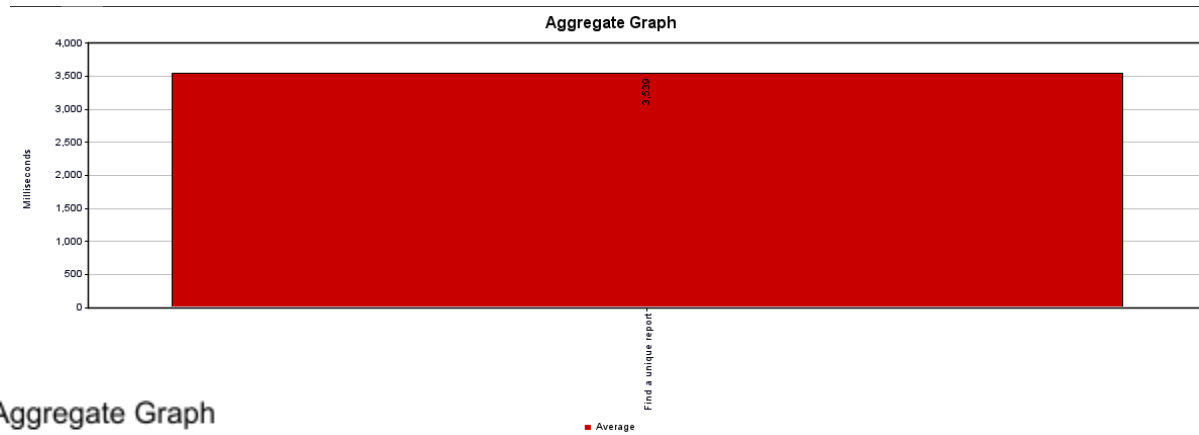
Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/s...	Sent KB/sec
Find a unique...	300	3555	3419	4545	4989	5540	2613	6260	0.00%	1.3/sec	0.00	0.00
TOTAL	300	3555	3419	4545	4989	5540	2613	6260	0.00%	1.3/sec	0.00	0.00

Response Time graph



Graph Results





Aggregate Graph

2) Find all reports

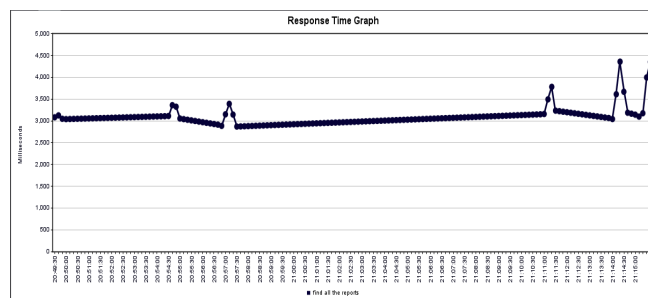
Summary report

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
find all the reports	1000	3532	2760	7144	620.23	0.00%	3.5/min	0.00	0.00	.0
TOTAL	1000	3532	2760	7144	620.23	0.00%	3.5/min	0.00	0.00	.0

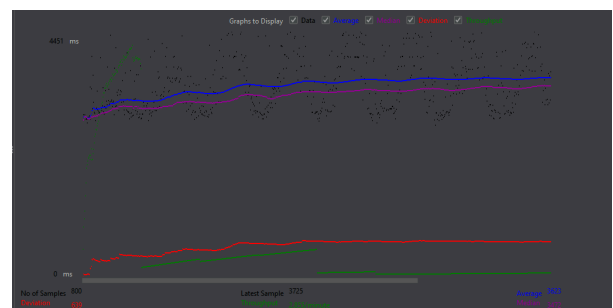
Aggregate report

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/s...	Sent KB/sec
find all the re...	1000	3532	3361	4422	4738	5341	2760	7144	0.00%	3.5/min	0.00	0.00
TOTAL	1000	3532	3361	4422	4738	5341	2760	7144	0.00%	3.5/min	0.00	0.00

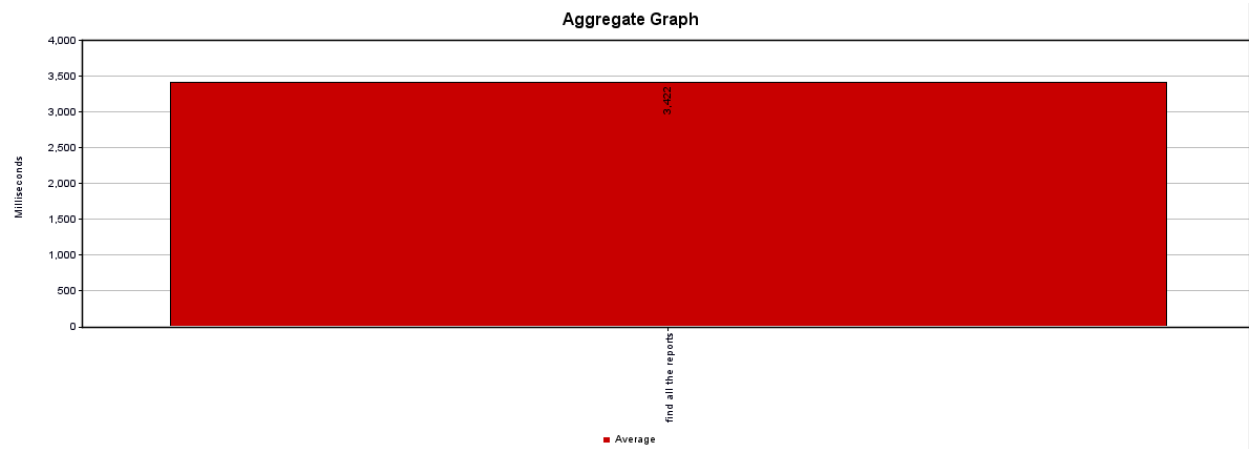
Response Time graph



Graph Results



Aggregate Graph



3)Find all users

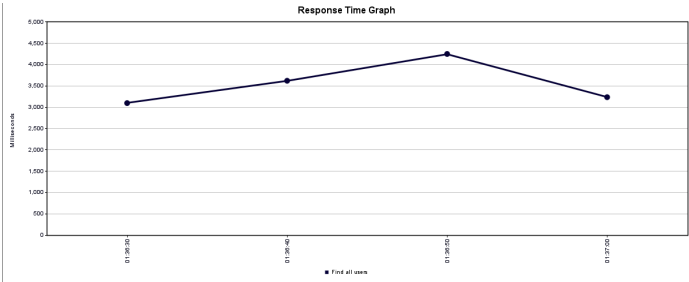
Summary report

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Find all users	500	3533	2649	6622	658.57	0.00%	1.9/min	0.00	0.00	.0
TOTAL	500	3533	2649	6622	658.57	0.00%	1.9/min	0.00	0.00	.0

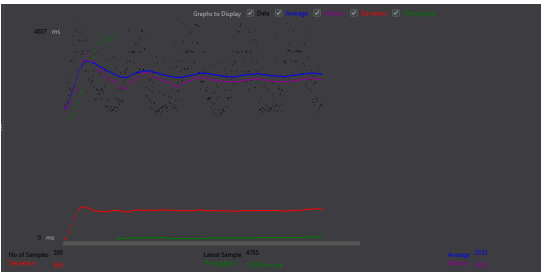
Aggregate report

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/s...	Sent KB/sec
Find all users	500	3533	3405	4441	4765	5265	2649	6622	0.00%	1.9/min	0.00	0.00
TOTAL	500	3533	3405	4441	4765	5265	2649	6622	0.00%	1.9/min	0.00	0.00

Response Time graph



Graph Results



4.3 Frontend Testing Deliverables

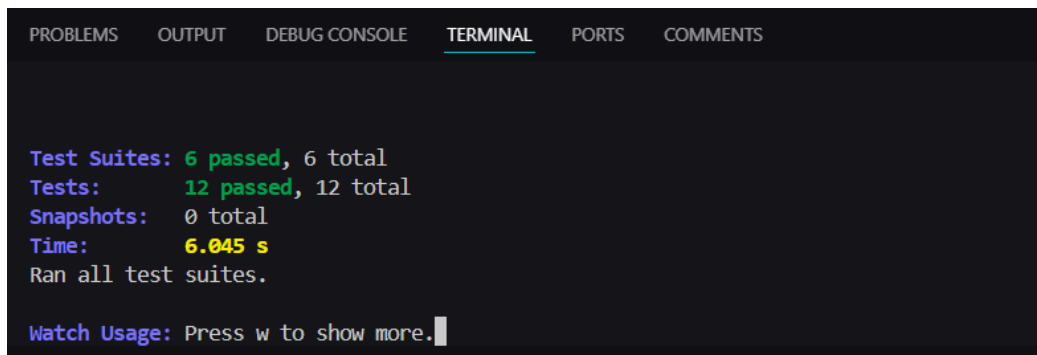
The front-end testing for this project was carried out in two primary stages: **Integration Testing** and **End-to-End Testing**. Unit testing was conducted manually by the developer during the development phase.

4.3.1 Integration Testing:

Integration testing was performed using **Jest** and **React Testing Library** to ensure that various components worked together as expected. This stage focused on verifying the interaction between different UI elements, form submissions, API calls, and ensuring the overall functionality of the application.

Components tested during this phase include:

- **AudioRecorder**
- **NavbarButton**
- **Navbar**
- **OAuth**
- **PatientCard**
- **StatCard**



The screenshot shows a terminal window with a dark background and light-colored text. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is selected and underlined), 'PORTS', and 'COMMENTS'. The terminal output displays the following information: 'Test Suites: 6 passed, 6 total', 'Tests: 12 passed, 12 total', 'Snapshots: 0 total', 'Time: 6.045 s', and 'Ran all test suites.' At the bottom, it says 'Watch Usage: Press w to show more.' with a cursor at the end of the line.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS

Test Suites: 6 passed, 6 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        6.045 s
Ran all test suites.

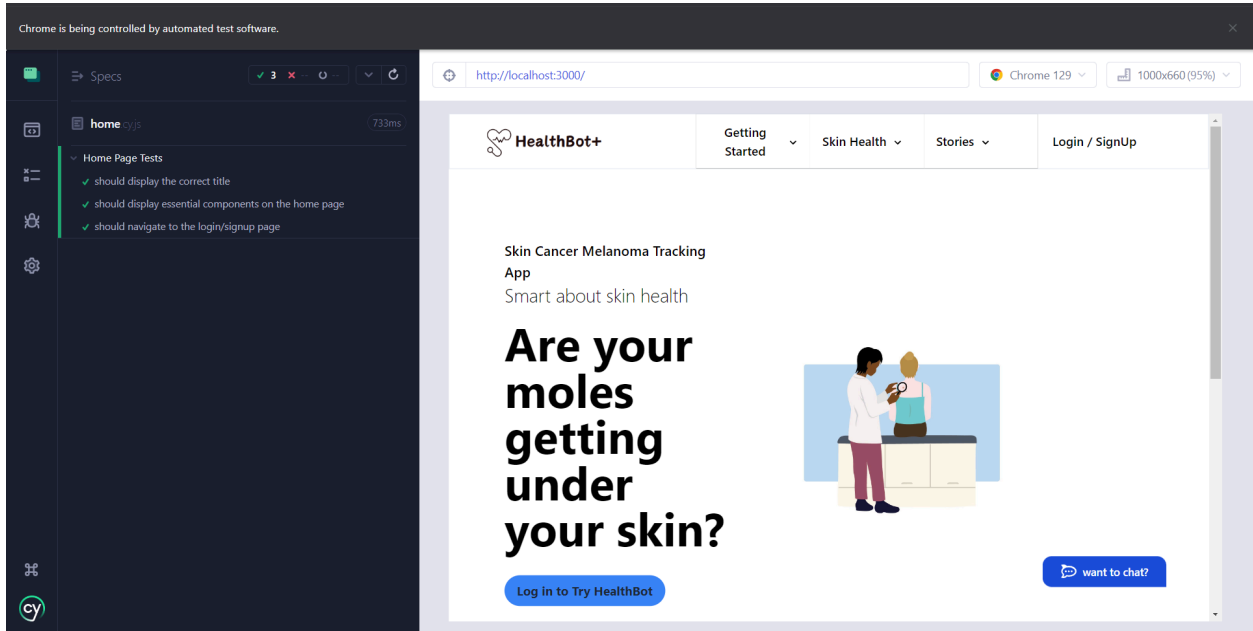
Watch Usage: Press w to show more.
```

4.3.2 End-to-End (E2E) Testing:

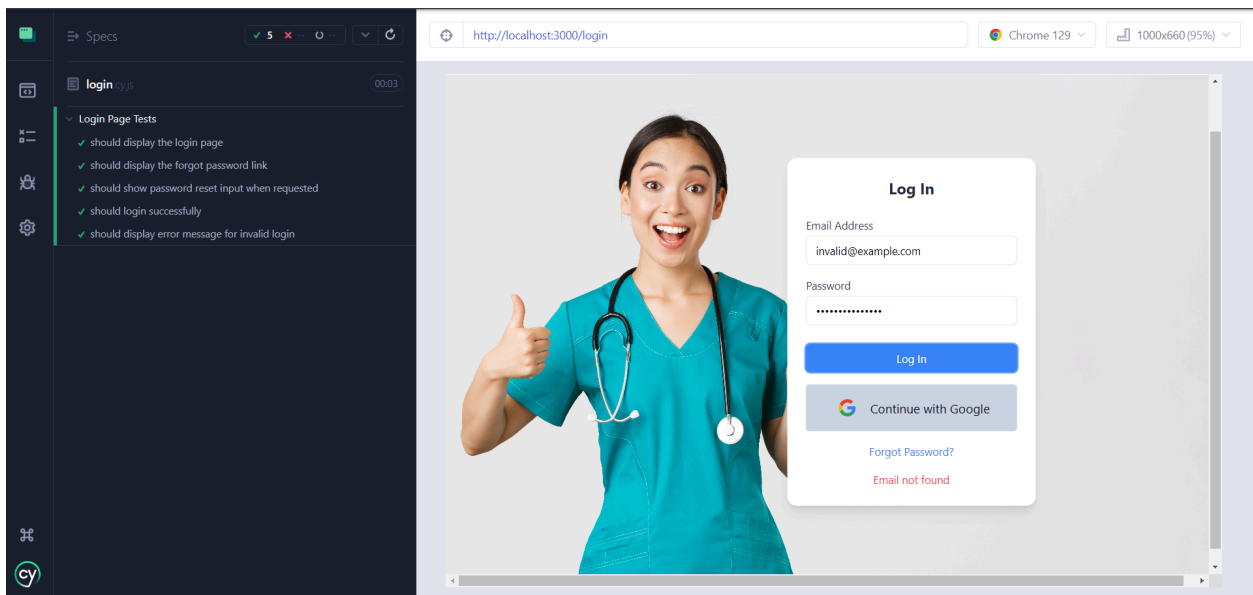
E2E testing was performed to simulate real-world user scenarios, ensuring that the entire application workflow—from user input to API responses—functioned as expected. This testing helped identify issues that could affect the user experience.

During this phase, we tested the following pages to validate the user flow:

○ Home



○ Login



○ Login/Signup

The screenshot shows the Cypress test runner on the left and a web browser on the right. The browser displays the 'HealthBot+' login/signup page at http://localhost:3000/login_signup. The page features a navigation bar with 'Getting Started', 'Skin Health', 'Stories', and 'Login / Signup'. The main content area has a large heading 'Empowering skin health with AI.' and a 'Get started' section with 'Log In' and 'Sign Up' buttons.

Cypress Specs:

- login_signup cypress
- 00:02
- Login/Signup Page Tests
 - ✓ should display the correct title
 - ✓ should render the empowering message
 - ✓ should render the AI message
 - ✓ should display 'Get started' heading
 - ✓ should navigate to the login page when 'Log In' button is clicked
 - ✓ should navigate to the signup page when 'Sign Up' button is clicked

○ SignUp

The screenshot shows the Cypress test runner on the left and a web browser on the right. The browser displays the 'HealthBot+' signup page at <http://localhost:3000/signup>. The page features a navigation bar with 'Getting Started', 'Skin Health', 'Stories', and 'Login / Signup'. The main content area has a background image of a smiling woman in medical scrubs holding a smartphone, and a 'Enter Your Details' form with fields for Name, Country, Birthday, Sex, and Description, followed by a 'Next' button.

Cypress Specs:

- signup cypress
- 00:06
- Signup Component Tests
 - ✓ renders the Signup component
 - ✓ validates required fields in Step 1
 - ✓ validates fields in Step 1 and navigates to Step 2
 - ✓ submits the form successfully

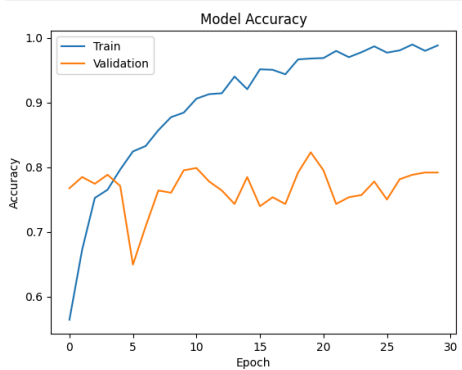
4.4 Deliverables for ML/DS Models

4.4.1 Deliverables for Melanoma Detection Model

In this section, we outline the key deliverables generated during the testing process of the Melanoma Detection Model. These deliverables provide valuable insights to stakeholders, demonstrating the model's performance, accuracy, and interpretability. The following artifacts serve as evidence of the model's evaluation and testing efforts, offering tangible benefits to various stakeholders.

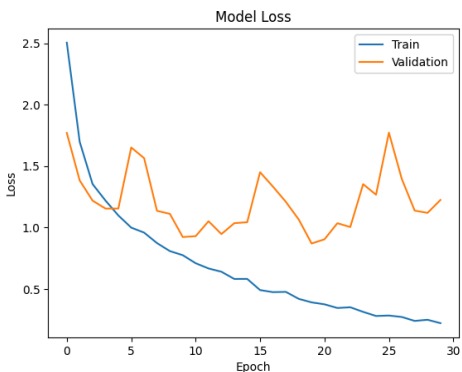
- **Train and Validation Accuracy Plot**

This plot represents the model's training and validation accuracy over multiple epochs. It is crucial for assessing the model's performance improvements during training and its generalization to unseen validation data. Stakeholders can use this to evaluate how well the model learns and prevents overfitting.



- **Train and Validation Loss Plot**

This plot shows the comparison of training and validation loss across epochs, providing insight into the learning process of the melanoma detection model. It helps stakeholders assess the model's optimization process and whether it is converging without overfitting, indicating how efficiently the model is learning.



- **Evaluation Metrics**

The evaluation metrics for the Melanoma Detection Model include Precision, Recall, F1 Score, and Accuracy, which collectively provide a comprehensive view of the model's performance. These metrics are critical for understanding how

	Metric	Score
0	Precision	0.764706
1	Recall	0.845528
2	F1 Score	0.803089
3	Accuracy	0.832237

well the model distinguishes between melanoma and non-melanoma cases, while minimizing both false positives and false negatives. All metrics are presented in a single table for stakeholders to quickly assess the model’s effectiveness in a clinical context.

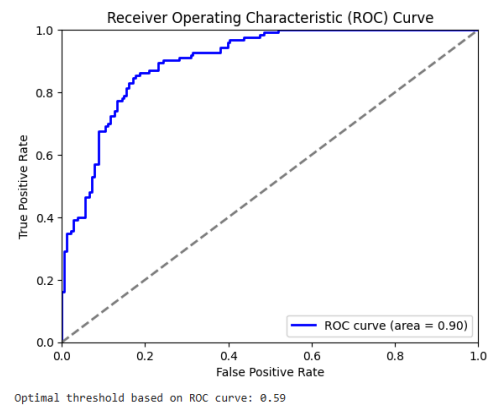
● **Classification Report**

The classification report for the melanoma detection model summarizes the model’s performance metrics for both melanoma and non-melanoma classe(malignant and benign), including precision, recall, F1 score, and support. This report helps stakeholders evaluate how well the model distinguishes between melanoma and benign cases and identify areas where the model could improve in detecting melanoma more accurately.

	precision	recall	f1-score	support
benign	0.89	0.82	0.85	181
malignant	0.76	0.85	0.80	123
accuracy			0.83	304
macro avg	0.83	0.83	0.83	304
weighted avg	0.84	0.83	0.83	304

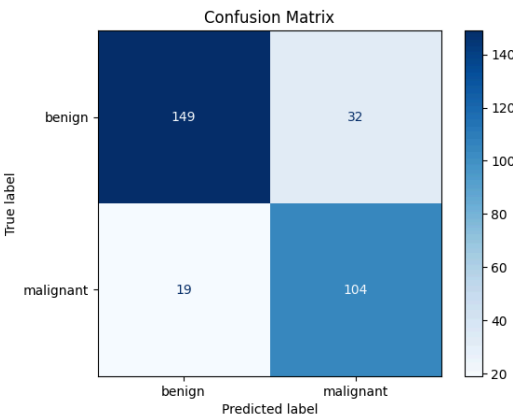
● **ROC Curve and AUC Score**

The ROC curve provides a graphical representation of the model’s sensitivity versus specificity for melanoma detection. The Area Under the Curve (AUC) score quantifies the model’s ability to discriminate between malignant and benign cases. A higher AUC score reflects better model performance, which is critical for ensuring the model’s reliability in clinical settings.



● **Confusion Matrix**

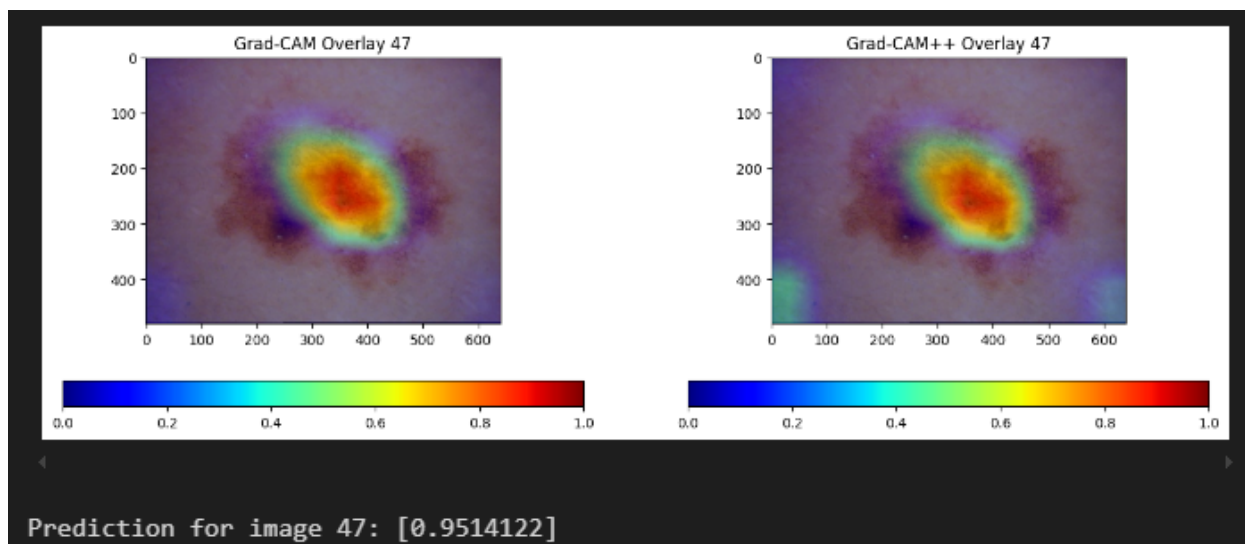
The confusion matrix for the melanoma detection model provides a detailed breakdown of the true positives, true negatives, false positives, and false negatives. This information is crucial for understanding where the model makes errors, helping stakeholders evaluate the pattern of



misclassifications and the types of cases that are most challenging for the model.

- **Grad-CAM and Grad-CAM++ Output**

The Grad-CAM and Grad-CAM++ visualizations highlight the regions of the skin images that were most influential in the melanoma detection model's predictions. These outputs offer explainability by showing which areas of the image the model focused on when identifying melanoma, providing stakeholders with valuable interpretability and transparency for the model's decision-making process, which is essential in healthcare applications.



4.4.2 Deliverables for Skin Disease Detection Model

The following deliverables are essential for assessing the model's performance in detecting multiple skin diseases, providing valuable insights to stakeholders:

- **Model Training and Validation Metrics**

A combined plot that showcases both the model's accuracy and loss over the course of training and validation. This graph offers insights into how well the model learns from the data and provides an indication of overfitting or underfitting.

```
458/458 [=====] - 2s 4ms/step - loss: 0.1891 - accuracy: 0.9638 - val_loss: 0.1579 - val_accuracy: 0.9578
Epoch 16/50
458/458 [=====] - 2s 4ms/step - loss: 0.8953 - accuracy: 0.9677 - val_loss: 0.1445 - val_accuracy: 0.9536
Epoch 00015: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
458/458 [=====] - 2s 4ms/step - loss: 0.8224 - accuracy: 0.9938 - val_loss: 0.0907 - val_accuracy: 0.9771
Epoch 17/50
458/458 [=====] - 2s 4ms/step - loss: 0.8094 - accuracy: 0.9984 - val_loss: 0.0951 - val_accuracy: 0.9771
Epoch 18/50
458/458 [=====] - 2s 4ms/step - loss: 0.8060 - accuracy: 0.9994 - val_loss: 0.0936 - val_accuracy: 0.9780
Epoch 19/50
458/458 [=====] - 2s 4ms/step - loss: 0.8041 - accuracy: 0.9997 - val_loss: 0.0924 - val_accuracy: 0.9792
Epoch 00019: ReduceLROnPlateau reducing learning rate to 1.00000000474974514e-05.
Epoch 20/50
458/458 [=====] - 2s 4ms/step - loss: 0.8029 - accuracy: 0.9999 - val_loss: 0.0909 - val_accuracy: 0.9780
Epoch 21/50
458/458 [=====] - 2s 4ms/step - loss: 0.8027 - accuracy: 0.9999 - val_loss: 0.0902 - val_accuracy: 0.9788
Epoch 22/50
458/458 [=====] - 2s 4ms/step - loss: 0.8026 - accuracy: 0.9999 - val_loss: 0.0902 - val_accuracy: 0.9783
Epoch 00022: ReduceLROnPlateau reducing learning rate to 1.0000000056873453e-06.
458/458 [=====] - 2s 4ms/step - loss: 0.8025 - accuracy: 0.9999 - val_loss: 0.0902 - val_accuracy: 0.9783
Epoch 24/50
458/458 [=====] - 2s 4ms/step - loss: 0.8025 - accuracy: 0.9999 - val_loss: 0.0902 - val_accuracy: 0.9783
Epoch 25/50
458/458 [=====] - 2s 4ms/step - loss: 0.8024 - accuracy: 0.9999 - val_loss: 0.0900 - val_accuracy: 0.9783
Epoch 00025: ReduceLROnPlateau reducing learning rate to 1.000000111628885e-07.
Epoch 26/50
458/458 [=====] - 2s 4ms/step - loss: 0.8024 - accuracy: 0.9999 - val_loss: 0.0900 - val_accuracy: 0.9783
Epoch 00026: early stopping
```

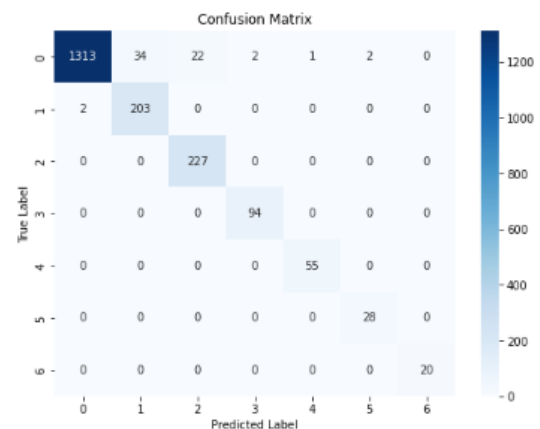

- **Evaluation Metrics**

A comprehensive table that includes key metrics such as Accuracy, Precision, Recall, F1 Score, and Support for each of the seven skin diseases. These metrics provide stakeholders with a detailed view of the model's performance across different disease classifications.

Test Accuracy: 96.855%				
	precision	recall	f1-score	support
nv	1.00	0.96	0.98	1374
mel	0.86	0.99	0.92	205
bk1	0.91	1.00	0.95	227
bcc	0.98	1.00	0.99	94
akiec	0.98	1.00	0.99	55
vasc	0.93	1.00	0.97	28
df	1.00	1.00	1.00	20
accuracy			0.97	2003
macro avg	0.95	0.99	0.97	2003
weighted avg	0.97	0.97	0.97	2003

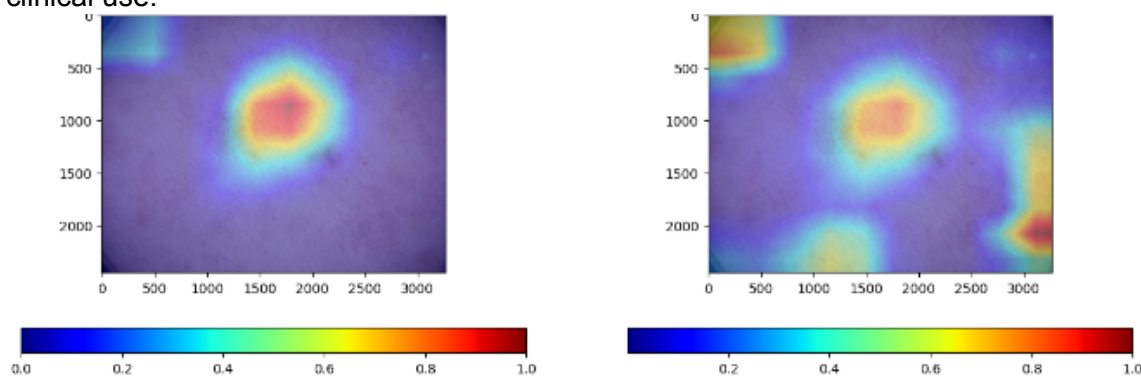
- **Confusion Matrix**

The confusion matrix presents a clear view of the model's classification performance, showing true positives, true negatives, false positives, and false negatives for each disease class. This is essential for understanding the specific types of classification errors and guiding further model improvements.



- **Grad-CAM and Grad-CAM++ Visualizations**

Grad-CAM and Grad-CAM++ provide visual explanations of the model's predictions by highlighting the image regions that influence its decisions. These insights enhance model transparency, making the predictions more interpretable and trustworthy for clinical use.



5. Risks, Dependencies, Assumptions, and Constraints

Risk	Mitigation Strategy	Contingency (if Risk is realized)
Prerequisite entry criteria are not met.	Developers will clearly define the prerequisites for starting testing, including data setup and model training. Users will aim to meet the prerequisites.	Review and address any missing prerequisites. Consider revising the test schedule.
Test data proves inadequate.	Developers will specify the requirements for test data, ensuring users provide sufficient, protected data for accurate testing.	Redefine the test data or collect additional data. Review and update the test plan and components (scripts). Consider retesting or load test failure.
Database requires refresh.	Admin or Developer will ensure that the MongoDB database is regularly refreshed according to the needs of the testing phase.	Restore database data and restart testing. Clear database if needed.
Data is corrupted.	System Admin will regularly back up the MongoDB database to prevent data loss.	Restore the most recent backup and restart the affected components.
API endpoint failure (Flask backend).	Monitor the Flask backend to ensure stability during load testing and report generation. Add logging for errors.	Restart the Flask server or review and fix the API. Re-run failed tests if necessary.

6. References

1. Cypress Testing with React:

- https://youtu.be/6BkcHAEWeTU?si=0EtY0yHRH2_ARYIt

2. A Beginner's Guide to Unit-Testing With Jest:

- <https://medium.com/geekculture/a-beginners-guide-to-unit-testing-with-jest-549a47edda>

3. Performing MongoDB performance testing with JMeter:

- <https://www.linkedin.com/pulse/mongodb-jmeter-groovy-muhammad-atif-niaz-ms-se-bs-ce--wawef/>
- <https://blog.nonstopio.com/mongodb-performance-testing-using-jmeter-f98382844fbc>
- <https://www.blazemeter.com/blog/mongodb-performance>
- <https://www.dragonflydb.io/faq/mongodb-performance-testing-with-jmeter>

4. Flask backend testing with Pytest

- https://www.youtube.com/watch?v=qnrHgUoBmvM&ab_channel=TheDevWorld-bySergioLema