Software Testing Plan

November 8, 2024

FLOOD BUSTERS

Team Members
Dylan Anderson, Jennie Butch, Noah Gooby
Nathan Hill, Jade Meskill

Sponsored By
Dr. Eck Doerry

Team Mentor
Vahid Nikoonejad Fard

Capstone Instructor
Isaac Newton Shaffer

Version
1.0

Overview: This document outlines the main implementations for the HydroCams software testing plan and process.

Table of Contents

1 Introduction

Flooding is the single most common and destructive natural disaster, causing over $3.7 billion in damage and claiming more than 120 lives annually across the United States. Nationwide, the frequency of disastrous flood incidents has more than doubled since 2000 and is expected to more than triple by 2050. Floodbusters is dedicated to addressing this problem through the development of a reliable, efficient, and adaptable image processing application that supports accurate marker detection and distance calculations across a variety of environmental conditions and marker configurations. This application can be used to easily generate calibration files for smart cameras, vastly lowering the amount of work and resources required to support automated flood detection. By providing features like multicolor marker detection, interactive zero-point calibration, and customizable configuration options, our application aims to meet or exceed the high demands of our client.

Software testing is paramount to make sure our application meets these high standards of functionality, accuracy, and usability. In general, software testing includes identifying and fixing any issues in a system that could impact users, guaranteeing that all components work as expected, and verifying that the system performs reliably under a variety of conditions. To achieve this, we will utilize a well-rounded testing approach that will help us validate each feature and interaction. We will employ unit testing, integration testing, and usability testing as the key pillars of our strategy, covering every aspect of the application, from individual functions and module interactions to the overall user experience.

As for our testing plan, we will use unit testing to address the accuracy and consistency of critical functions like image processing, UI elements, and JSON export, helping us verify that each functional block is sound as an individual component. Integration testing will focus on the interaction between these functions, guaranteeing that all components work together harmoniously. Lastly, we will utilize usability testing to observe real users' interactions with our application, helping us to further refine the user experience for ease of use and efficiency.

Given the intended use case of this application, we have tailored our testing plan to focus primarily on unit and integration testing to guarantee the accuracy and stability of core functionalities, while usability testing will focus on refining our interface and experience. This balance ensures that our most critical components are thoroughly validated while still receiving valuable user feedback. In the following sections, we will outline our specific approaches for each type of testing, discussing the strategies and processes that will allow us to achieve a high-quality final product.

2 Unit Testing

Unit testing is a vital part of our testing process. It allows us to ensure that key parts of our application work correctly as individual components. By testing these modules individually and in isolation, we can detect and address issues early on, ultimately providing a more stable foundation for the entire system.

We have decided to use Jest to test JavaScript-based units and pytest for the Python components. These tools are incredibly useful, supporting asynchronous testing and mocking, along with providing detailed logs. Our aim is to maintain high coverage across all critical components, focusing on modules like our Flask, Marker Detection, Marker Calculation, Canvas, Image Upload, Configuration, and JSON Export modules. These units are essential for providing accurate marker detection, distance calculations, and output data, demanding thorough testing.

For each module, we will define equivalence partitions and edge cases in order to create targeted test cases, along with explaining our success criteria and general execution plan. Combined, these unit tests will allow us to verify the resiliency and accuracy of our system.

2.1 Marker Distance Calculation Module (Jest)

The primary function tested here is the *recalculateDistances()* function, which computes the real-world distances between markers using pixel measurements, user-defined sizes, and a calculated scale factor. Since accurate measurements are absolutely crucial for the success of our application, this unit needs to be thoroughly tested to make sure it is capable of handling various distances and scale factors correctly.

2.1.1 Equivalence Partitions

1. Normal vertical distances: Cases where markers are within expected vertical ranges from the zero-point (e.g., within 15 feet).
2. Extremely close vertical distances: Cases where markers are very close to each other and/or the zero-point (e.g., within less than 2 inches).
3. Long distances: Cases where markers are outside expected vertical ranges from the zero-point (e.g., 30 feet).

2.1.2 Edge Cases

Edge cases such as a marker being placed precisely on the zero-point or on the edge of an image will be addressed. These cases should return appropriate results, such as a distance of 0 if a marker overlaps the zero point, or an accurate distance if a marker lies on the edge of the image.

2.1.3 Execution Plan

These test cases will be run in isolation, with known inputs and expected outputs. For this unit, we are aiming for a 90% accuracy rate with regard to the calculated distances. This will allow for slight discrepancies, such as 2” of error in a 36” measurement, but will otherwise expect precise calculations. All results will be recorded in Jest’s test logs, with our primary focus on verifying the accuracy of the results from each equivalence partition.

2.2 Marker Detection Module (pytest)

The key function in this module is the *detect_markers()* function, which algorithmically detects the markers in the user-uploaded image. This is accomplished using OpenCV, as well as parameters input by the user, such as a set of marker colors, and a minimum and maximum marker area (in pixels). The foundation of this project is the detection of markers, so this module must be tested rigorously.

2.2.1 Equivalence Partitions

1. Normal ratio of marker to distance: Markers are visible clearly in the image.
2. Small ratio of marker to distance: Markers are small and not clearly visible.
3. Large ratio of marker to distance: Markers are large and clearly visible in the image.
4. No detected markers: No objects matching the parameters are detected.

2.2.2 Edge Cases

Edge cases are difficult to define for this module, but could be created by using markers of a similar color to the background, or the markers being particularly near or far from the camera. However, these scenarios are highly unlikely to occur and would ultimately fall out of the scope of our project.

2.2.3 Execution Plan

Once again, these tests will be run in isolation, using testing images containing a known number of markers, of a known size and color. As the marker detection itself is heavily influenced by the distance from the camera to the marker, the accuracy rating of this module is hard to clearly define. pytest will output the results of these tests to the console, or optionally a log file.

2.3 Flask Server (pytest)

The Flask server plays a central role in our system, handling requests, processing images, managing configurations, and facilitating data transfer between the back and front end. We will focus its unit tests on verifying its response to valid and invalid requests, ensuring it returns the appropriate data or handles errors gracefully.

2.3.1 Equivalence Partitions

1. Valid image uploads: Requests with a valid image file (JPG, PNG) and all required parameters for color selection, contour area, and marker size. The server is expected to process the image and return data for markers, distances, and the image URL.
2. Invalid image uploads: Requests without an image or with an unsupported file type (e.g., PDF, TXT). The server is expected to respond with an appropriate error message and code (e.g., 400 Bad Request).
3. Valid configurations: Requests with valid configurations, such as marker size and contour area being appropriate values. The server is expected to accept the submission request without error.
4. Invalid configurations: Requests with missing configuration parameters or with values outside expected ranges (such as negative contour area or marker size). The server is expected to validate the inputs, reject those that are invalid, and return the appropriate error message.
5. Error simulations: Requests that result in excessive processing time. The server should limit the max processing time and return an error when it is exceeded.

2.3.2 Edge Cases

1. Empty requests: Requests with no data or files attached. The server should respond with a 400 error.
2. Extremely large image files: Uploads that are close to or exceed the servers processing limit. The server should either process it within a time limit or reject it gracefully with a corresponding error message.
3. Concurrent requests: Instances where multiple users are sending requests concurrently. The server should handle and process each request independently, without any errors or interference between requests.

2.3.3 Execution Plan

We plan to use pytest to execute these tests. Before each test, we will initialize a test instance of the Flask server. Mock image files and configuration requests will then be sent to simulate each of the aforementioned test scenarios. Finally, the test instance will be torn down, and any test-generated files or data cleaned up. Success will be measured by the server's response to each of the test cases, by gracefully handling errors and providing the expected outputs.

2.4 Configuration Module (Jest)

The Configuration Module provides users with a variety of settings to customize the image processing parameters, such as color selection, minimum contour area, and known marker size. This module accepts and validates user inputs, ensuring they are applied correctly. Testing will confirm that all configuration options are properly received, validated, stored, and retrieved without error.

2.4.1 Equivalence Partitions

1. Valid configuration updates: Changes of values within expected ranges for each parameter, such as valid color selections (RGB), minimum-contour areas with reasonable values, and marker sizes with typical dimensions. The module should accept and store these values without error.
2. Invalid color selections: Invalid or empty RGB values. This module should reject these values and provide feedback on why they are invalid and revert to default settings.
3. Out-of-bounds contour values: Invalid minimum/maximum contour values, such as negative values or values that are excessively large (greater than a reasonable maximum). These values should be rejected, and the user prompted to provide a valid input.
4. Out-of-bounds marker sizes: Marker size values that are either too large, too small, or negative. These values should be gracefully rejected, and revert to their default values.

2.4.2 Edge Cases

1. No configuration data: Using empty or null values for all fields. These should revert to their default values.
2. Non-numeric values for numeric fields: Providing non-numeric values like letters or symbols should result in the module rejecting these values and reverting to default values.
3. Concurrent updates with invalid values: Submitting configurations that have both valid and invalid fields should result in the module successfully applying the valid changes, while rejecting the invalid entries without affecting the overall configuration state.

2.4.3 Execution Plan

Each of these tests will be performed using Jest, as this module is written in JavaScript. Similar to the Flask module, a fresh instance of the configuration module will be initialized before each test, and all configurations reset after, to avoid any interference. Success will be measured through this module's handling of edge cases, and its ability to revert to default values when erroneous inputs are provided.

2.5 Canvas Module (Jest)

The Canvas Module is responsible for rendering and annotating images with marker outlines, labels, and distance lines. It is also responsible for handling user interactions such as panning, zooming, and selecting markers. Thorough testing of this module is required to ensure that it provides a smooth and intuitive user experience, free of error.

2.5.1 Equivalence Partitions

1. Valid pan and zoom levels: Cases where panning and zooming interactions occur within typical limits (such as not panning completely away from the image, and maintaining zoom between 0.5x and 4x). The canvas should be updated smoothly, while keeping markers and lines in proportion with the pan and zoom adjustments.
2. Extremely high/low zoom levels: Cases where the zoom is set to extreme values, such as approaching 0x or over 10x. The actual zoom level should be constrained between minimum and maximum levels, to prevent the canvas from becoming distorted or invisible.
3. Marker, line, and label updates: Cases where markers have been added, updated, or removed during pan and zoom states. These markers should update properly, along with their distance lines and labels.
4. Concurrent marker interactions: Cases where there are simultaneous updates to multiple markers. Each marker should be updated independently on the canvas without interference between them.

2.5.2 Edge Cases

1. Pan outside canvas boundaries: Applying a pan value that is outside the canvas boundaries. The canvas should immediately reposition the canvas back to center if the user tries to drag too far.

2. Rapid pan / zoom actions: Very fast, repeated panning and zooming actions. The canvas should respond fluidly without lagging, distorting elements, or misplacing markers/lines/labels.
3. Overlaying labels on canvas edge: Cases where markers are near the edge of the canvas view and may cause labels to overlap or go out of view. The module should respond by automatically adjusting its placement to remain visible.

2.5.3 Execution Plan

Jest will again be used to test this module, in addition to custom JS scripts to simulate rapid canvas interactions. We will initialize a fresh canvas before each test, and reset it afterward to maintain a consistent environment. Success will be measured by the ability of the canvas to smoothly handle each interaction, without noticeable lag or distortion. Furthermore, all annotations (markers, labels, lines) must display consistently in their correct position and scale regardless of zoom level.

2.6 Image Upload Module (Jest)

The Image Upload module allows users to upload images for CV processing, which is the starting point for our application workflow and is crucial for its core functionality. This module handles a variety of file types and sizes, provides feedback on upload success or failure, and supports smooth interaction with the back end.

2.6.1 Equivalence Partitions

1. Valid image formats: Cases where standard image formats (JP[E]G, PNG) are being uploaded. The module should upload these to the back end without error.
2. Invalid file types: Cases where non-image or otherwise invalid files are uploaded (PDF, GIF, TXT). These file types should be rejected, and a helpful error message displayed to inform the user of accepted file types.
3. Very large and very small files: Cases where very small (<500 KB) or very large (>50 MB) images are uploaded. Small files should be uploaded quickly, and large files should trigger a useful warning if the file exceeds a practical size limit. If it exceeds a maximum limit, the file should be rejected, and the user informed.

2.6.2 Edge Cases

1. Empty file upload: A user initiates upload without selecting a file. The module should detect this and inform them via an alert message that they must select an image first.

2. Extreme image dimensions: The uploaded image's dimensions exceed a reasonable threshold (e.g., a 10,000x10,000 image is uploaded). The module should reject the upload and inform the user of the maximum dimensions accepted.
3. Network error during upload: A network error (loss of connection) occurs during the upload process. The module should handle this failure gracefully and inform the user of the network failure, allowing them to retry the upload process without refreshing the page.
4. Corrupt image files: The submitted image has corrupted or incomplete data. The module should inform the user, reject the upload and allow them to try again.

2.6.3 Execution Plan

We will stick with Jest for this unit, which will be especially useful in mocking network errors. As with the other units, we will initialize a clean environment (no previous uploads, no images stored in back end) before each test case. Success will be based on this module's ability to correctly handle a variety of valid and invalid uploads, with graceful error handling and appropriate feedback for any invalid case.

2.7 JSON Export Module (Jest)

This module allows users to export processed marker data, including information on marker positions, dimensions, colors, and distances from other markers in a JSON format, which will ultimately be used for smart-camera calibration. Given its importance, this module clearly needs to generate accurate and readable JSON files that consistently align with the data presented within the application.

2.7.1 Equivalence Partitions

1. Standard export with valid data: Cases with a typical dataset where markers have all necessary attributes (position, size, color, and distances). A correct JSON file should be generated for download.
2. Minimal data set: Cases where there is minimal data (e.g., a single marker). The module should continue to export this data to a downloadable JSON file without error.
3. Maximal data set: Cases where the amount of markers and distance data reach the expected upper limits. The module should successfully generate a JSON file without any truncation, and experience no performance issues during the generation.
4. Invalid marker attributes: Cases where markers are missing attribute data. The JSON export should still complete without error, while including "N/A" for any missing or invalid fields.

2.7.2 Edge Cases

1. Extreme attribute values: Cases where markers have extreme attributes such as position coordinates, size, or distance values. These values should be accurately represented in the JSON file without altering the structure of the file.
2. Network or disk error during export: Cases where there is a network or disk error while generating or downloading the JSON file. The current file should be deleted, the user informed of the error, and allowed to retry the generation and download process.

2.7.3 Execution Plan

We will conduct these tests using Jest, with our main focus being data accuracy and structural consistency. Before each test, we will prepare mock marker data with varying attributes, and reset the environment after each test. Success will be measured by verifying that all JSON files are correctly structured, complete, and are generated/downloaded without error.

3 Integration Testing

Integration testing primarily focuses on verifying the interactions between the various modules within our workbench. This testing phase aims to ensure that data flows correctly between components, requests, and responses are handled correctly, and that the overall system behaves as expected when multiple modules are integrated. For this process, we will utilize two testing tools, Jest and pytest, depending on the specific module we are testing. These tools provide adequate code coverage for all modules.

3.1 Image Upload to Marker Detection Pipeline

This integration point verifies that the images uploaded through the workbench are received and processed by the marker detection pipeline correctly. This ensures that the upload interface is capable of processing the module without issue and that markers are being detected correctly.

Firstly, when a valid image is uploaded, it is expected to be processed and returned with the markers detected. Secondly, when an invalid image is uploaded, the workbench should not accept the file. Similarly, if the image is corrupted or of an otherwise invalid file type, the workbench should reject it. Finally, if the uploaded image does not have any detectable markers, the system should return a “No Markers Detected” response.

Jest will be used to create a testing environment and simulate file uploads. Once the environment is set up, the responses from the file uploads will be validated. These responses should help us ensure that the system handles the image upload process properly.

3.2 Configuration Module to Marker Detection and Distance Calculation

These integration tests examine the interactions between the configuration module and the marker detection and distance calculation modules. This allows us to confirm that the configuration settings are being applied correctly, and validate the behavior of the detection and calculation process.

When using valid configuration settings, it is expected that the system will return the correct marker information and distances, according to the configuration. If invalid configuration settings are supplied, the system is expected to reject the data, and potentially revert to the default values.

Jest will again be used to create a testing environment, and the various configurations used within. These configurations will be used to verify that the system accepts and rejects the proper values, and returns the anticipated results.

3.3 Marker Detection to Distance Calculation

This integration test focuses on verifying that data provided from the marker detection module (position, size, etc.) is easily accessible by the distance calculation module. The goal is to test the data flow and have the marker information align properly between the modules.

In our testing scenarios, we will gather a set of detected markers with known coordinates to verify that the distance calculation module correctly receives and computes distances from this data. We will also test this pipeline with a variety of test images in which the markers are very close to each other, or very far apart to ensure that our scaling methods are robust at various distances. Finally, we will simulate incorrect marker data, such as markers missing a coordinate, to confirm that the system rejects it and displays an error to the user.

To execute these tests, we will use known marker sizes and distances in a variety of images to ensure that our system reflects these within an acceptable margin of error. We expect the distance calculation module to return accurate data or appropriate error messages for all cases.

3.4 Canvas Module and Marker Interactions

This integration point will test how the canvas module interacts with the detected markers. We want to ensure that the markers are being displayed correctly, and that user interactions are being handled correctly.

Primarily, we want to ensure that the canvas renders valid markers correctly, displaying in the proper position and being undisturbed by user interactions like panning or zooming. Additionally, markers must be selectable by the user, and the interface display the corresponding details. Finally, a marker must be able to be deleted upon being selected, at which point it will be removed from the canvas and its information removed from the list.

In order to implement these tests, Jest will be used to simulate a canvas, and the user interactions with it. This allows us to verify that the canvas is interacted with and markers are rendered properly.

4 Usability Testing

Usability testing is critical in ensuring that our application is accessible, intuitive, and effective for users. This form of testing focuses on realistic user interactions, allowing us to observe how they navigate and operate the application to accomplish specific tasks. By allowing representative test-users to interact hands-on with our application, we can gather valuable feedback on its functionality, ease of use, and overall user experience.

4.1 Considerations

The target audience of the image workbench is members of the FloodAware project, who will be setting up and calibrating the camera hardware.

We will assume they have an adequate understanding of technology, enough to competently operate the workbench. The workbench has clear mechanisms for uploading and modifying images, and performing the necessary calculations with them. These design elements provide an easy-to-use, intuitive interface. After image modification and calculations are complete, the corresponding measurements and marker details are provided to the user.

4.2 Testing Plan

Workbench usability testing will be conducted through trials performed by our client, and potentially other members of the FloodAware team. This will give us a good idea of how real users feel about the overall user experience. We will also be surveying a group of our peers to get external opinions.

The users testing the software will complete the main tasks that the workbench is designed for. They will upload images, configure parameters, submit the image for CV marker detection and distance calculations, and download the generated calibration file.

A user survey will be the primary method of receiving feedback, which will give us information on how users feel about specific aspects of the workbench.

4.3 Timeline

We have begun the testing and validation phase of the project. Currently, we are addressing the remaining issues with the computer vision detection, and implementing additional features. Upon completion, we will begin the peer surveys, and are planning on completing them before the end of November.

5 Conclusion

Throughout this document, we have outlined our plans to rigorously test our product to ensure that we meet, or even exceed, the expectations of our client. In order to do so, we have to test each part of the product individually. Unit testing will be conducted using pytest and Jest, to verify the functionality of individual methods or functions, creating a solid code foundation for the rest of the product. Integration testing will evaluate the interactions between system components, ensuring that they can communicate without issue. Finally, usability testing will assess whether users can interact with the product effectively, providing an ideal user experience. By following this comprehensive software testing plan, we are confident that we can deliver a reliable, polished, and easy-to-use product to our client.