COCKPIT SWITCH AND INDICATOR TESTING TOOL

PROJECT DOCUMENTATION

JFJ Aviation & Defence GmbH
Krishna Sharma
June 2025

INDEX

Chapter 1: Introduction

In modern aviation systems, rigorous validation of cockpit controls and indicators is essential to ensure safety, compliance, and system integrity. This project presents a Cockpit Switch & Indicator Testing Tool — a lightweight, browser-based application that streamlines the process of verifying the behavior of cockpit switches and indicator lights connected via an OPC UA (Open Platform Communications Unified Architecture) server.

The tool was built with the goal of providing real-time validation, test automation, and clear visual feedback for both input and output hardware elements. It is designed to simulate the end-to-end testing process for cockpit switches (e.g., toggles, rotary selectors) and indicators (e.g., LED alerts) in an intuitive and user-friendly format that can be used during system installation, quality checks, or maintenance.

At its core, the tool loads a structured set of test definitions (defined in a JSON file), categorizes them into logical panels (e.g., *Flight Control Panel*, *PTMS Panel*), and then allows users to either:

- Manually execute and validate each test step one by one, or
- Automatically poll and evaluate the expected behavior in real time, without user intervention.

Furthermore, the tool maintains a detailed progress report both globally and per panel, supports stateful test tracking, and enables interaction with the OPC UA server for reading input signals and writing output commands (e.g., activating a light). It is especially suited for hardware validation in test benches or simulated cockpit environments.

This document outlines the design, architecture, and features of the tool in detail, explaining how it integrates frontend and backend components to achieve its goals.

Chapter 2: Project Objectives

The primary objective of this project is to create a robust and intuitive tool that enables efficient testing of cockpit controls and indicators through real-time interaction with an OPC UA server. This tool is intended for use by engineers, testers, or technicians during system integration, validation, or troubleshooting.

The specific goals of the tool are as follows:

2.1 Functional Objectives

i) Real-time Input Monitoring
Read switch positions or input values from the OPC UA server and verify if they match expected conditions (e.g., *"Change to 1"*, *"Increase in value"*, etc.).

ii) Controlled Output Activation
Send commands to the OPC server to activate hardware outputs (e.g., turning on an indicator light), and prompt the user to visually confirm behavior.

iii) Support Both Manual and Automatic Test Modes
- Manual Mode: Users initiate and validate each test manually.
- Auto Mode: The system polls test inputs and proceeds automatically when expected outcomes are detected.

iv) Progressive Test Unlocking
Enforce a sequential testing strategy by locking tests until the previous one passes — ensuring structured validation.

v) Panel-Based Grouping of Tests
Organize tests by cockpit panel (e.g., *Flight Control Panel*, *PTMS Panel*), with separate progress tracking and summary statistics per panel.

vi) User Feedback and Logging
- Display clear status indicators (Pending, Passed, Failed).
- Log all test results to a CSV file with timestamps for traceability and auditability.

vii) Handle Both Input and Output Logic Seamlessly
Distinguish between test types and apply appropriate logic for polling, writing, and confirmation.

vii) Minimal Setup, Web-Based Interface
Make the tool accessible from any modern browser without requiring additional software or plugins.

2.2 Non-Functional Objectives

i) Extensibility
Easily configurable through a single JSON file — making it simple to add new panels, switches, or expected behaviors without touching the core logic.

ii) Portability
Runs on any machine with Python and Flask, including development boards like Raspberry Pi.

iii) Visual Clarity
Clean, pastel-colored UI with dark mode support for use in testing environments.

iv) Robustness
Fault-tolerant against OPC read/write errors and designed to gracefully handle missing nodes or test mismatches.

Chapter 3: Architecture Overview

The Cockpit Switch & Indicator Testing Tool is structured as a modular Flask-based web application with clear separation between:

- Frontend interface (HTML/JS/CSS)
- Backend logic (Flask/Python)
- OPC UA communication (via *opcua* Python library)
- Data/configuration layer (JSON test definitions)

The tool is composed of the following key components:

3.1 High-Level Workflow

i) Load Test Plan
On startup, the application reads a structured JSON file that defines all cockpit switches and indicator tests, grouped under named panels.

ii) Panel Selection
The user is presented with a homepage listing all available panels, each with its own progress bar and set of tests.

iii) Test Execution
Users can select a panel and test each switch manually (by pressing a button), or let the system run tests automatically by polling the OPC UA server for expected results.

iv) OPC UA Communication
All read/write operations are handled by a wrapper class (*OPCReader*) that abstracts the underlying OPC UA communication logic.

v) Progress Tracking & Logging
Each test is marked as "passed" only when the actual value from the server matches the expected result. Results are logged to a CSV file.

3.2 System Components

i) *test_plan.json* (Configuration)

- Defines all switches and their tests.
- Each test contains metadata like *Id*, *Text*, *Picture*, *Expected value*, *OPCNode*, and *type* (input or output).

ii) *app.py* (Backend Server)

- Flask app that:
    - Loads the JSON test plan and flattens it into panel groups.
    - Serves HTML pages via Jinja templates.
    - Exposes routes for:
        - Reading switch values
        - Writing output values
        - Confirming output test status
        - Resetting progress
        - Tracking test progression

iii) *switch_reader.py* (OPC UA Interface)

- A Python class (*OPCReader*) that:
    - Connects to an OPC UA server.
    - Reads node values (*read_node*).
    - Writes output values with correct type handling (*write_node*).

iv) *templates/*.html* (Frontend)

- Jinja2-based HTML templates:
    - *index.html* — Home screen showing all panels and progress bars.
    - *panel.html* — Displays all tests in a given panel.
    - Dynamic controls to run tests, activate outputs, confirm results.

v) *static/js/ and Inline Scripts**

- JavaScript handles:
    - Polling in auto mode
    - Dynamic test display
    - Progress updates
    - Skip/confirm logic
    - Output test sequencing

vi) *test_log.csv* (Logging)

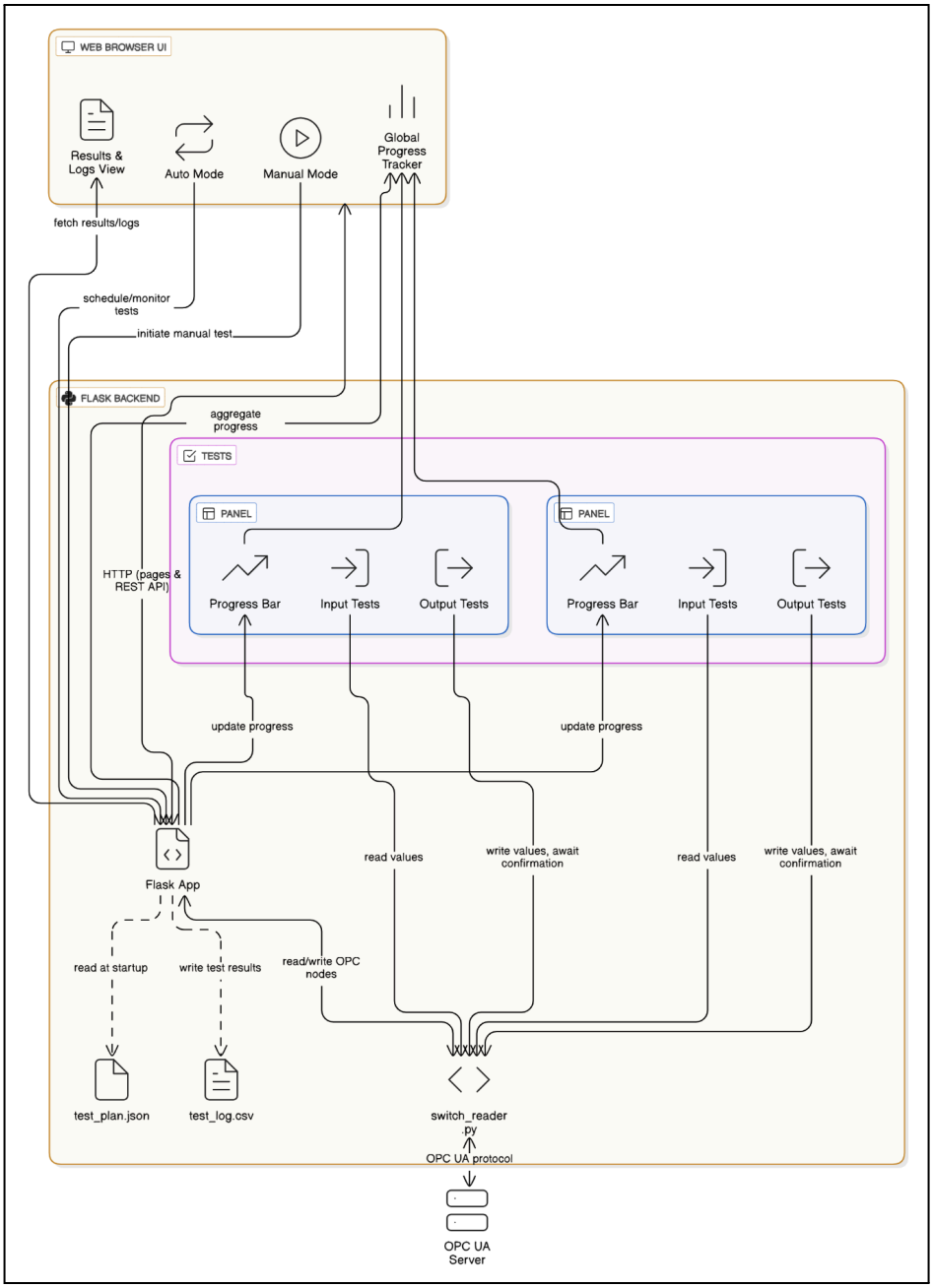
- A flat CSV log of every test attempt.
- Contains timestamp, test ID, actual vs expected value, and whether it matched.

3.3 Deployment Environment

- **Backend**: Python 3.x + Flask

- **OPC Communication**: *opcua* Python package
- **Frontend**: HTML5, Bootstrap 5, JavaScript
- **Device Compatibility**: Raspberry Pi, Linux, Windows
- **Network Requirements**: Access to OPC UA server IP (e.g., *opc.tcp://192.168.60.101:4840*)

The system architecture diagram of the project is shown in Fig 1.



[Fig 1: Architecture Diagram of Project]

Chapter 4: Features and Functionality

The Cockpit Switch & Indicator Testing Tool incorporates a range of features designed to streamline and automate the testing process of cockpit panels. Key functionalities include:

4.1 Panel-Based Testing Structure

- Tests are organized by cockpit panels (e.g., *Flight Control Panel*, *PTMS Panel*), allowing focused and manageable testing sessions.
- Each panel has its own dedicated page displaying all associated switches and tests.

4.2 Support for Input and Output Tests

- Input Tests: Monitor and validate cockpit switch positions or sensor states by reading values from OPC UA nodes and comparing them with expected conditions
- Output Tests: Send commands to activate outputs (e.g., indicator lights) and prompt the user for visual confirmation, ensuring correct hardware response.

4.3 Manual and Automatic Testing Modes

- Manual Mode: Users manually trigger each test by clicking buttons, allowing flexible and user-driven verification.
- Automatic Mode: The system continuously polls OPC UA nodes and automatically progresses through tests once expected results are detected, minimizing human intervention.

4.4 Progressive Test Unlocking & Sequential Validation

- Tests are locked by default and only unlocked once the previous test in sequence passes, enforcing a logical order of validation.
- Skipped tests in auto mode are revisited after initial tests complete, ensuring comprehensive coverage.

4.5 Real-Time OPC UA Communication

- Seamless integration with an OPC UA server to read input node values and write output commands.
- Robust error handling ensures graceful recovery from communication failures.

4.6 User Interface & Visual Feedback

- Intuitive web interface built with Bootstrap for responsive design.
- Clear status indicators (Pending, Passed, Failed) with badges and color codes.
- Visual aids include images illustrating expected switch positions or indicator states.
- Dark mode support for user comfort in varied lighting conditions.

4.7 Detailed Progress Tracking

- Individual panel progress bars update dynamically as tests pass.
- Global progress bar on the homepage provides an overview of all panels' completion status.
- Progress updates occur automatically without requiring page refresh.

4.8 Logging and Audit Trail

- Every test execution is logged with timestamp, test ID, current and expected values, and pass/fail status.
- Logs are stored in a CSV file for easy review and traceability.

4.9 Flexibility and Extensibility

- Test configurations are stored in a JSON file, enabling easy addition or modification of panels, switches, and test cases without changing the codebase.
- Supports multiple types of expected test behaviors such as value changes, increases/decreases, and output activation sequences.

Chapter 5: Implementation Details

This section covers the core technical aspects of how the Cockpit Switch & Indicator Testing Tool is implemented, including the main components, data handling, and logic flow.

5.1 Backend - Flask Application (*app.py*)

- **Initialization:**
  On startup, the Flask app loads the *test_plan.json* file, parsing all switch and test definitions into a flat list (*flat_tests*). Each test is augmented with metadata such as panel name, switch name, and a *passed* flag initialized to *False*.

- **Routing and Views:**
  - */* — Home page displaying all panels with their overall progress bars.
  - */panel/<panel_name>* — Displays the individual panel page with tests listed, supporting both manual and auto modes (controlled via URL query parameter).
  - */read/<test_id>* — Reads the current value from the OPC UA server for a given test ID, compares it against expected criteria, and updates pass/fail state accordingly.
  - */confirm/<test_id>* — Confirms an output test as passed based on user input (e.g., user confirming an indicator light is ON).
  - */output-step* — Manages output test phases by writing ON/OFF values to OPC UA nodes.
  - */reset/<panel_name>* and */reset-all* — Reset progress flags for a panel or all tests.
  - */get_overall_progress* — Provides real-time JSON progress data for dynamic frontend updates.
  - */next-panel* — Returns the next panel with pending tests to support automated navigation.

- **Progress Tracking:**
  Test pass states are stored in memory during runtime. Frontend requests progress data regularly to update progress bars without requiring page reloads.

- **Logging:**
  Each test attempt is recorded in a CSV log file with timestamps, test IDs, actual vs expected values, and match results.

5.2 OPC UA Communication (*switch_reader.py*)

- Encapsulated in the *OPCReader* class, which handles:
    - Connection management to the OPC UA server.
    - Reading node values with error handling and logging.
    - Writing node values for output tests, including automatic conversion to appropriate OPC UA data types (Boolean, Int32, Float).

5.3 Frontend Interface (*panel.html*, *index.html*)

- Uses Jinja2 templating to dynamically render panels, tests, and progress indicators based on backend data.

- **Manual Mode:**
    - Users click buttons to run individual input tests or activate outputs.
    - Test cards unlock progressively as tests pass.
    - Visual status badges indicate test results.

- **Automatic Mode:**
    - The frontend polls */read/<test_id>* endpoints repeatedly.
    - Tests automatically proceed to the next when a pass condition is met.
    - Output tests involve multi-phase activation with server commands and user confirmations.
    - Skipped tests are re-queued to ensure all tests are eventually evaluated.
    - Provides a fullscreen single-test view with smooth transitions.

- **JavaScript Logic:**
    - Handles polling, test state updates, progress bar refreshes.
    - Implements “skip” functionality in auto mode.
    - Manages output test sequences with separate confirmation steps.
    - Includes utility functions to update the UI dynamically and maintain test flow.

5.4 Data Storage and Configuration

- The entire test plan is stored in a human-readable JSON file, facilitating easy edits without code changes.
- Logging uses a simple CSV format, which can be imported into spreadsheets or analysis tools for audit and review.

5.5 Error Handling and Robustness

- OPC UA communication failures are caught and logged, preventing crashes.
- UI disables buttons appropriately to avoid repeated or invalid actions.
- Progress and state are kept consistent across manual and automatic modes by sharing a common *passed* flag per test.

Chapter 6: Challenges and Solutions

During the development of the Cockpit Switch & Indicator Testing Tool, several technical and design challenges were encountered and addressed to ensure a robust and user-friendly system.

6.1 Challenge: Real-Time Synchronization of Test Status and Progress Bars

- **Problem:** The progress bars on individual panel pages and the global progress bar did not update automatically after a test passed, requiring manual page refreshes.
- **Solution:** Implemented asynchronous JavaScript polling to fetch updated progress data from backend endpoints (*/get_overall_progress* and refreshed panel progress). The frontend dynamically updates progress bars and test statuses in real time without requiring a reload.

6.2 Challenge: Activating Output Tests via “Activate Light” Button

- **Problem:** The "Activate Light" button was not triggering backend calls correctly, leading to no action or feedback.
- **Solution:** Fixed event binding issues in the frontend JavaScript to ensure the *activateOutputManual* function is properly called on button click. Added debugging logs and ensured the */output-step* endpoint is invoked with correct parameters. Verified successful OPC UA node writes.

6.3 Challenge: Ensuring Correct Test Order Execution

- **Problem:** Tests belonging to the same panel but defined in different JSON switch objects were not executed in the intended sequence, leading to premature marking of panel completion.
- **Solution:** Flattened and sorted the test list to preserve the order defined in the JSON file. Grouping by panel respects the test sequence by ordering tests as they appear. Added locking logic to only unlock tests sequentially after the previous one passes.

6.4 Challenge: Managing Skipped Tests in Auto Mode

- **Problem:** Tests skipped during automatic mode risked being permanently ignored, compromising test completeness.
- **Solution:** Designed a re-queuing mechanism that appends skipped tests back to the test list for later evaluation, ensuring all tests eventually run or are manually handled.

6.5 Challenge: OPC UA Data Type Compatibility

- **Problem:** Writing to OPC UA nodes sometimes failed due to incorrect data types being sent.
- **Solution:** Enhanced the OPCWriter utility to detect OPC UA node data types and convert input values accordingly (Boolean, Int32, Float), preventing type mismatch errors.

6.6 Challenge: User Experience in Multi-Phase Output Tests

- **Problem:** Output tests require multiple confirmation steps (turning lights ON and OFF), which complicates the UI flow.
- **Solution:** Developed a clear step-wise interface in auto mode that guides the user through each phase with distinct buttons and feedback messages, improving clarity and reducing errors.

Chapter 7: Conclusion

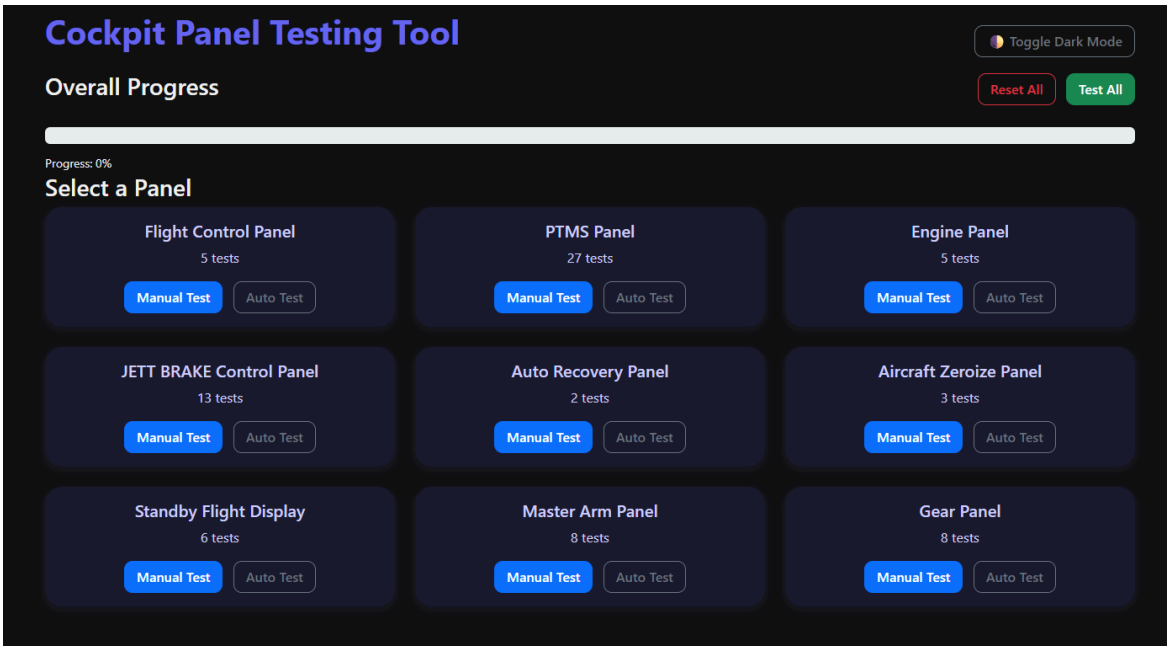
The Cockpit Switch & Indicator Testing Tool successfully automates the verification of cockpit switches and indicators by integrating OPC UA communication with a user-friendly web interface. This tool streamlines the testing workflow for complex aerospace panels, ensuring that both input switches and output indicators are thoroughly validated with minimal manual intervention.

By combining manual and automatic testing modes, progressive reveal of tests, real-time feedback, and detailed logging, the system offers flexibility and reliability suited for rigorous testing environments. Despite initial challenges with synchronization, test sequencing, and OPC UA data handling, thoughtful design and iterative improvements resulted in a robust and extensible solution.

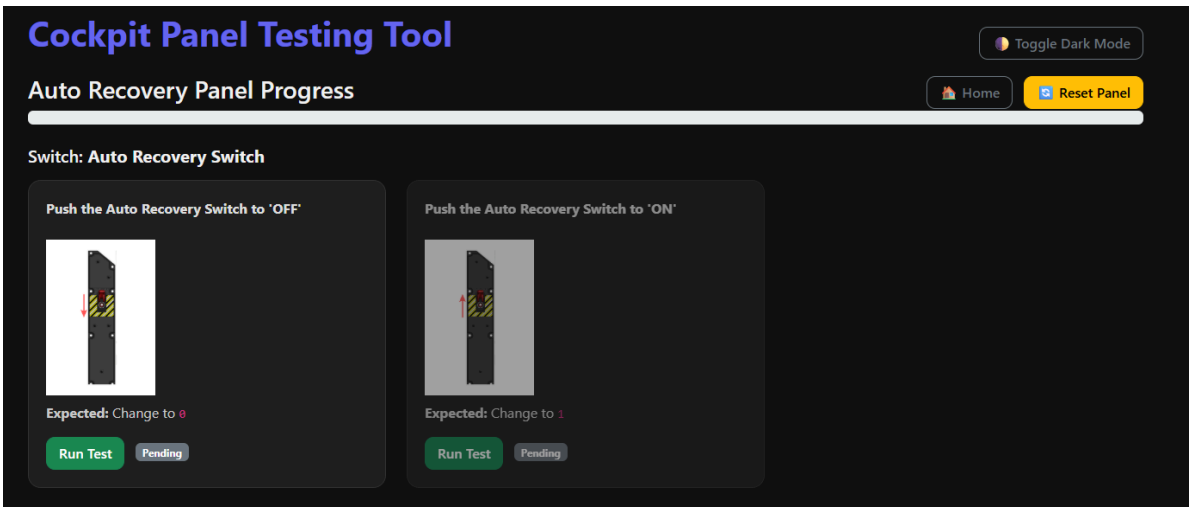
The modular architecture based on a JSON-driven test plan and clear separation between backend logic and frontend presentation facilitates easy maintenance and scalability. Future work can expand support for additional hardware interfaces, enhance user experience, and incorporate advanced analytics on test results.

Overall, this project demonstrates the effectiveness of combining modern web technologies with industrial communication protocols to build practical, domain-specific testing applications.

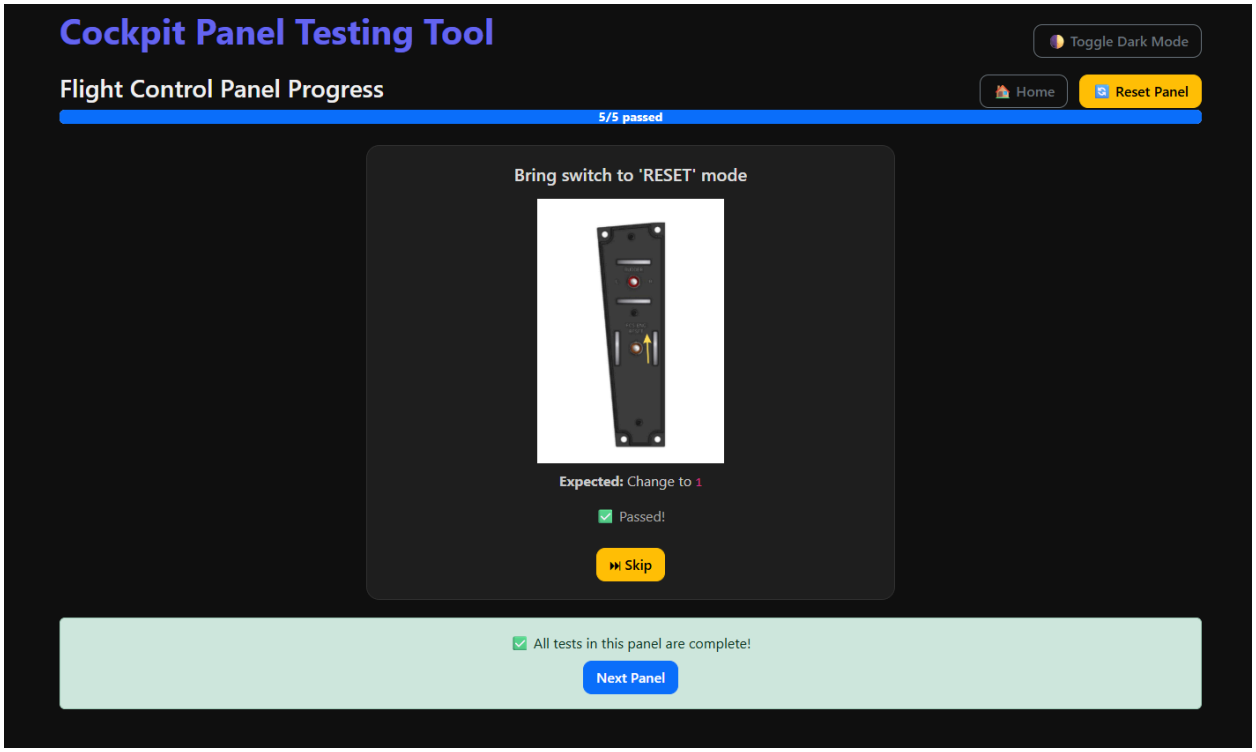
Chapter 8: Screenshots



[Fig 2: Landing page]



[Fig 3: Example of panel in manual mode]



[Fig 4: Example of panel in auto mode]