

5 Testing

5.1 Unit Testing

- For the frontend, we are using the React Testing Library for Unit testing. We are testing individual windows, interfacing between windows (such as changing views), and also testing for both regular and erroneous inputs. It sounds simple, and usually the tests are, but as the frontend continues to grow in scale, the number of different paths a user can take or buttons they can click rises significantly, requiring more unit tests.
- The drone software, written in C++, utilizes the Google Test library for automated unit testing. The major sections of the software being tested are the serial communication code (between the flight controller and the on-board computer), the camera controller code, the flight plan generation code, and the telemetry parsing and collection. Within each of these sections, individual functions are tested for common expected cases and possible edge cases, as well as for larger classes as a whole, to ensure that class state and methods interact correctly.
- The backend uses xUnit as its primary testing framework within the .NET environment. All tests are organized under the AutomationDrone.Tests project, which includes:
 - **Base/** – shared setup utilities for test classes
 - **Services/** – unit tests for core service-layer logic
 - **TestUtils/** – test data factories and helper objects

Service-layer tests focus on validating logic, data transformations, and interactions with the data layer. SQLite (in-memory) is used during testing to simulate database behavior and constraints, ensuring that service operations behave correctly without requiring a live database.

5.2 Interface Testing

The frontend UI is rendered using React for seamless testing and development. This allows frontend designers to make changes in the code and see their effect immediately, to determine if the output is what they envisioned or not.

The backend API is tested using Postman to simulate network POST, GET, PUT, and DELETE calls to the specified endpoints. This allows both the frontend and backend to determine which input fields are required for each database call, as well as verifying that their API calls will function correctly before they are implemented into the codebase.

The PostgreSQL database where we store the flight data is also viewable via the VM that we were provided by the university, and this allows us to view the data that we POST and PUT to make sure we are both sending the right data and also that our GET requests are receiving the correct data from the right endpoint.

5.3 Integration Testing

- Integration between the frontend and backend occurs through API requests, where the frontend sends flight and location data to be stored in the database. Testing for this involves simulating API requests using Postman and then replicating the same format as those successful requests to integrate into the frontend functionality. This is then tested at a later time individually to ensure functionality still behaves within the expected bounds, and if errors occur, that they are handled properly.
- Within the drone software, the interaction between all major components is tested end-to-end using our Google Test test suite with mocked interfaces. The software is used to communicate over serial with the drone flight controller and with camera hardware attached to the drone's on-board computer, so using these mocked interfaces allows us to test (to the extent possible) how the software as a whole works together with general inputs that we define, without needing to risk damaging the actual hardware with real-world tests.

The backend integration tests target the API layer, validating end-to-end request handling across controllers, services, and the database. Tests are implemented using xUnit for consistency with the existing .NET test environment.

These tests verify that the backend provides correct and stable responses to the frontend and the AI microservice. Critical integration paths include:

- **Flight Operations:** create, update, retrieve (single and list), and delete flights
- **Flight Data Operations:** create, update, retrieve (single and list), and delete flight data
- **Location Management:** create and delete locations

Each integration test issues HTTP requests against the API, ensuring the full request pipeline operates correctly, including routing, validation logic, service-layer interactions, and database access.

5.4 System Testing

For the AI vision subsystem, system testing will focus on validating the end-to-end pipeline from drone image input to the growth/weed/pest outputs consumed by the backend and dashboard. The goal is to ensure that, given a UAV image, the system produces correct detections and growth estimates, in the expected format, within the required latency.

At the unit level, we will test individual Python components, including image preprocessing (resizing, normalization, and tiling), model loading, inference wrappers, and post-processing logic (thresholding, class mapping, and growth percentage calculation). These tests will be implemented using pytest and will utilize small, fixed test images and mock model outputs to verify correctness and handle edge cases.

Interface tests will verify that the AI microservice API matches the contracts defined with the backend, including the correct HTTP method, request schema (image or image URL, crop type), and response schema (classes, bounding boxes/masks, confidence scores, and growth percentage). These tests will use automated API calls against a staging instance to ensure backward compatibility as the model evolves.

Integration/system tests will run the full pipeline using a held-out set of UAV images and ground-truth labels. For each test batch, we will compute accuracy/IoU and compare against target thresholds defined in the requirements. We will also measure end-to-end latency (from image received to JSON response) to verify real-time constraints. Model-level evaluation will use the built-in YOLO/Ultralytics validation tools, while higher-level system tests will be scripted in Python and integrated into our CI pipeline.

5.5 Regression Testing

All changes made to the project are posted to the Git repository as merge requests, which are then reviewed by other team members before being merged. The repository has CI pipelines set up, which detect the specific section of the project where changes were made (frontend, backend, drone software, or Terraform) and run all available automated tests on it, reporting the results back to the GitLab interface and blocking the request from being merged. Running the entire test suite for any change ensures that earlier features, which were previously unit/integration tested, will run again with the changes, allowing us to detect unintentional changes to earlier behavior.

Critical system features, as dictated by our project requirements, will be covered by these automated unit and integration tests wherever possible, thereby reducing the risk of testing new changes against the actual drone due to undetected regressions.

- Drone serial communication is extensively tested with unit tests and a mock serial port implementation in software to ensure the drone flight controller receives the commands it

expects. Additionally, it will test entire mocked flights with simulated data transmission between the mock flight controller and our drone software to ensure that changes do not cause issues with the autonomous flight capabilities.

- The frontend's critical features, such as field waypoints and data transmission to the drone, will also include automated testing in a similar form, with unit tests on a low level and integration tests against the entire flow as much as possible.
- The backend is set up to automatically test that every API endpoint functions according to the project requirements and that data persistence and storage are correct.
- The cloud infrastructure configuration utilizes Terraform, and merge requests that modify infrastructure will generate a plan (detailing the changes that will be made if applied) in the CI logs. This plan is reviewed before being applied, which helps uncover unintentional changes to previously deployed infrastructure.

5.6 Acceptance Testing

To demonstrate that our design requirements are being met, our drone will accurately fly along a pre-designated flight path and take pictures of the crops in that field. These pictures will then be uploaded to our app's frontend, which will take the flight data and pass it to our backend database. From there, our database will send the crop images to the AI model, where the trained model will analyze these pictures to determine if there are any diseases or pests that could limit, or otherwise damage, the crop's growth. To involve our clients in the acceptance testing, we will reach out to farmers and ISU farms to gauge their willingness to test our product and provide feedback, whether based on their UI/UX experience or the accuracy of the AI model responses. This feedback provides an iterative approach to improving the current design.

5.7 Security Testing

When considering security testing for our project, we must ensure that we cover every aspect of our project. The components we have identified are the frontend dashboard, backend (API), drone software, AI microservice, and cloud infrastructure. Because the system can give the drone commands to operate autonomously, it is crucial to prevent unauthorized access to ensure the integrity of the flight instructions and the data/information itself. We can use STRIDE as a guide for a threat model for this project to identify the risks. From there, we can ensure that each threat is assigned a specific test case, ensuring our testing covers all relevant security requirements.

Backend security testing can be performed using Postman & xUnit, where we validate authentication, authorization, and input validation by sending malformed payloads, invalid tokens, and large numbers of requests to ensure the backend handles potentially malicious behavior correctly. The drone software can be tested through services like Google Test and

mock serial interfaces, ensuring the onboard computer only accepts commands that are currently authenticated and formatted. For the frontend, React Testing Library and manual testing can be used to test secure token handling, form validation, and unauthorized page access. The AI microservice can have similar testing for payload validation and endpoint isolation checks.

For infrastructure, Terraform plans and CI-based security checks can verify that our cloud resources are configured correctly, adhering to the principle of least privilege. Static analysis tools can be used for this (such as SonarCloud, ESLint, or cppcheck), but probably won't be throughout this project.

Altogether, these efforts can ensure that security is a priority in our project and is implemented throughout the entire development process.

5.8 User Testing

Our project will undoubtedly involve extensive user testing, as it will be used exclusively by farmers. We want to ensure that our product flow, UI, drone, and everything else are usable, safe, and accessible to users without requiring advanced technical skills to understand how to use them. We will be focusing on asking farmers how they feel about the product, the UI, and the drone. Their input is invaluable to us as they are the target audience of our product. Once we have completed most of the product's functionality, we plan to solicit feedback from farmers, asking them if we can survey their fields and what they think about the product. Their suggestions will be taken seriously and implemented if possible. We want to create a product that will actually be used, and if our users aren't happy with it, then they won't use it.

5.9 Results

Thus far, we have successfully implemented a backend API and integrated it with the frontend user interface. We can successfully send and retrieve flight and location data, provided the proper flight ID is provided, allowing users to access only their own flight data. This will later be expanded to include authentication and user accounts, allowing us to track flight data per user rather than users per flight data.

At this point in the semester, we have not received the drone parts to assemble the drone and begin testing with a flying drone. However the plans for testing a flying drone begin with us mounting the drone onto a test stand to test the pitch, roll, and yaw to make sure our PID loop has been controlled. Next is to test indoors while the drone is tethered to the ground to ensure that the drone does not accelerate into the air when the drone is outside. We will then test the drone indoors without the tether to test functionality before flying outside. There will be a kill

switch implemented to the motors so that the drone can stop flying immediately in the event of unexpected behavior. Plans for a drone return after flying too far is considered as well.

A supplementary Raspberry Pi and camera has been provided to begin testing drone software. At this point, actual working hardware testing has not been completed yet.