TED UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

Test Plan Report

CMPE 492 – Senior Design Project II

by

Berk Kaya

İlhan Ün

İrem Ayça Uçankale

Alperen Aktaş

Onur Turan

1. Introduction **3**
1.1 Purpose and Goal of the Test Plan 3
1.2 Scope of Testing 3
1.3 Testing Approach Overview 4
2. Test Items and Features **4**
2.1 Items Being Tested (System Components/Modules) 4
2.2 Features to be Tested 5
2.3 Features Not to be Tested (Out of Scope) 5
3. Testing Methodology **6**
3.1 Unit Testing 6
3.2 System and Integration Testing 6
3.3 Performance Testing 6
3.4 User Acceptance Testing (UAT) 6
3.5 Beta Testing 6
4. Test Environment **6**
4.1 Hardware Requirements 6
4.2 Software Requirements 6
4.3 Tools and Testing Frameworks 6
5. Roles and Responsibilities **7**
5.1 Testing Tasks 7
5.2 Personnel Assignments 7
6. Schedule and Resources **7**
6.1 Test Schedule 7
6.2 Resource Allocation 7
7. Control Procedures **7**
7.1 Defect Reporting and Tracking 7
7.2 Change Management 7
8. Risks **7**
8.1 Associated Testing Risks 7
8.2 Mitigation Strategies 7
9. Test Cases **7**
10. References **7**

1. Introduction

This document defines the scope, approach, resources, and schedule of the testing activities to be conducted for the Figion system. Since the system involves quality control and artificial intelligence integration, this test plan aims to verify the system's reliability under industrial conditions.

1.1 Purpose and Goal of the Test Plan

The Figion project aims to automate the detection of aflatoxin contamination in dried figs using computer vision and edge-computing infrastructure. The main goal of this test plan is to verify that this system, which will replace the existing manual workflow, meets stringent industrial and safety requirements. The core objectives focused on by the tests are:

- **Real-Time Performance (Latency):** Testing whether the system operates with a maximum latency of 1 second per fig (including image capture, inference, and displaying the result) and matches the manual processing rate of 175 kg per hour.
- **High Reliability and Accuracy:** Verifying that the system achieves a minimum Recall (Sensitivity) rate of 95% in aflatoxin detection to prevent public health risks and food safety violations. Additionally, testing that the Precision rate is a minimum of 85% to **prevent commercial losses caused by false positives.**
- **Data Integrity and Traceability:** Verifying that high-resolution images are saved directly to the file system for each scanning session, and decision results (score, label, timestamp) are saved to the SQLite database without loss, in compliance with ACID standards.

1.2 Scope of Testing

The testing process will cover both the edge hardware components and the software subsystems of the system.

- **Software Scope:** Tests will be conducted on the Vision package hosting the YOLOv11n (in ONNX format) inference engine, the Control package managing hardware communication, the Data Management package handling database (SQLite) and file I/O operations, and the PyQt6-based desktop User Interface (UI) package.
- **Hardware Scope:** The integration of the USB camera video feed within the dark room (test box) isolated from external light, the 365nm high-intensity UV LED array, and the USB relay board controlling these lights with the system will be tested.
- **Out of Scope:** Since the system will operate offline on standard computer hardware using edge computing principles rather than cloud computing, testing external cloud services or server connections is out of scope.

1.3 Testing Approach Overview

Because the Figion system architecturally has a concurrent and multi-threaded structure, tests will be executed progressively in accordance with this modular design (Producer-Consumer pattern and Qt Signal & Slot).

- **Unit Testing:** Because the dependencies among the Vision, Control, and Data packages in the system are minimized, each package will be tested in isolation. The Vision package will be tested for speed and accuracy using pre-recorded images (test dataset) without launching the UI, while the Data Management package will be tested for SQL integrity using sample records.
- **Concurrency and Performance Testing:** To ensure that interface updates and image processing run smoothly at the same time, it will be verified that the main thread (UI Thread), the background thread (AI Worker Thread), and image saving services (ImageArchiver) operate without blocking each other (non-blocking).
- **Boundary Conditions and Error Handling (Stress Testing):** The system's ability to transition to a safe state will be tested by simulating edge cases such as hardware initialization failures (camera not found), full storage (Disk full), or excessive conveyor speed (Frame Dropping).

2. Test Items and Features

2.1 Items Being Tested (System Components/Modules)

The testing activities will evaluate the modular subsystems and edge hardware components that make up the Figion architecture. The specific items to be tested include:

- **Vision Subsystem (Package):** The core analytical component, which includes the CameraManager, Preprocessor, InferenceEngine (running the YOLOv11n ONNX model), and PostProcessor classes.
- **Control Subsystem (Package):** The operational management layer, encompassing the SystemController, StateManager, HardwareController (managing the UV relay), and HardwareMonitor components.
- **Data Management Subsystem (Package):** The persistence layer, which consists of the SessionManager, DatabaseHandler (for SQLite interactions), ImageArchiver (for file system storage), and ReportGenerator.
- **User Interface (UI) Subsystem (Package):** The PyQt6-based presentation layer, including the MainWindow, VideoPanel, ControlPanel, StatisticsPanel, and NotificationManager.
- **Hardware Components:** The physical devices mapped to the software, specifically the high-resolution USB Camera and the USB Relay board that controls the electrical power of the UV LED lights.

2.2 Features to be Tested

The testing process will verify the functional requirements defined for the operator workflow and system behavior. The features to be tested are:

- **Real-Time Visual Feedback:** Displaying the live image frame from the camera on the interface and instantly overlaying the model's decision with a green (Healthy) or red (Aflatoxin) bounding box.
- **Scan Start/Stop & Session Management:** Initiating and pausing the scan with a single interface button , alongside the creation of a unique Batch ID, resetting live counters, and initializing a new CSV file and image folder for the session.
- **Live Counters and Status Indication:** Real-time updating of the statistics panel to accurately reflect the total number of scanned figs, the number of healthy figs, the number of aflatoxin-contaminated figs, and the contamination percentage .
- **Image Recording and Archiving:** The automatic, background saving of the UV image of each scanned fig to the local file system without causing the main user interface to freeze.
- **Report Generation and Export:** The ability to generate and export verifiable screening results into a CSV format based on date and batch, containing specific columns (Date, Time, Batch_ID, Fig_ID, Decision).
- **Hardware Status and Error Notification:** The system's ability to check camera and UV light connections at startup and display a clear warning (e.g., "Camera not found. Please check the USB connection.") rather than crashing if the hardware is disconnected

2.3 Features Not to be Tested (Out of Scope)

Certain operational constraints and design boundaries exclude specific features from this test plan:

- **Model Training and Retraining:** The system is designed to perform real-time inference using a pre-trained YOLOv11n model; therefore, the computational processes and data augmentation techniques required for training the AI model are not evaluated during runtime system testing.
- **Chemical Laboratory Analysis (HPLC):** The Figion system is a non-destructive, image-based pre-screening tool; the definitive, destructive chemical analysis (HPLC) used to establish ground truth is a separate physical procedure entirely outside the scope of this software application.

3. Testing Methodology

The Figion system employs a progressive, multi-threaded testing strategy aligned with its **Producer-Consumer** architecture. Testing is divided into specialized phases to validate modularity, concurrency, and industrial robustness.

3.1 Unit Testing

Unit testing focuses on the four core packages—**Vision, Control, Data Management, and UI**—to verify their logic in isolation before they are integrated into the full pipeline.

- **Vision Package:** Tests verify that the InferenceEngine correctly processes frames using the **YOLOv11n** model in **ONNX** format. These tests are conducted using pre-recorded datasets to measure classification accuracy without the overhead of the user interface.
- **Control Package:** Validation of the StateManager ensures the system correctly transitions through operational modes (e.g., Initializing, Ready, Scanning, Error) and prevents illegal state jumps.
- **Data Management Package:** Tests ensure that the DatabaseHandler executes **ACID-compliant** transactions on the **SQLite** database. Additionally, the SessionManager is tested to ensure it generates unique **Batch IDs** and correctly resets counters for every new session.
- **UI Package:** Individual widgets like the StatisticsPanel and NotificationManager are tested for their ability to render mock data and display error popups independently.

3.2 System and Integration Testing

This phase validates the asynchronous communication and physical hardware-software mapping defined in the system architecture.

- **Concurrency Validation:** The **Qt Signal & Slot** mechanism is tested to ensure that the **UI Thread**, **AI Worker Thread**, and **Data Worker** operate simultaneously without causing race conditions or freezing the dashboard.
- **Hardware-in-the-Loop (HiL):** Testing verifies seamless communication between the software and physical peripherals, specifically checking that the **USB Relay** board correctly triggers the **365nm UV LED** array when the "Start" command is issued.
- **End-to-End Pipeline:** A complete scan scenario is executed to verify that images captured by the **USB Camera** move through preprocessing, inference, and recording stages without data loss.
- **Traceability Reconciliation:** A final check ensures the number of physical fig scans matches the number of images saved to the file system and the total rows in the **CSV** export.

3.3 Performance Testing

Performance testing benchmarks the system against strict industrial constraints to ensure it can replace manual inspection lines.

- **Latency Benchmarking:** The system is monitored to ensure total end-to-end processing—from frame capture to UI update—does not exceed **1 second** per fig.
- **Industrial Throughput:** The system is run continuously to simulate a **2-hour shift** at a processing rate of approximately **$175~kg/hour$**.
- **Stress and Boundary Testing:** Tests simulate abnormal conditions, such as "Disk Full" or "Camera Disconnect Mid-Scan," to verify that the system transitions to a safe state (turning off UV lights) without crashing.
- **Thermal Stability:** Because the system runs on standard laptop CPUs, performance is measured under artificial high-load conditions to ensure that **thermal throttling** does not cause the system to violate the 1-second latency rule.

3.4 User Acceptance Testing (UAT)

UAT evaluates the system from the perspective of quality assurance managers and production line operators.

- **Usability Validation:** Testing confirms that a new operator can learn to navigate the dashboard and initiate scanning sessions within **30 minutes** of training.
- **Functional Goal Alignment:** Verification that the **95% Recall** (Sensitivity) for aflatoxin detection is consistently met to satisfy public health obligations.
- **Dashboard Feedback:** Operators assess the clarity of real-time indicators, ensuring the **Live Counters** for "Aflatoxin-infected" vs. "Healthy" figs update instantly and accurately.

3.5 Beta Testing

Beta testing assesses the system's robustness in realistic, non-idealized factory environments.

- **Environmental Robustness:** The system is tested for accuracy in the presence of external light leaks or false reflections from naturally shiny fig skins.
- **Motion Handling:** Testing verifies that the camera's shutter speed and the inference engine can handle figs moving on a conveyor belt without causing **motion blur** that might degrade AI accuracy.
- **Edge Computing Stability:** Deployment is tested on various standard workstation hardware to ensure the **ONNX Runtime** performs consistently across different CPU architectures.

4. Test Environment

The test environment must provide a consistent, light-isolated setting to ensure that the AI model can reliably detect the faint fluorescence of aflatoxin without external interference.

4.1 Hardware Requirements

The hardware setup bridges the gap between digital analysis and the physical production line. The following components are essential for performing hardware-in-the-loop and end-to-end integration tests.

Component	Specifications	Purpose
Workstation	Standard laptop/PC with a multi-core CPU and integrated GPU.	Runs the Figion application, processes the YOLO model on the CPU, and manages the UI.
USB Camera	High-resolution camera with programmatic exposure and white balance control.	Captures 24-bit RGB frames of figs under UV light at a resolution of at least 640x640.
UV Lighting	365nm high-intensity UV LED array.	Exposes the fluorescent properties of aflatoxin-contaminated areas on dried figs.
Control Relay	USB Relay module (e.g., CH340 chip).	Enables software-controlled power management for the UV LED system.
Dark Box	Custom-built, light-isolated prototype cabinet.	Eliminates ambient light and houses a miniature conveyor belt mechanism.
Storage	Sufficient local hard drive capacity.	Archives thousands of high-quality JPEG/PNG images and the SQLite database.

4.2 Software Requirements

To achieve the strict **1-second latency** requirement while maintaining high reliability, the software stack relies on optimized libraries for CPU-based inference and hardware-accelerated interfaces.

- **Operating System:** Standard desktop OS (Windows or Linux) capable of supporting OpenCV and Qt drivers.
- **Development Language: Python**, adhering strictly to the **PEP 8** style guide for readability and maintenance.
- **Core Libraries:**
    - **PyQt6:** Used for building the high-performance, hardware-accelerated desktop UI.
    - **OpenCV (cv2):** Handles image acquisition, camera parameter adjustment, and frame preprocessing.
    - **ONNX Runtime:** Provides the environment for optimized CPU-based model execution.
    - **PySerial / pyhid-usb-relay:** Facilitates serial communication with the relay board for light control.
- **Data Management:**
    - **SQLite3:** Local, file-based database for **ACID-compliant** persistence of metadata.
    - **CSV Standards:** Reporting modules must comply with **RFC 4180** for universal data compatibility.

4.3 Tools and Testing Frameworks

Comprehensive testing requires a suite of tools to monitor performance metrics, verify data integrity, and manage the AI training pipeline.

- **Testing Frameworks:** Python-based unit testing frameworks (such as unittest or pytest) are used for modular validation of the Vision and Data packages.
- **AI & Model Tools: * Ultralytics:** Used for managing the YOLOv11 model architecture and benchmarks.
    - **ONNX Export Tools:** Converts PyTorch models (.pt) into optimized production formats.
- **System Monitoring:**

    - **Stress-Testing Tools:** Artificially load the CPU to verify system stability against **thermal throttling**.
    - **Latency Timers:** Programmatic benchmarking to ensure the "one-second rule" is maintained from capture to UI display.
- **Data Verification:**
    - **SQLite Viewing Tools:** Independent browsers used to verify database schemas and successful record persistence.
    - **Git/GitHub:** Used for version control and tracking changes in both code and configuration baselines.
- **Validation Dataset:** A custom-built library of **10,000 unique images** (50/50 split between 'Contaminated' and 'Healthy') serves as the ground truth for all accuracy tests.

5. Roles and Responsibilities

5.1 Testing Tasks

To ensure the reliability, accuracy, and real-time performance of the Figion system, the testing protocol is divided into comprehensive phases targeting both software architecture and hardware integrations:

- **Unit Testing (Component Level):** Independent testing of the four core packages (Vision, Control, Data Management, UI). This includes verifying that the InferenceEngine correctly processes images using the YOLOv11n ONNX model and ensuring the DatabaseHandler executes CRUD operations on the SQLite database while strictly adhering to ACID principles.
- **Integration and Concurrency Testing:** Validating the asynchronous communication across the system's Producer Consumer architecture. This requires testing the Qt Signal & Slot mechanism to ensure that the UI Thread, AI Worker Thread, and ImageArchiver Worker operate simultaneously without creating bottlenecks or freezing the main interface. Hardware integration tests will also verify seamless communication between the software, the USB camera, and the relay module.
- **Performance and Latency Benchmarking:** Strict timing tests to guarantee that the total end-to-end latency from capturing a raw frame to displaying the classification result on the UI does not exceed 1 second per fig. This is a critical constraint to ensure the system matches the manual industrial processing pipeline speed of approximately 175 kg/hour.
- **AI Model Accuracy and Validation:** Rigorous evaluation of the model's classification capabilities to meet public health obligations. The testing must confirm a minimum Recall (Sensitivity) of 95% to ensure contaminated figs are strictly identified , alongside a minimum Precision of 85% to minimize false positives and prevent financial loss for exporters.
- **Environmental and Usability Testing:** Physical testing of the hardware setup inside the isolated prototype dark box under constant 365nm UV LED illumination. Furthermore, usability testing will be conducted to verify that factory operators can learn and comfortably navigate the PyQt6 based dashboard in under 30 minutes.

5.2 Personnel Assignments

To execute the testing phases efficiently, responsibilities have been distributed among the team members based on the system's subsystem decomposition:

- **İrem Ayça Uçankale – AI & Vision Subsystem Testing Lead:** Responsible for evaluating the performance of the YOLOv11n ONNX model on the CPU. This includes monitoring inference stability, calculating Recall and Precision metrics , and validating the output of the Non Maximum Suppression (NMS) and thresholding algorithms.
- **Berk Kaya – System Integration & Performance Lead:** Oversees the end-to-end testing of the layered architecture and thread management. Responsible for stress-testing the system to verify the strict sub 1 second latency requirement and ensuring flawless data flow between the AI background worker and the main UI.
- **İlhan Ün – Hardware & Control Logic Lead:** Manages the testing of physical peripherals, including the USB camera configurations and PySerial communication with the USB relay board. Responsible for validating the finite state machine (StateManager) to ensure the system handles hardware exceptions gracefully (disconnected camera).
- **Alperen Aktaş – Data Management & Traceability Lead:** Focuses on the integrity of the persistent data storage. Responsible for testing atomic SQLite database transactions via the DAO layer , verifying the functionality of the asynchronous background ImageArchiver , and ensuring all CSV exports comply strictly with RFC 4180 standards.
- **Onur Turan – User Interface (UI/UX) & Usability Lead:** Conducts testing on the PyQt6-driven Presentation Layer. Responsible for ensuring that the VideoPanel accurately renders bounding boxes in real-time , validating the live updates of the StatisticsPanel , and confirming that visual and auditory error notifications operate correctly.

6. Schedule and Resources

6.1 Test Schedule

The testing lifecycle is structured as a cumulative, 4 phase schedule to systematically identify and resolve issues:

- **Phase 1: Component & Hardware Isolation Testing:** Independent testing of software utility modules and DAO classes. Simultaneously, physical testing of the 365nm UV LED array and USB camera focus parameters inside the light isolated dark box environment.
- **Phase 2: Subsystem Integration & Concurrency Testing:** Connecting the Application Logic Layer with the Presentation and Data Layers. This phase focuses on testing the PyQT signal slot mechanisms, ensuring background workers execute without causing UI redraw issues or memory bottlenecks.
- **Phase 3: Model Verification & Latency Benchmarking:** Deploying the integrated system on standard laptop hardware (edge computing). Testing with real dried fig samples to benchmark the 1 second latency constraint and validating that the AI model consistently hits the target 95% Recall and 85% Precision thresholds.

- **Phase 4: User Acceptance Testing (UAT) & System Stability:** Conducting long-term operational tests, simulating a factory environment by running the system for 2 hours of uninterrupted scanning. This phase ensures there are no memory leaks, validates the robustness of the SQLite database during high volume data writes , and confirms the accuracy of batch based CSV reporting.

6.2 Resource Allocation

To successfully conduct the tests and deploy the prototype, the following critical resources are allocated:

- **Data Resources:** A robust, custom-built dataset consisting of 10,000 unique dried fig images captured under strictly controlled UV lighting conditions. This dataset is balanced with an target of 5,000 'Contaminated' and 5,000 'Healthy' specimens, validated through expert visual inspection as a proxy method for HPLC analysis.
- **Hardware Resources:** A standard workstation/laptop equipped with a multi core CPU and integrated GPU to run edge computing inference tasks without relying on cloud servers.
    - A high-resolution USB Camera capable of programmatic exposure adjustment.
    - A USB Relay module to control the high intensity 365nm UV LED system.
    - A custom built, light isolated dark box prototype equipped with a miniature conveyor belt mechanism.
- **Software and Tooling Resources:** The ONNX Runtime environment for optimized CPU based model execution.
    - Python based testing frameworks for unit validation.
    - SQLite database viewing tools to verify data persistence and table schemas.
    - PyQt6 libraries to render hardware accelerated desktop interfaces

7. Control Procedures

The Figion system integrates artificial intelligence, hardware control, image acquisition, and persistent data management within a single industrial workflow; therefore, testing activities must be governed by clear and well-defined control procedures.

7.1 Defect Reporting and Tracking

Defect reporting and tracking define how faults discovered during testing are identified, documented, prioritized, assigned, resolved, and verified. In the Figion project, this process is especially important because a failure may affect not only software correctness, but also

operator safety, industrial throughput, traceability, and public health. For example, a missed aflatoxin-positive fig, a frozen user interface, a corrupted database record, or a relay that leaves the UV lights on unexpectedly are all defects of different kinds, but each one can compromise system reliability in a significant way.

Whenever a defect is observed during any testing phase, a formal defect record must be created immediately. Defects may be discovered during unit testing, integration testing, hardware-in-the-loop tests, performance testing, user acceptance testing, or beta testing. The person who identifies the defect is responsible for entering it into the project's defect log or tracking system. Each defect record should contain enough detail for another team member to understand, reproduce, and investigate the issue without ambiguity. At minimum, the defect entry should include the following information:

- A unique defect identifier
- Title and short summary of the problem
- Date and time of discovery
- Reporter name
- Test environment in which the issue occurred
- Related subsystem or component
- Steps to reproduce the problem
- Expected result
- Actual result
- Severity and priority level
- Attachments such as screenshots, log files, stack traces, sample images, or database excerpts
- Current status of the defect
- Assigned team member responsible for investigation and correction

To make triage more consistent, defects should be classified according to severity. Severity reflects the impact of the defect on system operation, safety, and project objectives. In the context of Figion, the following severity structure is appropriate:

Critical Defect:

A failure that makes the system unsafe, unusable, or fundamentally unreliable. Examples include failure to detect contaminated figs at an unacceptable rate, corruption of session records, database inconsistency, inability to start or stop scanning safely, or a hardware control failure that leaves the UV LEDs active when they should be off.

Major Defect:

A serious failure that does not completely stop the system, but significantly degrades essential functionality. Examples include repeated UI freezing during scanning, lost image files during archiving, incorrect live counters, export failures, or substantial latency violations.

Minor Defect:

A defect that does not affect the core validity of scanning results but still reduces usability, consistency, or professional quality. Examples include formatting issues in CSV export, small visual errors in the interface, delayed but still correct counter refresh, or non-critical wording mistakes in notifications.

Trivial Defect:

A cosmetic or documentation-related issue with negligible operational impact, such as alignment issues, inconsistent capitalization, or non-functional wording inconsistencies in the interface.

In addition to severity, each defect should also be assigned a priority. Priority determines how urgently the team should address the issue. A defect may be severe but rare, or relatively minor but frequent enough to disrupt testing. Therefore, severity and priority should be evaluated together during defect triage meetings.

Once reported, each defect moves through a defined lifecycle. A practical defect status flow for the Figion project is as follows:

New:

The defect has been reported but not yet reviewed.

Under Review:

The team has examined the issue and confirmed that it is valid.

Assigned:

The defect has been assigned to a responsible developer or subsystem owner.

In Progress:

Investigation or correction is currently being performed.

Resolved:

A fix has been implemented, but not yet independently verified.

Retest Pending:

The corrected functionality is waiting for validation by the tester.

Closed:

The fix has been verified successfully, and the defect is considered completed.

Rejected / Not a Bug:

The reported behavior is expected, already documented, or caused by incorrect test setup rather than a real system defect.

Deferred:

The issue is acknowledged, but its correction is postponed to a later milestone because of limited time, low priority, or low operational impact.

For high-risk defects, especially those affecting AI accuracy, data integrity, hardware safety, or real-time performance, retesting must include regression testing. This means that once a fix is applied, the team should not only verify the specific corrected behavior but also check whether related functionality has been unintentionally broken. For example, a fix to improve image saving concurrency should be followed by validation of UI responsiveness, database consistency, and latency measurements, since these subsystems interact closely.

Defect trend analysis should also be part of the reporting process. At regular intervals, such as weekly project meetings, the team should review the number and type of open defects, identify recurring failure categories, and assess whether the system is improving in stability. Metrics that can support this review include:

- Number of open defects by severity
- Number of defects discovered per subsystem
- Average time to resolve a defect
- Reopened defect count
- Number of regression failures after fixes
- Distribution of defects across test phases

These metrics help reveal structural weaknesses. For example, a repeated concentration of defects in data archiving or camera initialization may indicate that the subsystem architecture or interface contracts need improvement rather than isolated bug fixing.

Since the Figion system operates in a food-safety-related context, the reporting procedure must also emphasize traceability. Every defect that affects classification outcomes, data consistency,

or session records should be traceable to a specific software version, hardware configuration, dataset version, and test case identifier. This ensures that test evidence remains auditable and that important failures can be analyzed retrospectively if needed.

7.2 Change Management

Change management defines how modifications to the Figion system are proposed, evaluated, approved, implemented, documented, and revalidated. In a project like Figion, changes may originate from many different sources: failed test cases, newly discovered hardware limitations, improvements to the AI pipeline, user feedback from operators, performance bottlenecks, or updated project priorities. Without a structured change management process, even well-intentioned improvements could create inconsistencies, reduce reliability, or invalidate previous testing results.

A change in the Figion context may involve one or more of the following:

- Modification of AI model thresholds or inference parameters
- Replacement or update of the YOLO model file
- Changes in camera resolution, exposure, or preprocessing rules
- Updates to the relay communication logic
- Revisions to database schema or export format
- UI layout or workflow changes
- Error-handling improvements
- Performance optimizations affecting thread behavior
- Changes in session management, naming conventions, or storage structure

Every proposed change should first be recorded as a change request. A change request should clearly define what is being changed, why the change is necessary, which subsystems are affected, and what risks are associated with the modification. At minimum, the change request should include:

- Unique change request ID
- Description of the requested change
- Reason and justification
- Source of the request
- Affected subsystem(s)
- Expected benefit
- Potential risks and side effects
- Estimated implementation effort
- Testing impact

- Approval status

Once created, the change request should be reviewed by the team. Because Figion is built as a layered and modular system, changes should be evaluated not only from the perspective of the directly affected package but also in terms of cross-package effects. For example, changing the confidence threshold in the Vision package might improve Recall but may also reduce Precision and alter the contamination ratio displayed in the UI. Similarly, modifying image-saving behavior may affect latency, storage usage, and session traceability. Therefore, change review should explicitly consider the following questions:

- Does the change affect public health obligations such as Recall performance?
- Does it introduce risks to data integrity or auditability?
- Does it alter the system's real-time latency target?
- Does it impact operator workflow or usability?
- Does it require updates to test cases, documentation, or reports?
- Does it change any interface contract between packages?

If the proposed change is accepted, it should be implemented in a controlled way. First, the relevant design and requirement documents must be updated if the change affects documented system behavior. This is important because the Figion project already has Analysis, High-Level Design, Low-Level Design, and Test Plan documents, and consistency between these artifacts must be preserved. Then, the implementation should be performed in a separate development branch or controlled versioned environment so that unstable changes do not directly affect the main tested version of the system.

After implementation, the changed component must undergo targeted retesting. However, in Figion, targeted retesting alone is not sufficient for many changes because of subsystem interdependence. A model update, for example, may require:

- Unit testing of the updated inference behavior
- Accuracy testing on the validation dataset
- Performance testing for CPU inference time
- Integration testing with UI overlays and counters
- Data logging validation
- End-to-end scanning tests under realistic conditions

For this reason, each approved change should be accompanied by an impact-based test selection. Low-risk cosmetic changes may require only limited verification, while any modification affecting model behavior, hardware communication, concurrency, or persistence should trigger broader regression testing.

Version control is a central part of change management. Every approved change should be linked to a specific software version or commit identifier. Test results must also reference the version under test. This ensures that the team can determine exactly which version passed or failed a given test scenario. If a later modification causes instability, the team must be able to trace the regression back to a defined change. This is especially important for industrial systems, where reproducibility and accountability are major quality concerns.

Configuration management should also be treated as part of change management. The Figion system depends not only on source code but also on configuration files, threshold values, ONNX model versions, database schemas, export templates, and possibly camera settings. A change in any of these items can alter system behavior. Therefore, the team should maintain a versioned configuration baseline for each test cycle. This baseline should identify:

- Software version
- Model version
- Confidence and IoU thresholds
- Camera parameters
- Database schema version
- Test dataset version
- Hardware setup notes

No major change should be considered complete until all associated documentation, configuration records, and test evidence have been updated. In other words, change closure requires both technical completion and documentation completion.

Finally, emergency changes should be handled with special caution. If a critical issue is discovered late in testing, such as a defect affecting contamination detection or safe shutdown behavior, a fast correction may be necessary. Even in such cases, the team should not bypass documentation and retesting entirely. At minimum, the emergency fix must be logged, its rationale must be documented, and a focused regression test must confirm that the system remains stable.

Through this process, change management ensures that the Figion project evolves in a controlled and verifiable way. Rather than treating modifications as isolated coding activities, it frames them as engineering decisions that must preserve the system's core goals: public-health-oriented accuracy, industrial speed, data traceability, and robust operation under real production conditions.

8. Risks

Testing the Figion system involves more than checking whether individual functions work correctly. Because the project combines artificial intelligence, hardware control, real-time computer vision, data persistence, and operator interaction in an industrial context, the testing phase itself carries significant risks. Some risks may reduce the effectiveness of the tests, while others may threaten the validity of the conclusions drawn from them. Therefore, risk management is an essential part of the test plan. In the Figion project, risks must be identified not only in software terms, but also in relation to hardware dependencies, environmental conditions, data quality, safety expectations, and time limitations.

8.1 Associated Testing Risks

One of the most important testing risks is the risk of insufficient or unrepresentative test data. Since aflatoxin detection under UV illumination is a specialized application domain, there is no large, standardized, publicly available dataset that fully represents the project's operating conditions. If the testing dataset is too small, too clean, too balanced in an unrealistic way, or collected under conditions different from the real scanning box, the measured Recall and Precision values may look strong in testing but fail to generalize in practice. This risk is especially serious because the project's main public health objective depends on detecting contaminated figs reliably.

A second major risk is the risk of label uncertainty in the evaluation dataset. The project documentation already notes that HPLC-based confirmation is destructive and difficult to apply to every sample, which means that many labels may rely on expert visual interpretation under UV light. If these labels are inconsistent or partially subjective, then even a technically correct model may appear inaccurate, or a flawed model may appear better than it really is. As a result, testing outcomes may not fully reflect the true contamination status of the figs.

Another important risk is the risk of hardware instability during test execution. The Figion system depends on a USB camera, UV LED illumination, a relay-controlled lighting mechanism, and a dark-box environment. Hardware faults such as intermittent USB disconnections, unstable power supply, relay communication errors, LED intensity degradation, or camera driver issues may interrupt testing or generate misleading results. In such cases, it may become difficult to determine whether a failure is caused by the software logic or by the test setup itself.

There is also a risk of performance variability across test environments. Since the project is explicitly designed to run on standard laptop CPUs or integrated GPUs, even small differences in hardware specifications, background processes, thermal conditions, or operating system

configuration may affect inference time and overall latency. A system that satisfies the one-second requirement on one machine may violate it on another, especially during long-duration or stress tests. This creates uncertainty in performance validation unless the test environment is tightly controlled and documented.

A related concern is the risk of thermal throttling and long-run degradation. The project aims to operate for extended periods, such as two hours of uninterrupted scanning. During continuous use, CPU temperature may increase and trigger thermal throttling, which can slow down inference and increase latency. This is a critical testing risk because short tests may pass while realistic shift-length operation fails.

Another serious risk is the risk of concurrency-related defects remaining hidden during limited testing. The system architecture uses multiple threads and asynchronous communication patterns to keep the interface responsive while image processing, database writing, and file archiving happen in the background. Concurrency bugs such as race conditions, queue overflows, missed signals, deadlocks, or timing-dependent data mismatches may not appear consistently. These defects are often difficult to reproduce and may only emerge under high load, prolonged execution, or unusual timing conditions. If the test campaign is too narrow, such issues may survive into deployment.

The Figion test process also faces a risk of incomplete integration coverage. Individual subsystems may pass unit tests in isolation while still failing when combined. For example, the Vision package may classify frames correctly, the Data Management package may save records correctly, and the UI may display counters correctly, but the end-to-end pipeline could still produce mismatched image counts, missing session records, or delayed overlays. Because the project is a tightly coordinated hybrid system, subsystem-level success does not guarantee full operational success.

There is additionally a risk of false confidence from laboratory-like testing conditions. Tests performed in calm, well-controlled, artificial environments may not capture real factory disturbances such as vibration, ambient light leakage, operator interruptions, conveyor speed variation, or reflective fruit surfaces. If testing is too idealized, the system may appear more robust than it truly is.

A very important domain-specific risk is the risk of underestimating false negatives. In many software systems, occasional misclassification may be acceptable, but in Figion, a false negative means that a contaminated fig may continue down the production line. This is directly tied to food safety and public health. Therefore, any weakness in the testing design that hides false negatives, such as class imbalance, insufficient contaminated samples, or overly forgiving evaluation procedures, represents a high-severity risk.

From the data and traceability perspective, there is a risk of silent persistence failures. Because image files are saved asynchronously and metadata is written into SQLite, the system could theoretically continue scanning while silently dropping images, duplicating records, or creating mismatches between stored files and exported CSV entries. If these failures are not explicitly tested under load, the traceability promise of the system may be undermined without immediate visual evidence.

Another testing risk is the risk of schedule pressure reducing test depth. The project timeline is limited, and the same team is responsible for design, implementation, integration, and testing. Under time pressure, the team may naturally focus on visible or demonstrable functionality while postponing stress tests, negative tests, or long-duration reliability checks. This could leave important edge cases insufficiently examined.

Human factors also introduce risk. There is a risk of usability issues being underestimated if testing focuses too heavily on technical performance. Even if the model is accurate and the system is stable, confusing notifications, unclear status indicators, delayed counters, or ambiguous error messages may reduce operator trust and lead to misuse. Since the Figion system is intended for line operators and quality personnel, usability failures are operational risks as well as interface issues.

Finally, there is a risk of change-related regressions during late project stages. If the team makes last-minute adjustments to improve Recall, UI behavior, or latency, these modifications may unintentionally break previously validated functions. In a project with interdependent packages, even a small change in thresholding, session logic, or hardware control may produce new failures elsewhere.

8.2 Mitigation Strategies

To reduce the risk of insufficient or unrepresentative test data, the team should build the evaluation dataset as carefully as possible around the actual operating scenario of the Figion system. Test images should be collected under the same or very similar UV wavelength, camera placement, dark-box geometry, exposure settings, and conveyor conditions used by the prototype. Rather than relying only on a single clean dataset, the team should include variation in fig appearance, contamination visibility, brightness, blur level, and reflective surface properties. Where possible, the dataset should include borderline and difficult examples, not only obvious cases.

To mitigate label uncertainty, the team should use a layered labeling strategy. Samples with laboratory confirmation through HPLC or another definitive method should be treated as the highest-confidence evaluation cases whenever available. For visually labeled data, clear

annotation criteria should be documented and applied consistently. If feasible, multiple reviewers should examine ambiguous samples, and disagreements should be resolved through explicit review rather than informal judgment. This will not eliminate all uncertainty, but it will make the testing basis more transparent and defensible.

Hardware instability risks can be reduced by introducing a hardware readiness checklist before each major test cycle. This checklist should include camera connection verification, relay response confirmation, UV light functionality, cable inspection, power stability checks, and sufficient storage availability. Test logs should also record the hardware configuration used in each session, so that failures can later be interpreted correctly. If practical, spare cables, adapters, or replacement peripheral units should be kept available to reduce downtime and distinguish software faults from hardware faults more quickly.

To address performance variability, the team should define a reference test environment and document it in detail. Key variables such as processor model, RAM, operating system, background applications, thermal state, and software versions should be controlled as much as possible. Performance benchmarks should be repeated multiple times rather than measured only once. Reporting should include not just average latency, but also worst-case latency and variance, since real industrial usefulness depends on consistency as much as on average speed.

Thermal throttling risk should be mitigated through long-duration performance tests that reflect realistic workloads rather than short bursts. The team should deliberately observe inference time over extended sessions and note whether performance drifts upward as temperature rises. If such behavior is observed, mitigation may include better cooling, lower processing overhead, reduced image resolution where acceptable, or software-level throttling strategies. Most importantly, the test report should clearly distinguish between short-run and sustained performance.

Concurrency-related risks require targeted stress and isolation tests. The team should intentionally create situations in which multiple operations happen simultaneously: rapid frame capture, intensive inference, frequent database writes, and continuous image archiving. These tests should monitor symptoms such as frozen UI elements, lost records, queue growth, delayed signals, or mismatched counters. Logging should be designed to help identify timing-related problems. Because concurrency defects may be intermittent, suspicious scenarios should be repeated multiple times rather than assumed resolved after a single successful run.

To prevent incomplete integration coverage, the test strategy should emphasize end-to-end scenarios in addition to unit and subsystem testing. In particular, the system should be tested from the operator’s perspective: start a session, activate hardware, capture frames, run inference, save images, write results, update counters, export reports, and close the session. Outputs from each stage should be reconciled. For example, the number of processed figs

shown in the UI, the number of saved images, the number of database records, and the number of CSV rows should be checked for exact agreement.

The risk of overly idealized testing can be reduced by deliberately introducing realistic disturbances. The team should test under mild ambient light leakage, different fig surface characteristics, faster belt conditions, empty-belt periods, and temporary hardware interruptions. Even if the prototype environment cannot fully replicate a factory, testing should still move beyond perfect-laboratory assumptions.

To address the false-negative risk specifically, evaluation procedures should prioritize Recall analysis on contaminated examples. Instead of reporting only overall accuracy, the team should inspect confusion matrices, review missed contaminated cases individually, and analyze failure patterns. If the system misses faint fluorescence or certain contamination shapes more often, this should be explicitly documented and fed back into threshold tuning or dataset improvement. Since false negatives are ethically more severe than false positives in this project, test interpretation must reflect that priority.

Silent persistence failures should be mitigated by reconciliation checks and audit tests. After each significant test batch, the team should verify that the number of saved images matches the number of database records and exported report lines. File paths stored in the database should be sampled or automatically validated against the file system. Power interruption and disk-full simulations should be included to ensure that data integrity remains acceptable under abnormal conditions.

Schedule pressure can be mitigated by prioritizing risk-based testing rather than treating all tests as equally important. The team should identify a core test subset that absolutely must be completed before project closure: AI Recall validation, end-to-end traceability checks, one-second latency validation, hardware failure handling, and long-duration stability testing. Lower-impact cosmetic or convenience issues can be deferred if necessary, but these core risks should never be left untested.

Usability-related risks can be reduced by involving representative users, even informally, in acceptance-style evaluations. Operators or classmates unfamiliar with the internals of the system can be asked to perform simple tasks such as starting a session, interpreting counters, responding to an error popup, and exporting a report. Their confusion or hesitation can reveal interface problems that purely technical testing may miss.

Finally, change-related regression risk should be controlled through disciplined change management and regression testing. Every significant modification, especially in model thresholds, control logic, threading, or persistence, should trigger a defined set of retests. No performance improvement or quick bug fix should be accepted solely on the basis of local

success. Instead, the team should confirm that the broader system goals—accuracy, responsiveness, safety, and traceability—still hold after the change.

By applying these mitigation strategies, the Figion project can make its testing process more realistic, reliable, and aligned with the system's real-world responsibilities. In this way, risk management becomes not just a defensive activity, but a structured mechanism for improving the credibility and engineering quality of the final system.

9. Test Cases

Integrating AI into industrial vision systems, especially when it comes to food safety, means we need a rock-solid testing plan. For our Figion project, we are building a system that detects aflatoxin in dried figs by looking for specific fluorescence under UV light. Since this is an edge computing setup, it has to run smoothly on standard hardware right on the factory floor, not just in a perfect laboratory.

Because we are dealing with public health, we have a very strict goal: our system needs a 95% Recall rate (meaning we must successfully catch at least 95% of the contaminated figs). At the same time, we need an 85% Precision rate so we don't accidentally throw away perfectly healthy figs and cause financial loss for the producers. On top of all this, the system has to be fast. To keep up with the manual production line speed of 175 kg per hour, our software must process each fig in under 1 second.

The test cases below are designed to check everything from basic unit operations (like making sure the camera connects) to full hardware-in-the-loop stress tests.

9.1 Vision Subsystem Tests

The Vision package is the brain of our system. It takes raw camera frames, prepares them, and runs our YOLOv11n model using the CPU. We specifically chose the YOLOv11n architecture because it runs much faster on a CPU compared to older versions, which is exactly what we need to hit our 1-second time limit.

- **TC-VIS-001 (Camera Initialization):** We will test if the system can successfully connect to the USB camera and lock the auto-exposure and white balance. This is crucial because we need consistent settings to capture the UV glow in the dark room.
- **TC-VIS-002 (Frame Capture & Preprocessing):** We need to make sure the system grabs a full 24-bit RGB frame without dropping any data, and resizes it to 640x640 pixels in under 15 milliseconds.
- **TC-VIS-003 (YOLOv11n Inference):** We will feed a known test image to the AI model

to see if it correctly draws bounding boxes around the fluorescent spots.

- **TC-VIS-004 (Post-processing & NMS):** We test the Non-Maximum Suppression (NMS) logic. If the AI draws multiple overlapping boxes over a single spot of aflatoxin, this test ensures the system merges them into one clean detection.
- **TC-VIS-005 (Resource Release):** When the operator stops the scan, the system must safely release the camera and clear the RAM to prevent memory leaks during long shifts.

9.2 Control & Hardware Interface Tests

This part of the code manages our physical hardware, especially the USB relay that turns the high-intensity UV LEDs on and off. Since UV light can be harmful to human eyes, the system must strictly control when these lights are active.

- **TC-CTRL-001 (Relay Connection):** We test the serial connection to the USB relay. It should establish a handshake without crashing the app if the connection fails.
- **TC-CTRL-002 (UV Light Trigger):** We will physically verify that sending an 'ON' command from the software actually turns the lights on, and 'OFF' turns them off.
- **TC-CTRL-003 (State Machine Security):** We will try to force the system into an illegal state (like jumping from 'Scanning' directly to 'Error' without a real trigger). The system's state manager must block this.
- **TC-CTRL-004 (Missing Hardware at Startup):** What happens if someone forgets to plug the camera in? The system should detect this at startup, disable the 'Start' button, and show a clear warning message rather than crashing to the desktop.

9.3 Data Management & Traceability Tests

Figion isn't just a simple detector; it's a traceability hub. Factory managers need a record of everything that happens.

- **TC-DAT-001 (Session Management):** Starting a new scan should automatically create a unique Batch ID (like BATCH_20251126_001) and reset all the counters to zero.
- **TC-DAT-002 (Database Integrity):** We will simulate a sudden power loss while the system is writing a result to the SQLite database. Thanks to ACID compliance, the database must not corrupt.
- **TC-DAT-003 (Asynchronous Image Saving):** Saving large images takes time. We will push 50 images rapidly into the system to verify that the background worker saves them to the hard drive without making the live video feed freeze.
- **TC-DAT-004 (CSV Export):** We will generate an export report for a batch with thousands of records. We'll check the CSV file to ensure there are no missing labels, corrupted text, or misaligned columns.

9.4 User Interface (UI) Tests

Our UI is built with PyQt6. It needs to be super responsive because if it lags, the operator won't

trust the system.

- **TC-UI-001 (Live Video Rendering):** We test if the UI can receive processed frames and draw the green (Healthy) or red (Aflatoxin) bounding boxes smoothly at 60 FPS (processing under 16ms per frame).
- **TC-UI-002 (Live Counters):** As the AI makes decisions, the total, healthy, and contaminated counters (and the percentage) should update instantly on the screen.
- **TC-UI-003 (Session Log Scrolling):** We will flood the system with rapid detection results to ensure the scrolling table widget doesn't consume too much RAM or freeze the app over time.
- **TC-UI-004 (Error Popups):** We will trigger a simulated hardware error to verify that a clear warning window pops up and safely disables the scanning controls.

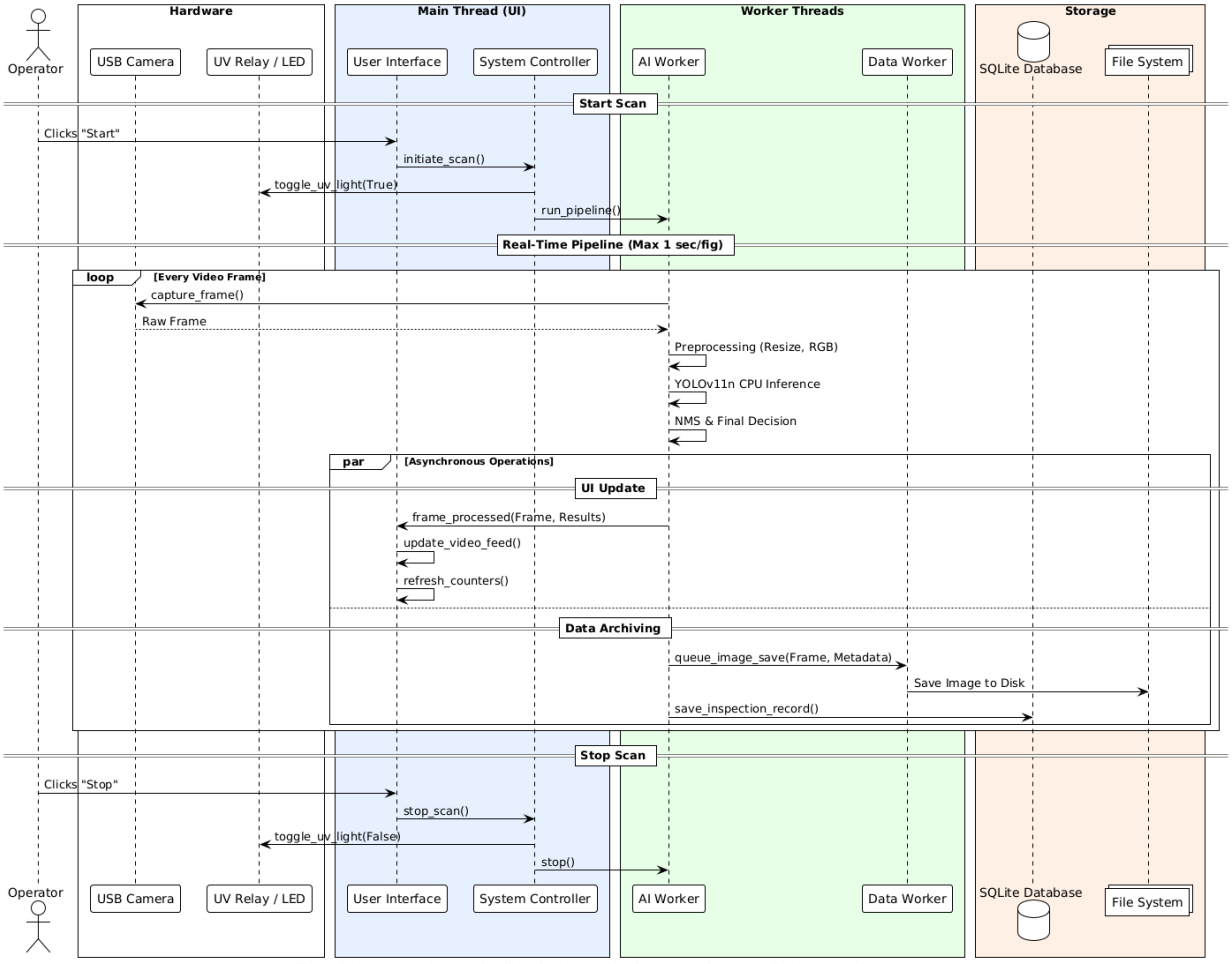

Figure 1: Sequence Diagram detailing the interactions between subsystems during a scanning session.

9.5 System Integration (End-to-End) Tests

This is where we test how well all the separate packages talk to each other.

- **TC-INT-001 (Start Scan Flow):** The operator clicks 'Start'. We verify that the Batch ID

is created, the UV lights turn on, and the video feed appears on the screen all within 1 second.
- **TC-INT-002 (Thread Isolation):** We will simulate a heavy AI workload that maxes out the CPU. We then check if the UI buttons are still clickable and responsive. The AI thread must not block the UI thread.
- **TC-INT-003 (Complete Pipeline Check):** We will run 500 figs through the system. At the end, we must find exactly 500 images saved in the folder and exactly 500 matching rows in the database.

9.6 Performance & Latency Tests

Since we are running this on a standard CPU to keep costs low, performance is our biggest technical hurdle.

- **TC-PERF-001 (The One-Second Rule):** We will measure the exact time from when the camera takes the picture to when the result is logged and displayed on the screen. It absolutely must be under 1000 milliseconds.
- **TC-PERF-002 (Throughput Test):** We will run the system continuously to simulate a 2-hour shift at 175 kg/hour. The system must process this volume without dropping frames unnecessarily.
- **TC-PERF-003 (Thermal Throttling Resistance):** Factory floors get hot. We will artificially max out the computer's CPU with a stress-testing tool to make it heat up and slow down. We'll then verify that our inference time still stays within acceptable limits.

9.7 AI Accuracy Tests

Hardware speed means nothing if the AI makes the wrong call.

- **TC-AI-001 (Maximum Recall - Public Health):** We will run 5,000 images of confirmed aflatoxin-contaminated figs through the model. It must correctly flag at least 4750 of them (Recall $\geq$ 95%).
- **TC-AI-002 (Precision - Commercial Health):** We will feed 4,000 perfectly healthy fig images into the system. The false-positive rate must be low enough to maintain a Precision score of at least 85%.
- **TC-AI-003 (Faint Fluorescence):** We will test the model with very faint, blurry glowing spots to ensure it catches early-stage aflatoxin, not just the bright, obvious spots.

9.8 Environmental & Hardware-in-the-Loop Tests

The real world is messy, and our physical testing box isn't perfect.

- **TC-ENV-001 (Ambient Light Leaks):** We will shine bright lights outside the dark test box to ensure our mechanical curtains actually block the light and don't cause false positive detections.
- **TC-ENV-002 (Motion Blur):** We will run the conveyor belt at max speed and check if

the camera's shutter speed is fast enough to freeze the motion without blurring the fig.

- **TC-ENV-003 (False Reflections):** We will run naturally shiny (but healthy) figs under the camera to ensure the AI doesn't confuse natural skin reflection with toxic fluorescence.

9.9 Exception & Boundary Conditions Tests

What happens when things go completely wrong on the factory floor?

- **TC-EXC-001 (Camera Disconnect Mid-Scan):** We will physically yank the USB cable out while the conveyor is running. The UI must instantly show an error.
- **TC-EXC-002 (Disk Full):** We will fill up the computer's hard drive completely. The system should gracefully stop saving images, but keep logging text data to the SQLite database and warn the user.
- **TC-EXC-003 (Empty Conveyor Belt):** We will leave the system running with no figs on the belt. It should process the empty black frames without accidentally hallucinating and detecting ghost objects.

10. References

1. Kuru incirlerde AFLATOKSİN ve okratoksin a bulaşisinin önlenmesi. (n.d.). https://www.tarimorman.gov.tr/GKGM/Belgeler/Uretici_Bilgi_Kosesi/Egitim/Hijyen_Kilavuz/kuru_incir_aflatoksin_okratoksin_a_onleme_azaltma.pdf

2. Ömer Barış Özlüoymak. (2014). Development of an UV-Based Imaging System for Real-Time Aflatoxin Contaminated Dried Fig Detection and Separation. Tarım Bilimleri Dergisi/Ankara Üniversitesi Ziraat Fakültesi Tarım Bilimleri Dergisi, 20(3), 302–302. https://doi.org/10.15832/tbd.87873

3. Kılıç, C., Özer, H., & İnner, B. (2024). Real-time detection of aflatoxin-contaminated dried figs using lights of different wavelengths by feature extraction with deep learning. Food Control, 156, 110150. https://doi.org/10.1016/j.foodcont.2023.11015

4. Ultralytics. (n.d.). YOLO11 vs. YOLOv8: Performance Comparison and Benchmarks. Ultralytics Documentation. https://docs.ultralytics.com/compare/yolo11-vs-yolov8/