Software Testing Plan

Team: INSIGHT
Sponsor: Mike Taylor
Faculty Mentor: Scott Larocca
Team Members:
Joshua VanderMeer, Michael Vertin,
Aidan Hebert, Forrest Hartley

Date: 3/28/2025

Table of Contents

1. Introduction

Software testing represents a crucial phase within the software development life cycle. This is mainly to ensure that reliability, functionality, and performance requirements are all met. A well-structured testing plan is essential for ensuring each one of these criteria and validating that the software meets any additional stated requirements. This document aims to outline our overall approach for testing, detailing the methodologies and tools used to verify and ensure the integrity of our software.

Our testing approach includes comprehensive coverage of unit testing, integration testing and useability testing, each selected to cover a critical aspect of the implemented software system. Unit testing will primarily focus on validating individual frontend and server side components, ensuring each separate component functions correctly. Integration testing will then assess interactions between these individual components, verifying overall functionality and coherence. Finally, useability testing will assess the overall useability and intuitiveness of the software, directly aligning the final product with the end users expectations.

To achieve these goals regarding our testing coverage, it's essential to employ industry-standard testing tools that fit the given piece of software. In our instance, we will employ the Jest framework and the React Testing Library. Both of which will be used for ensuring unit testing is carried out for our frontend components. For facilitating our server-side functionality testing, we will be utilizing the Supertest framework. Our integration testing approach will focus on key interaction points between the server, client, and database to validate the overall system functionality and behavior. Our usability testing will involve real users interacting with the system to gather feedback and identify areas for improvement.

By implementing this structured testing approach, we aim to enhance software quality, improve system reliability, and deliver a robust solution that meets client expectations. This document provides a detailed roadmap for executing our testing strategy, ensuring that the final product aligns with the defined requirements and performs optimally in real-world scenarios.

2. Unit Testing

2.1 Introduction to Unit Testing

Unit testing is a fundamental software testing method used to ensure that individual units of source code work correctly on their own, independent of external systems or dependencies. The primary goal is to isolate each piece of logic in the system, verify that it behaves as intended under various conditions, and catch potential bugs early in the development lifecycle. Effective Unit testing leads to higher software quality, while also simplifying the process of making changes, and increasing confidence in the reliability of the application.

For our reverse image AI search tool, each unit test targets a discrete functionality. This might be file upload handling, server endpoint responses, or data transformations that occur within the server. Through these tests, we expect to confirm that:

1. Each function or component processes input data correctly.
2. The system gracefully handles valid, invalid, and boundary input cases.
3. UI components respond with the correct status and data.

We will be using Jest, a popular JavaScript testing framework for both our client-side and server-side tests, alongside Supertest for server endpoint testing. We will also employ the React Testing Library to test React components by simulating user interactions, such as file uploads and button clicks. For coverage tracking, we will use Jest's built-in coverage reporting to track which lines and branches of our code are executed by our tests. Our plan includes monitoring metrics such as:

- **Line coverage**: Percentage of executed lines in each file.
- **Branch coverage**: Percentage of executed logical branches (e.g., if/else paths).
- **Function coverage**: Percentage of functions invoked at least once during testing.

By analyzing these metrics, we can identify areas of code that need more thorough testing. Any module with minimal coverage will be prioritized for additional tests until we meet our coverage goals

2.2 Units Under Test

Our reverse image AI search tool handles both client and server functionalities. Below is an overview of what we consider our "units" for testing:

1. **Client (React) Units**

    - **File Input Handling**: Ensuring the file input field can receive images, properly processes them via a FileReader, and updates the preview state.
    - **Image Upload and State Management**: Confirming that the upload button triggers the correct API call, handles success and error responses, and updates UI accordingly (displays "Press Upload to Search," "Search successful," or error messages).
    - **Rendering of Search Results**: Verifying that neighbor images are rendered correctly when the server responds with an array of image data.

2. **Server (Node.js / Express) Units**

    - **/log_message Endpoint**: Checking that the server handles logging requests properly and returns a success response.
    - **/upload Endpoint**: Testing that the endpoint:
        - Correctly receives and processes base64-encoded images.
        - Forwards data to the external AI service (simulated in tests).
        - Returns a successful JSON response with neighbors or, in the case of failure, an appropriate error message.
    - **Logger Utility** (mocked): Ensuring side effects, such as writing to a log, do not cause unwanted issues in testing environments.

We focus most of our detailed tests on these critical paths. While there are other utility functions in the code, they are generally simpler and either tested implicitly or stable enough that we consider them low-risk areas.

2.3 Detailed Testing Plan

Our process for designing test cases includes identifying - equivalence partitions - grouping input values that should be treated similarly by the system - and boundary values - extreme ends of valid or invalid ranges that often expose edge-case bugs.

Below are key scenarios, organized by component or endpoint, along with example test inputs from the equivalence partitions and boundary values we will cover.

2.3.1 Client-Side Testing

1. **File Input Handling**
    - **Valid Image Files** (Equivalence Partition):
        - **Input**: A valid PNG or JPEG, or Webp file (e.g., test.png, ~100 KB).
        - **Expected Outcome**:
            - The FileReader loads the file, updates the preview to a base64 string, and shows the message: “Press Upload to Search.”
        - **Boundary Values**:
            - A very small file (e.g., a 1×1 pixel PNG)
            - A large file near the system’s maximum allowed size
    - **Invalid Files** (Erroneous/Robustness Partition):
        - **Input**: A text file (e.g., document.txt).
        - **Expected Outcome**:
          The application should ignore the file or display an appropriate error message telling the user to upload a valid image format.
2. **Uploading & State Management**
    - **Successful Upload** (Positive Path):
        - **Input**: A valid base64-encoded image.

        - **Expected Outcome**:
            - The UI sends the form data to the server and displays a "Search successful" message on receiving a 200 response.
            - Displays neighbor images in the search result list.
    - **Failed Upload** (Error Path):
        - **Input**: Simulated network error (thrown by axios.post).
        - **Expected Outcome**:
            - Display "Failed to upload image" without crashing the interface, demonstrating robust error handling.
3. **Search Result Rendering**
    - **Valid Neighbors** (Equivalence Partition):
        - **Input**: A server response containing 2 or more neighbor images.
        - **Expected Outcome**:
            - The application displays the neighbor images with correct src attributes set to their base64 strings.
    - **No Neighbors Found** (Boundary Condition):
        - **Input**: An empty neighbor list from the server, e.g., [].
        - **Expected Outcome**:
            - The application might display a message like "No similar images found" or show an empty state.

2.4 Summary & Rationale

Throughout this testing process, we will leverage:

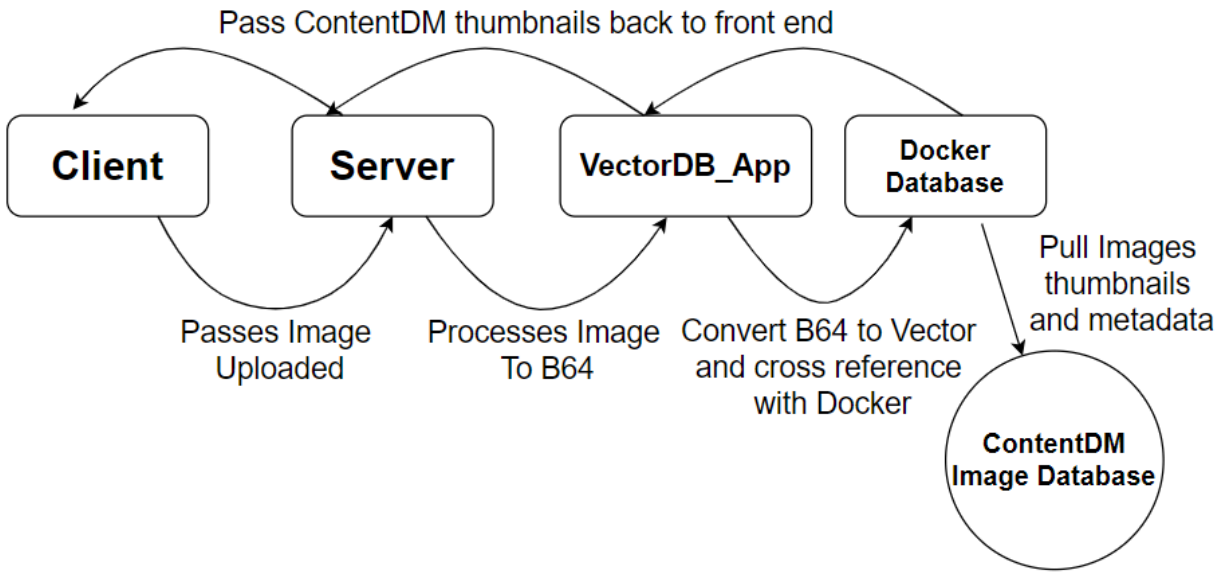
- Jest for its easy configuration, expressive test syntax, and built-in coverage collection.
- React Testing Library to ensure our UI is tested as a real user would interact with it (e.g., by selecting files, clicking buttons, and waiting for UI updates).
- Supertest to accurately simulate HTTP requests in our Node/Express environment, verifying the correctness of responses and ensuring no regressions as we evolve the server code.

We anticipate more unit tests to come as the codebase is tweaked and features are added. The addition of new unit tests in conjunction with regular assessment of current features should allow us to deliver a resilient and trustworthy reverse image search tool that meets the Cline Library's needs.

3. Integration Testing

3.1 Introduction

As you can see below, the modules of this project consist of five main parts that work together. The client handles the front-end display and functionality of the webpage, which then passes to the server running right behind the client, which will process data and pass it in the correct format to the VectorDB_App, which will convert the image to a vector to be cross-referenced with the Docker database. Once this is done it will put the correct nine images from contentDM as thumbnails and daisy-chain its way back to the client to be displayed.



Integration testing is a critical phase in the software development lifecycle that ensures the seamless interaction between different modules of components of a system. While unit testing verifies the correctness of individual functions or methods, integration testing focuses on

validating the communication and data flow between interconnected modules. The primary goals of integration testing include:

- **Verification of Data Exchange:** Must ensure that parameters, variables, return values, and shared data structures are correctly passed between modules.
- **Validation of Interfaces:** Confirms that APIs, function calls, and services adhere to their defined roles and modules.
- **Detecting System-Level Issues:** Helps to identify and resolve errors on a system level when dealing with interactions between modules.
- **Ensuring End-To-End Functionality:** Confirming that the integrated system behaves as expected under real-world conditions.

Our approach to integration testing will follow a pseudo-incremental approach, where intersections of modules will first be tested individually before proceeding to the full system testing. This approach will prioritize testing the key integration points, particularly where external dependencies (such as the databases on AWS and contentDM) are involved.

3.2 Integration Testing Plan

To systematically validate module interactions, we will test the following integration points:

3.2.1 Client-Server Communication

Test Cases:

- **TC-01:** Verify that an image uploaded via the UI is correctly converted to Base64 and transmitted to the Node.js server.
    - Test Steps:
        1. Upload a test image via the React frontend.
        2. Inspect the HTTP request payload to confirm Base64 encoding.
        3. Validate that the server (app.js) receives the payload without corruption.
    - Expected Result: The server logs confirm successful receipt of the image data.

- **TC-02:** Validate that search results returned from the server are correctly displayed in the UI.
    - Test Steps:
        1. Mock a server response containing a list of image metadata.
        2. Verify that the frontend dynamically renders thumbnails for each result.
    - Expected Result: The UI updates to display the returned images without errors.
- **Test Harness:**
    - Frontend: Jest + React Testing Library for UI interaction simulations.
    - Backend: Supertest for HTTP endpoint validation.

3.2.2 Server ↔ Vector Database Service

Test Cases:

- **TC-03:** Confirm that the server sends an image string to the Python service and receives embeddings.
    - Test Steps:
        1. Send a test image from app.js to vectordb_app.py via HTTP.
        2. Verify that the Python service processes the image and returns a valid embedding.
    - Expected Result: The embedding is a non-null numerical vector.

- **TC-04:** Validate that the server correctly relays search results from Milvus to the frontend.
    - Test Steps:
        1. Mock a Milvus response with sample image metadata.
        2. Confirm that the Node.js server formats and forwards this data to the UI.
    - Expected Result: The frontend receives structured JSON containing image references.

3.2.3 Vector Database Service ↔ Milvus

Test Cases:

- **TC-05:** Verify that embeddings generated by the Python service are inserted into Milvus.
    - Test Steps:
        1. Generate a test embedding using vectordb_app.py.
        2. Check Milvus logs to confirm successful insertion.
    - Expected Result: The embedding appears in Milvus with associated metadata.

- **TC-06:** Validate cosine similarity searches return relevant results.
    - Test Steps:
        1. Query Milvus with a known test embedding.
        2. Confirm that the top matches align with expected similarities.
    - Expected Result: The returned images are visually/contextually related to the query.

4. Usability/End-user Testing

4.1 Introduction

Usability testing is a type of testing in a controlled environment where users will test specific elements in a system. With usability testing, our goal is to identify issues that users would experience and receive feedback on how the system can be improved. The first phase of testing will involve researchers searching for images in the system. The second phase will involve the SCA adding images into the system.

End-user testing will focus on two classes of users: SCA employees and researchers. SCA employees will test both the search and insertion functionality. Researchers only need to test the search functionality.

4.2 Usability Testing Plan

4.2.1 Searching

For testing the searching component, we have no expectations for the experience with research of the test-takers. However, we do test with high and low levels of expectation of how accurate the results are. Because we do not have direct control over the quality of search results, the intensity of researcher testing will be low to moderate.

4.2.2 Insertion

For testing the insertion component, the test-takers will be members of the SCA. We expect them to have moderate to high levels of experience with command line usage, and high familiarity with SCA's ContentDM. Because the SCA is our client, the intensity of testing will be based on the different methods of insertions the SCA desires to use in the future.

4.3 Testing Plan

4.3.1 User Studies - Searching

Participants: people of different backgrounds
Test description: get images from some source, and use those images to test the system's search functionality
Frequency: After search-related updates
Measurements: Perceived accuracy, time required to access the system and perform the search, feedback
Analyzing measurements: identify the part of the process that is most time-consuming, discuss feedback in group meetings
Acceptance: Researchers should be able to attain the image(s) that they want from the SCA archives.

4.3.2 User Studies - Insertion

Participants: SCA representative (Mike Taylor?)
Test description: Test the SCA's ability to insert images into the database
Frequency: After database-related updates
Measurements: manual time, insertion time, errors, feedback
Analyzing measurements: prioritize manual time required, minimize time/knowledge required to fix errors, discuss feedback in group meetings
Acceptance: The SCA representative should be able to insert large data sets in a time frame they find reasonable.

4.3.3 User Studies - System Maintenance

Participants: SCA representative (Mike Taylor?)
Test description: Test the SCA's ability to handle errors/crashes
Frequency: After potential errors/crashes are discovered
Measurements: time to fix issues, difficulty, feedback
Analyzing measurements: minimize time/knowledge required to fix errors, discuss feedback in group meetings
Acceptance: The SCA representative should be able to resolve crashes or other issues with minimum difficulty or internal knowledge of the system.

5. Conclusion

The Software Testing Plan outlined in this document establishes a comprehensive framework to ensure that the system is functional, reliable, and user-friendly. Through a structured testing process that includes unit testing, integration testing, and usability testing, we aim to identify any and all potential defects before deployment.

Our Unit testing coverage ensures that individual components function as expected, with test coverage focused on core functionalities such as file input handling, server interactions, and search result rendering. By leveraging tools like Jest, React Testing Library, and Supertest, we maximize code reliability and maintainability. Integration testing further validates the

interactions between system components, ensuring seamless communication between the frontend, backend, and external services such as the vector database. By employing a pseudo-incremental approach, we systematically verify data exchange, API responses, and service interoperability, reducing the risk of failures in production. Finally, usability testing involves real users interacting with the system, providing valuable insights into its accessibility and efficiency. By collecting feedback and analyzing user behavior, we refine the system to enhance its usability and overall user experience.

Together, these testing methodologies create a robust quality assurance process that minimizes errors, optimizes performance, and ensures that the final product meets the expectations outlined in the requirements and design documents. By following this plan, we are confident that our software will deliver a high level of reliability and usability, ultimately supporting the project's success.