Testing Report: Testing Open-Source Application

Students Name:	Sowmya Talasila, Kevin Malone, Jennifer Tsan
Project/Application:	Screenshot to Code
Report Date:	December 8th, 2024

Table of Contents

1. **Introduction**
   1.1. **Test Project Name:** Screenshot to Code
   1.2. **Summary of the Rest of the Test Plan:**
   This report documents the testing activities, strategies, and results for the Screenshot to Code open-source application. It outlines the features tested, assumptions made, testing approaches used, specific test cases, and final recommendations based on test outcomes.
   1.3. **GitHub:**
   https://github.com/abi/screenshot-to-code

2. **Feature Description**
The Screenshot to Code application is a sophisticated tool designed to bridge the gap between design and development by converting visual representations into functional code. It uses advanced AI models and robust backend processing to deliver efficient and precise outputs. Key features are as follows:

API Integration
The application integrates with APIs from providers such as OpenAI and Anthropic. It handles:

- **API Key Management:** Securely stores and validates API keys for accessing AI models.
- **Provider Selection:** Allows users to switch between OpenAI and Anthropic models based on their preferences or requirements.
- **Error Handling:** Ensures appropriate responses for expired or invalid API keys to avoid disruptions during processing.

Code Conversion
This is the core functionality of the application. It includes:

- **Image to Code Translation:** Converts visual designs, including screenshots and mockups, into HTML, Tailwind CSS, React, or Vue code.
- **Support for Multiple Formats:** Processes images in various formats such as JPG and PNG.
- **Resolution Handling:** Adapts to varying image qualities (low, medium, and high resolution) to produce optimized outputs.
- **Scalability:** Ensures outputs are scalable and maintainable, suitable for integration into modern web development projects.

Frontend-Backend Communication
A robust communication mechanism ensures seamless interaction between the user interface and backend processes:

- **User Input Processing:** Captures and validates user inputs, such as image files and optional configurations.
- **Data Transformation:** Transmits user inputs to the backend, where they are processed using AI models, and returns the generated code.
- **Real-Time Feedback:** Provides users with instantaneous feedback on the status of their requests, including error messages or confirmation of successful code generation.

This feature set positions Screenshot to Code as a valuable tool for developers, enabling them to streamline the transition from design to implementation and save significant development time.

3. Assumptions

3.1. Test Case Exclusions

Certain test cases were excluded from this project due to constraints in time, resources, or scope. These exclusions are outlined below:

Advanced Performance Testing Scenarios:

- Performance testing to measure the system's behavior under high-load conditions or stress scenarios was not conducted.
- Such tests would require high-end systems and specialized performance testing tools, which were outside the scope of this project.
- Examples include testing the application's response time with hundreds of concurrent API calls or processing large batches of high-resolution images simultaneously.

Security Testing for API Calls:

- While API integration was tested for functionality and error handling, specific security tests such as token encryption, API misuse prevention, and authentication robustness were not included.
- These tests typically involve advanced penetration testing tools and dedicated expertise in API security, which were not part of the current project scope.

3.2. Test Tools, Formats, and Organizational Scheme:

Tools:

To ensure thorough and efficient testing, the following tools were employed:

- **Python unittest:** A lightweight framework for creating and running unit tests. It was primarily used to validate backend functionality.
- **Pytest:** A more advanced testing tool, Pytest was utilized for its flexibility and support for parameterized testing. It allowed for the creation of reusable and scalable test cases.
- **Manual Testing via CLI:** Some scenarios, particularly those involving visual validation or edge cases, were manually tested through the command-line interface to ensure accuracy.

Formats:

Consistent formats were used for input/output validation and result logging:

- **JSON Files:**
  - JSON was chosen for its readability and compatibility with the application's input/output processes.
  - It was used to define structured test inputs and verify the correctness of generated outputs.
- **Text Logs:**

    - Logs were maintained to capture the results of each test case, including details like inputs, expected outcomes, actual results, and identified issues.
    - These logs served as a reference for debugging and reporting.

Organizational Scheme:
To streamline the testing process and ensure systematic coverage, test cases were organized into distinct categories and groups:

- **Categorization by Functionality:**
    - Tests were grouped based on the feature or module being tested, such as input validation, backend processing, or integration testing.
- **Input Domain Modeling:**
    - Input variables (file formats, resolutions, API keys) were segmented into logical blocks for structured testing.
    - Techniques such as boundary value analysis and equivalence partitioning were applied to ensure comprehensive coverage.
- **Graph-Based Testing:**
    - Recursive functions in the backend were tested using graph-based models, which mapped decision points and execution paths.
    - This approach helped identify edge cases and verify the correctness of complex operations.

Summary
The assumptions made in this testing project allowed the team to focus on critical functionalities while acknowledging limitations in advanced performance and security testing. By using robust tools and structured formats, the testing process was organized to maximize efficiency and effectiveness, ensuring that the application met its primary functional and integration requirements.

4. Test Approach

4.1. Addressing Past Issues:
Past issues, such as improper input validation, were addressed by designing boundary and equivalence tests for the input variables.

4.2. Special Testing Considerations:
- Ensuring API calls don’t exceed allocated limits.
- Handling large file sizes during testing.

4.3. Test Strategy:
- **Input Domain Modeling:** Used to validate inputs (image types, resolution, and API keys).
- **Graph-Based Testing:** Tested recursive backend functions for truncating strings and processing nested data.

- **Tools:** Pytest for automation, manual inspection for visual output validation.

4.4. Test Categories

Functional Testing

This ensures that each feature of the Screenshot to Code application performs as expected. Test cases validate the core functionalities, such as:

- Correct conversion of images into code (HTML, Tailwind, React, Vue).
- Proper handling of API keys, including valid, invalid, and expired keys.
- Frontend-backend interactions, ensuring smooth input processing and output generation.

Boundary Testing

This focuses on testing the application's behavior at the edges of input ranges to identify potential issues. Examples include:

- Verifying how the application handles minimum and maximum file sizes.
- Testing image resolutions at both low and high extremes.
- Assessing API key validity for expired and nearly expired keys.

Integration Testing

This ensures that individual components work together easily. Test scenarios validate:

- Communication between the frontend and backend, ensuring correct data flow.
- Compatibility of the API integration with OpenAI and Anthropic models.
- End-to-end functionality, from user input to code generation and feedback delivery.

5. Test Cases

5.1. Test Group Definition

Test cases for the Screenshot to Code project are structured and organized into the following groups:

Input Validation Tests:

- **Objective:** Verify that the application correctly handles various input types, formats, and conditions.
- **Subgroups:**
  - Valid and invalid image formats (JPG, PNG, unsupported file types).
  - Image size and resolution boundaries.
  - API key validation (valid, invalid, expired keys).

Backend Functionality Tests:

- **Objective:** Ensure that backend processes work as expected, including recursive functions and data truncation.
- **Subgroups:**
  - Recursive handling of nested data structures.
  - String truncation for long input strings.

- Error handling for invalid inputs.

Integration Tests:

- **Objective:** Validate the integration between frontend, backend, and external APIs.
- **Subgroups:**
  - Data flow from user input to backend processing.
  - API integration with OpenAI and Anthropic.
  - Frontend output display accuracy.

Functional Testing:

- **Objective:** Verify end-to-end functionality of key features.
- **Subgroups:**
  - Successful image-to-code conversion.
  - Accurate mapping to specific frameworks (HTML, Tailwind, React, Vue).

5.2. Test Cases

Test Case ID	Test Group	Description	Expected Outcome
TC-01	Input Validation	Test with valid JPG image.	Converts image to code successfully.
TC-02	Input Validation	Test with unsupported file type (e.g., BMP).	Displays error message: "Invalid file type."
TC-03	Input Validation	Test with a high-resolution PNG image.	Converts image to code without performance issues.
TC-04	Backend Functionality	Process nested data structure with varying string lengths.	Truncates strings longer than 40 characters.
TC-05	Backend Functionality	Test recursive function on deep nested dictionaries.	Processes all elements correctly without errors.
TC-06	Integration Testing	Validate OpenAI API key handling.	Returns successful API response.
TC-07	Integration Testing	Test frontend-to-backend communication with valid input.	Correctly displays generated code on the frontend.
TC-08	Functional Testing	Convert screenshot to Tailwind CSS code.	Outputs accurate Tailwind CSS equivalent.
TC-09	Functional Testing	Convert screenshot to React code.	Outputs functional React component.

5.3. Traceability Matrix

Requirement	Test Case ID(s)	Test Description
Input validation for images	TC-01, TC-02, TC-03	Ensure valid image formats, sizes, and resolutions are handled.
Recursive backend functionality	TC-04, TC-05	Validate proper handling of nested structures and truncation logic.
API integration	TC-06	Verify successful communication with external APIs.
Frontend-backend communication	TC-07	Ensure smooth data exchange between components.
Code conversion	TC-08, TC-09	Verify accurate code generation for different frameworks.

6. Test Environment

6.1. Multiple Test Environments

Testing was conducted across:

- **Local Machine:** Windows 10, Python 3.9, 32GB RAM.

6.2. Schematic Diagram. Provide a schematic diagram of the test environment setup if applicable.

6.3. Test Architecture Overview

The testing architecture for the Screenshot to Code application was designed to ensure comprehensive coverage of all critical components. It adopted a modular approach to facilitate targeted testing, scalability, and maintainability. Each module focused on specific aspects of the system, using appropriate tools and methodologies to validate functionality, performance, and integration.

Key Components of the Test Architecture
Input Validation Module:
This module handled testing the application's ability to correctly process various types of user inputs.

- **Purpose:** Ensure that the system can accurately validate and handle different input scenarios, including image file types, resolutions, and API keys.
- **Focus Areas:**
  - **File Validation:** Test different file formats (JPG, PNG, unsupported types like BMP) to ensure correct acceptance or rejection.
  - **Resolution Handling:** Validate that the system performs optimally with low, medium, and high-resolution images, including edge cases.

    - **API Key Verification:** Ensure the system correctly distinguishes between valid, invalid, and expired API keys.
- **Testing Techniques:**
    - Automated boundary value analysis for file size and resolution limits.
    - Equivalence partitioning to group valid and invalid scenarios.
    - Manual testing for error messages and visual output validation.

Backend Processing Module:
This module focused on testing the logic implemented in the backend, particularly recursive operations.

- **Purpose:** Validate the robustness and efficiency of backend functions, ensuring accurate data processing and transformations.
- **Focus Areas:**
    - **Recursive Logic:** Test functions that handle nested data structures, such as truncating long strings within dictionaries or lists.
    - **String Truncation:** Validate that strings longer than 40 characters are truncated appropriately, maintaining accuracy and readability.
    - **Data Integrity**: Ensure that recursive operations do not alter original data unintentionally.
- **Testing Techniques**:
    - Graph-based testing to map out decision points and execution paths.
    - Automated test scripts using Pytest to simulate various nested structures and evaluate function outputs.
    - Stress testing to evaluate performance under complex data scenarios.

Integration Testing Module:
This module tested the interactions between the frontend, backend, and external APIs to ensure smooth communication and functionality.

- **Purpose:** Validate the interoperability of the system's components, ensuring that data flows smoothly and consistently.
- **Focus Areas:**
    - **Frontend to Backend Communication:** Test the ability of the frontend to correctly send inputs to the backend and receive accurate outputs.
    - **API Integration:** Ensure successful communication with OpenAI and Anthropic APIs, including error handling for failed API calls.
    - **Output Validation:** Validate that the code generated by the backend is correctly displayed on the frontend in the expected format (HTML, Tailwind, React, Vue).
- **Testing Techniques:**
    - End-to-end testing to simulate real user workflows.
    - Automated integration tests to verify data flow and response handling.
    - Mock testing to simulate API responses and validate error handling.

Additional Considerations

Scalability:
The modular architecture allowed for easy scaling of the testing process. For example, additional input types or backend features could be tested by extending the respective modules without impacting others.

Error Handling:
Special attention was given to error handling to ensure that the system responded gracefully to invalid inputs or unexpected scenarios, such as expired API keys or corrupted image files.

Test Coverage:
Each module incorporated specific coverage criteria to ensure all functional areas and edge cases were thoroughly tested:

- Node and branch coverage for recursive backend logic.
- Boundary value and equivalence coverage for input validation.
- Path coverage for integration scenarios.

Automation:
Automation was used to streamline repetitive test cases, allowing the team to focus on more complex scenarios that required manual validation. Tools like Pytest enabled efficient execution and reusability of test scripts.

Reporting and Logging:
Logs were generated for every test case, capturing details like test inputs, expected outcomes, actual results, and any anomalies. This data was vital for debugging and refining the system.

Summary
The test architecture for Screenshot to Code provided a robust framework to evaluate all critical components of the application. By breaking the system into distinct modules, the team ensured that each area received focused attention while maintaining a cohesive testing strategy. This approach not only validated the current system but also laid the groundwork for efficient testing of future enhancements.

6.4. Equipment Table. List the equipment and resources used in the testing environment if applicable.

Equipment	Description
Python	Programming language used for source code
OpenAI Key	Key needed to access the Generative Pre-trained Transformer 4 (GPT-4) model

7. Testing Results

Provide a summary of testing results, including passed, failed, and unresolved issues.

Test Case ID	Test Group	Description	Result
TC-01	Input Validation	Test with valid JPG image.	Passed
TC-02	Input Validation	Test with unsupported file type (e.g., PDF).	Passed; Did not allow unsupported files to be uploaded.
TC-03	Input Validation	Test with a high-resolution PNG image.	Passed
TC-04	Backend Functionality	Process nested data structure with varying string lengths.	Passed
TC-05	Backend Functionality	Test recursive function on deep nested dictionaries.	Passed
TC-06	Integration Testing	Validate OpenAI API key handling.	Passed
TC-07	Integration Testing	Test frontend-to-backend communication with valid input.	Passed
TC-08	Functional Testing	Convert screenshot to Tailwind CSS code.	Passed
TC-09	Functional Testing	Convert screenshot to React code.	Passed

8. Recommendations on Software Quality

Offer recommendations on improving the quality of the software based on testing results.

Backend results may not be the results intended by the developer due to lack of knowledge of functions. Documentation and commentation of functions and control flow between source files would be recommended to ensure accurate and effective test results. Also it's better to implement mechanisms to collect user feedback because some parts of the software are confusing and having this would let the developer get prompt feedback on what needs to be changed. This will let them work on fixing it as soon as possible.