TED UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

Low-Level Design Report

CMPE 492 – Senior Design Project II

by

Berk Kaya

İlhan Ün

İrem Ayça Uçankale

Alperen Aktaş

Onur Turan

1. Introduction

This document details the object-level design and interface specifications for the Figion system. The system aims to automate the detection of aflatoxin in dried figs using computer vision and edge-computing infrastructure.

1.1 Object Design Trade-offs

During the object design phase, various trade-offs were made to meet the Figion project's real-time performance and edge computing requirements. The key engineering decisions made while bridging the gap between application objects and off-the-shelf components are as follows:

Artificial Intelligence Inference Engine (Maximum Accuracy vs. Speed and Hardware Constraints): The system must run on a standard computer processor (CPU) and produce results with a maximum latency of 1 second per object. To meet this time constraint, the YOLOv11n (Nano) model—which runs 20–30% faster on a CPU compared to previous generations—was chosen over heavy models with a high number of parameters. Additionally, by switching from the PyTorch format to the ONNX format, an inference engine optimized for the production environment was utilized. With this decision, the marginal increase in accuracy that a larger model could provide was sacrificed to achieve an industrial processing speed of 175 kg/hour.

User Interface Infrastructure (High Performance and Fluidity vs. Development Speed): Instead of web-based Python frameworks like Streamlit, which offer rapid prototyping capabilities, the Qt framework was chosen for its direct native hardware acceleration. While web-based solutions cause delays in live video streaming and redraw issues in the interface, using Qt enables the creation of high-performance desktop interfaces. Here, the ease of rapid development was sacrificed to prioritize the operator's ability to monitor the system in real-time and without interruption.

Concurrency and Thread Management (Architectural Complexity vs. UI Responsiveness): A multi-threaded architecture was designed using the Producer-Consumer pattern to ensure that image analysis and UI updates run simultaneously without issues. While the user interface is maintained on the main thread (UI Thread), the AI-based analysis of camera images is offloaded to a separate background thread (AI Worker Thread). Although this asynchronous architecture (Signal & Slot) increases code complexity and synchronization effort in the system, it is a necessary trade-off to prevent computationally intensive operations from blocking the live video stream.

Persistent Data Management (I/O Speed vs. Single-Database Integrity): Storing the high-resolution version of each fig photo directly in the database along with the analysis results

could cause I/O bottlenecks and significant slowdowns in the system. To mitigate this risk, visual files are stored directly in the file system using a hierarchical folder structure, while the SQLite database managed via the DatabaseHandler component stores only the file paths and metadata (score, decision, timestamp) of these images. Using file-based SQLite instead of setting up an external, heavy-duty database server supports the system's serverless architecture while ensuring reliability in accordance with ACID standards.

1.2 Interface Documentation Guidelines

The Figion system is architected as a modular, layered desktop application designed for the real-time detection of aflatoxin in dried figs. The software decomposition ensures that vision processing, hardware control, and data management can operate independently and efficiently on edge hardware.

1.2.1 Vision Subsystem Services

This subsystem is the core analytical component, handling the transformation of raw camera frames into classification results.

Service Name	Parameters	Description
initialize_camera	camera_id	Loads the camera driver and sets parameters like exposure and size.
capture_frame	None	Retrieves a single snapshot (raw frame) from the USB camera.
run_inference	frame	Processes the frame through the YOLOv11n model and returns a list of detected objects with class labels and confidence scores.
release_resources	None	Safely disconnects the camera and releases system resources.

1.2.2. Control Subsystem Services

The Control Subsystem manages the physical hardware status and the internal state of the application.

connect_relay(port): Establishes a serial connection to the USB relay board that controls the UV LED array
toggle_uv_light(state: bool): Activates (True) or deactivates (False) the high-intensity UV illumination
get_hardware_status(): Performs a "Health Check" to verify if the camera and relay are properly connected

1.2.3 Data Management Services

This subsystem provides an abstraction for the **SQLite database** and local file system to ensure traceability.

start_new_session(user_id): Creates a unique **Batch ID** record and initializes session-based counters.
save_inspection_record(data_object): Performs an atomic transaction to save fig metadata (ID, Result, Confidence) to the database.
queue_image_save(image, metadata): Offloads slow I/O operations to a separate worker thread to prevent UI freezing.
export_data(session_id, format='csv'): Generates a verifiable audit trail report in CSV format.

1.2.4 Interface(UI) Services

These services manage the presentation layer, ensuring real-time feedback for the operator.

update_video_feed(qimage): Renders the processed image with green (Healthy) or red (Aflatoxin) bounding boxes on the dashboard.
refresh_counters(stats_dict): Updates the live statistics panel showing total processed figs and contamination ratios.
display_error(message): Alerts the operator to critical issues, such as "Camera not found" or "Disk full".

1.3 Engineering Standards

The development of the Figion system adheres to globally recognized engineering standards to ensure technical consistency, maintainability, and professional integrity. These standards guide the modeling, coding, data management, and ethical considerations of the project:

1.3.1 Software Modeling and Documentation Standards

- **Unified Modeling Language (UML):** All architectural diagrams, including package relationships , class interfaces , and dynamic behavior (sequence and activity diagrams) , are developed using UML standards to ensure a universal technical language.
- **IEEE 1016-2009 (Standard for Software Design Descriptions):** The organization and content of this Low-Level Design (LLD) report follow the principles of IEEE 1016, ensuring that all solution objects, interfaces, and design trade-offs are clearly and completely documented.

1.3.2 Programming and Implementation Standards

- **PEP 8 – Style Guide for Python Code:** Since the core logic is implemented in Python , the team follows PEP 8 to maintain high code readability and consistency across the Vision and Control packages.
- **Clean Code Principles:** Object design follows "Single Responsibility" and "Modular Decomposition" principles to ensure that components like the InferenceEngine and DatabaseHandler can be updated independently without affecting the whole system.

1.3.3 Data and Reliability Standards

- **ACID (Atomicity, Consistency, Isolation, Durability):** To ensure data integrity in the "Traceability Hub," the SQLite database operations are designed to meet ACID standards, preventing data loss or corruption during power failures or unexpected shutdowns.
- **RFC 4180 (Common Format for CSV Files):** The ReportGenerator service exports inspection results (Batch ID, Result, Timestamp) in compliance with the RFC 4180 standard to ensure compatibility with third-party data analysis tools.

1.3.4 Performance and Quality Metrics

- **ISO/IEC 25010 (System and Software Quality Models):** The system's success is measured through specific performance sub-characteristics:
  - **Time-behavior (Latency):** Total processing time must not exceed 1 second per fig to match the $175~kg/hour$ manual processing rate.
  - **Functional Correctness:** The AI model is strictly evaluated using Recall (min. 95%) and Precision (min. 85%) metrics to meet public health and commercial reliability obligations.

1.3.5 Ethical and Professional Standards

- **ACM Code of Ethics and Professional Conduct:** The team adheres to the ACM code by prioritizing public health through high-accuracy aflatoxin detection and ensuring data transparency regarding the "proxy method" used for dataset labeling.

1.4 Definitions, acronyms, and abbreviations

Aflatoxin: A family of toxic and carcinogenic compounds produced by certain fungi found on agricultural crops such as dried figs. In this system, it refers to the target contamination that exhibits fluorescence under UV light.

UV (Ultraviolet): Electromagnetic radiation with a wavelength shorter than that of visible light. The system utilizes a 365nm high-intensity UV LED array in a dark environment to expose fluorescent contamination on the figs.

ACID (Atomicity, Consistency, Isolation, Durability): A set of properties of database transactions intended to guarantee data validity despite errors, power failures, or other mishaps. The system's SQLite database adheres to these standards for reliable storage.

DAO (Data Access Object): A structural design pattern that isolates the application/business layer from the persistence layer (usually a database) using an abstract API, hiding raw SQL queries from the main application logic.

DTO (Data Transfer Object): An object used to encapsulate data and send it from one subsystem of an application to another (e.g., passing InspectionResult between the Vision and Data Management packages).
NMS (Non-Maximum Suppression): A post-processing technique used in object detection models to eliminate redundant, overlapping bounding boxes and keep only the single most confident prediction for a detected object.

2. Packages

The low-level design of the Figion system is organized into packages in order to preserve modularity, reduce coupling between components, and make the system easier to maintain and extend. This packaging strategy is derived from the subsystem decomposition defined in the high-level design, where the system is separated into four major subsystems: Vision, Control, Data Management, and User Interface. At the low-level design stage, these subsystems are refined into implementation-oriented packages that encapsulate related classes, interfaces, and responsibilities.

The package organization also supports the layered architecture defined in the previous design phase. The Presentation Layer is mainly represented by the UI package, the Application Logic Layer is mainly represented by the Vision and Control packages, and the Data and Infrastructure Layer is represented by the Data Management package together with low-level hardware interaction utilities. This alignment ensures that the design remains consistent across analysis, high-level design, and low-level design phases.

2.1 Vision Package

The Vision package contains all classes related to image acquisition, preprocessing, model inference, and prediction refinement. Since the main purpose of Figion is to detect aflatoxin-contaminated figs from UV images, this package forms the analytical core of the system. In the high-level design, the Vision Subsystem is already defined as the component responsible for collecting and analyzing visual data. At the low-level design stage, this responsibility is refined into smaller solution objects with clearly separated duties.

This package includes classes such as CameraManager, FrameBuffer, Preprocessor, InferenceEngine, and PostProcessor. CameraManager is responsible for opening and configuring the USB camera, adjusting properties such as exposure and gain, and retrieving raw frames. FrameBuffer or a queue-based helper object is used to temporarily hold incoming frames before they are processed. Preprocessor converts raw frames into the format expected by the trained model, including resizing, color-space conversion, and normalization. InferenceEngine encapsulates the AI model execution process, including loading the ONNX model and performing inference on the CPU. Finally, PostProcessor interprets raw model outputs by applying thresholding and non-maximum suppression, then converts them into a final decision such as Healthyor Aflatoxin.

The package is designed to be computationally efficient, because the project requires the end-to-end latency for one fig to remain below one second. For this reason, the package is isolated from UI rendering and database operations. It only focuses on producing a classification result as quickly and accurately as possible. This separation is also consistent with the sequence diagram

in the high-level design, where the vision worker captures the frame, runs YOLO inference, and returns the result independently from UI updates and data logging.

2.2 Control Package

The Control package manages the operational behavior of the system. While the Vision package decides what is seen in an image, the Control package decides when and under which system state the scanning process is allowed to run. In the high-level design, this subsystem is described as responsible for hardware management and general system status. In the low-level design, this responsibility is refined into objects that coordinate hardware, user commands, and state transitions.

This package includes classes such as SystemController, StateManager, HardwareController, and HardwareMonitor. SystemController acts as the main coordinator of the application flow. It receives high-level actions from the UI, such as starting or stopping a session, and delegates these actions to the appropriate packages. StateManager implements the finite-state behavior of the system and keeps the current mode of operation, such as Initializing, Ready, Scanning, Paused, or Error. HardwareController is responsible for sending low-level commands to peripherals such as the UV relay module. HardwareMonitor checks whether required devices such as the camera and relay are connected and operational before scanning begins.

This package is particularly important for safety and robustness. The high-level design explicitly states that invalid transitions, such as attempting to start scanning when no camera is connected, must be prevented. Similarly, the boundary conditions define behaviors for cases such as startup without a camera, excessive conveyor speed, and unexpected runtime errors. Therefore, the Control package is not only a coordination layer but also a constraint-enforcement layer that protects the system from unsafe or inconsistent operation.

2.3 Data Management Package

The Data Management package is responsible for all persistent information generated by the system. In the analysis and high-level design reports, Figion is described not only as a detector but also as a traceability hub that stores images, session information, and classification results for later review and reporting. The purpose of this package is to implement that traceability requirement in a reliable and structured way.

This package includes classes such as SessionManager, InspectionRepository, DatabaseHandler, ImageArchiver, and ReportGenerator. SessionManager creates and closes sessions, generates unique batch identifiers, and resets counters when a new session starts. DatabaseHandler encapsulates SQLite interactions and executes create, insert, update, and query operations. InspectionRepository provides a higher-level abstraction over raw database access and allows other packages to save inspection records without directly writing SQL logic. ImageArchiver stores the captured fig images in the file system using the selected directory structure and naming convention. ReportGenerator exports stored inspection data into CSV format upon user request.

This package is intentionally separated from the Vision package because image saving and database writing are I/O-bound tasks and may introduce delays if mixed with inference logic. The high-level design already indicates that image saving should run in a separate worker thread and that SQLite should be used for reliable ACID-compliant storage. Therefore, the Data Management package is designed around asynchronous persistence and integrity preservation. This helps the system satisfy requirements such as “no duplicate or missing labels,” “CSV export by batch and date,” and “automatic archiving of each scanned fig image.”

2.4 UI Package

The UI package contains all classes that the operator directly interacts with. In both the analysis and high-level design reports, the user interface is described as the place where live images, counters, session logs, and hardware status are presented to the operator. In the low-level design, this package is refined into screen-level and widget-level components that provide a responsive desktop experience.

This package includes classes such as MainWindow, DashboardView, VideoPanel, ControlPanel, StatisticsPanel, SessionLogView, and NotificationManager. MainWindow acts as the root UI container. DashboardView organizes the major interface regions. VideoPanel is responsible for presenting the live frame and overlaying the classification result. ControlPanel handles actions such as start, stop, new session, and export. StatisticsPanel displays values such as total processed figs, contaminated count, healthy count, and contamination ratio. SessionLogView lists the most recently scanned figs and their labels. NotificationManager displays warnings and errors, such as camera connection failure.

The UI package does not directly perform inference or database access. Instead, it communicates with the control and service-level objects through signals, callbacks, or controller interfaces. This is consistent with the asynchronous Qt Signal & Slot communication model mentioned in the high-level design. As a result, the UI remains responsive even while image analysis and saving operations continue in the background.

2.5 Utility and Shared Package

In addition to the four core packages, the low-level design includes a small Utility/Shared package that contains reusable supporting components. These classes are not directly part of the domain logic, but they are necessary for consistent and maintainable implementation.

Typical classes in this package include ConfigManager, Logger, ErrorHandler, PathBuilder, and common data transfer objects such as InspectionResult or FrameMetadata. ConfigManager stores configurable parameters such as model path, confidence threshold, camera ID, and export folders. Logger writes system events and runtime errors into log files. ErrorHandler standardizes exception handling and user-facing error messages. PathBuilder creates valid storage paths for images and exported reports. Shared DTO-style objects help maintain a consistent data contract between packages.

This package improves code reuse and keeps package responsibilities clean. For example, instead of each package separately constructing file paths or formatting log messages, these concerns are centralized in shared support classes.

2.6 Package Relationships

The package structure is designed with directional dependencies. The UI package depends on the Control package to initiate actions and retrieve current system state. The Control package coordinates the Vision and Data Management packages but does not embed their internal logic. The Vision package sends classification results to the Data Management package through service interfaces or result objects. The Utility package can be used by all other packages, but it does not depend on them.

This arrangement minimizes coupling and makes testing easier. For example, the Vision package can be unit-tested with prerecorded images without launching the UI, and the Data Management package can be tested independently using sample records. Similarly, if the relay hardware or the camera changes in the future, the required modifications will mostly remain inside the Control package and its hardware-facing interfaces rather than affecting the entire codebase. This preserves the maintainability goal defined in earlier reports.

2.7 Rationale for the Package Design

This package organization was selected because it directly reflects the structure already established in the previous reports while refining it into implementable units. The analysis report identifies the user-facing goals, session-based workflow, and hardware failure scenarios; the high-level design formalizes subsystem decomposition, service boundaries, and layered interaction. The low-level package design preserves that logic and turns each subsystem into a concrete implementation boundary.

As a result, the proposed package design supports the main quality goals of the project: real-time performance, modularity, data integrity, portability, and robustness under industrial operating conditions. It also prepares the system for future extensions, such as multi-camera support, additional contamination classes, alternative models, or a web-based monitoring panel, without requiring a full redesign of the codebase.

3. Class Interfaces

In this section, we will look at the main classes of our software. These classes take the abstract ideas from our analysis phase and turn them into actual code that works on the factory floor. We will define exactly what each class does and how they talk to each other.

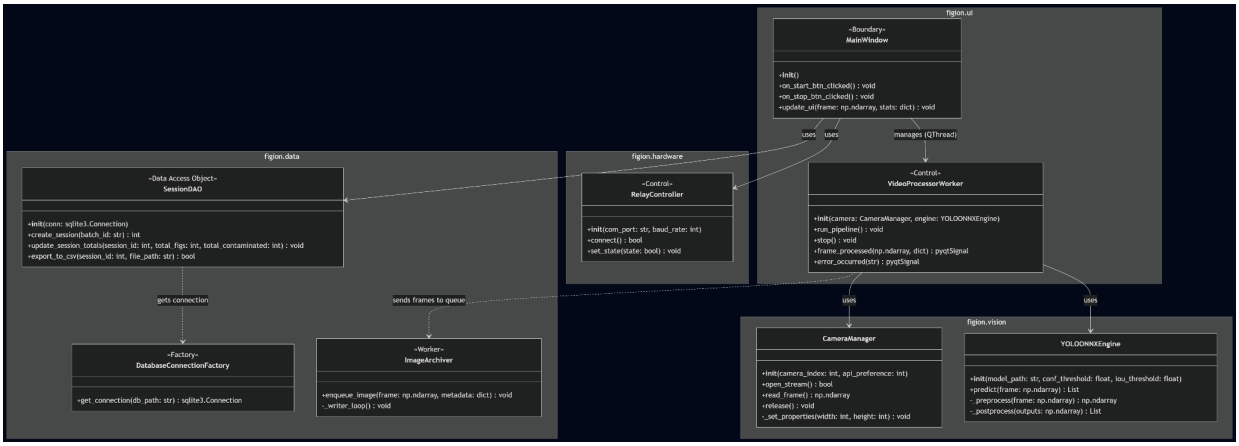

Figure 1: UML Class Diagram of the Figion System. This structural viewpoint illustrates the subsystem decomposition, narrow class interfaces, and exact dependencies across the vision, data, hardware, and user interface packages.

3.1 Figion Vision Component

This is the core of the Figion system. It grabs images from the USB camera, prepares them for the AI model, runs the ONNX model, and then translates the AI outputs into simple results we can use.

3.1.1 CameraManager

The CameraManager is a wrapper around OpenCV's cv2.VideoCapture class. It hides all the messy details of connecting to the hardware and dealing with different operating systems (like Windows vs. Linux). This keeps our main code clean and easy to read.

- Stereotype: «Entity»
- Visibility: Public

Method Signature	Return Type	Vis.	Description and Contract
__init__(camera_index: int, api_preference: int)	None	+	Pre: None. Post: Sets up the camera settings but doesn't connect to the hardware just yet.
open_stream() -> bool	bool	+	Pre: The camera is physically plugged in. Post: Connects to the camera. Returns True if it works, or throws a CameraNotFoundException if it fails to connect.
read_frame() -> np.ndarray	np.ndarray	+	Pre: open_stream() worked. Post: Grabs the latest image from the camera and returns it as a numpy array.
release() -> None	None	+	Post: Safely disconnects the camera so it doesn't cause memory leaks when the app closes.
_set_properties(width: int, height: int) -> None	None	-	Post: Forces the camera to a specific resolution and turns off auto-exposure. This is important for getting clear UV images in the darkroom.

3.1.2 YOLO Engine

This class handles the AI part. It runs the YOLOv11n model using ONNX. We use ONNX instead of regular PyTorch because it's much faster on a normal CPU and avoids Python's speed limits. It takes the raw outputs from the model and figures out exactly where the aflatoxin is.

- Stereotype: «Service»
- Visibility: Public

Method Signature	Return Type	Vis.	Description and Contract
__init__(model_path: str, conf_threshold: float, iou_threshold: float)	None	+	Pre: We have a valid .onnx model file. Post: Starts the ONNX session using the CPU and sets our confidence limits.
predict(frame: np.ndarray) -> List	List	+	Pre: The image is a valid numpy array. Post: Runs the AI model and returns a list of detected aflatoxin spots with their coordinates.
_preprocess(frame: np.ndarray) -> np.ndarray	np.ndarray	-	Post: Resizes the image to 640x640, changes the colors to RGB, and prepares it exactly how the model expects it.
_postprocess(outputs: np.ndarray) -> List	List	-	Post: Takes the raw AI output and cleans it up. It removes overlapping boxes (NMS) and drops predictions that aren't confident enough.

3.2 Figion Data Component

This part of the code handles saving and loading data. We use the Data Access Object (DAO) pattern so our main application logic doesn't have to worry about writing raw SQL queries. It also makes it easier to change the database later if we need to.

3.2.1 Database Connection Factory

This class simply gives us a safe connection to the SQLite database. It makes sure we don't run into issues when multiple parts of the app try to talk to the database at the same time.

- Stereotype: «Factory»
- Visibility: Public

Method Signature	Return Type	Vis.	Description and Contract
get_connection(db_path: str) -> sqlite3.Connection	sqlite3.Connection	+	Static Method. Post: Returns a safe connection to the database and turns on features that help with reading and writing simultaneously.

3.2.2 SessionDAO

This class saves and updates information about each scanning session in our database. A session is just a batch of figs processed by the worker.

- Stereotype: «Data Access Object»
- Visibility: Public

Method Signature	Return Type	Vis.	Description and Contract
__init__(conn: sqlite3.Connection)	None	+	Pre: Needs an open database connection.
create_session(batch_id: str) -> int	int	+	Post: Adds a new row to the Sessions table with the current time and returns the new session ID.
update_session_totals(session_id: int, total_figs: int, total_contaminated: int)	None	+	Post: Updates the final counts for a session when the user stops scanning.
export_to_csv(session_id: int, file_path: str) -> bool	bool	+	Post: Grabs the session data and writes it out to a CSV file so we have a physical report.

3.2.3 ImageArchiver

Saving large images to the hard drive takes time. If we did this on the main screen, the app would freeze. So, ImageArchiver runs in the background and saves images without slowing down the rest of the system.

- Stereotype: «Worker»
- Visibility: Public

Method Signature	Return Type	Vis.	Description and Contract
enqueue_image(frame: np.ndarray, metadata: dict)	None	+	Post: Puts the image into a waiting line (queue) to be saved later. It finishes instantly so the main app can keep running.
_writer_loop() -> None	None	-	Post: This runs continuously in the background. It takes images out of the waiting line and saves them to the computer's hard drive.

3.3 Figion Hardware Component

Our system needs a dark environment with special high-intensity UV lights. To keep the lights from overheating, we don't leave them on all the time. This component turns them on and off using a USB relay board.

3.3.1 RelayController

This class sends simple commands over a serial port to the USB relay to control the UV LEDs.

- Stereotype: «Control»
- Visibility: Public

Method Signature	Return Type	Vis.	Description and Contract
__init__(com_port: str, baud_rate: int)	None	+	Post: Sets up the basic settings for the serial connection.
connect() -> bool	bool	+	Post: Tries to open the connection to the relay. Returns False if it fails, instead of crashing the app.
set_state(state: bool) -> None	None	+	Post: If state is True, it sends the command to turn the lights on. If False, it turns them off.

3.4 Figion UI Component

This is the user interface built with PyQt6. The biggest challenge here is keeping the screen responsive. If the AI takes too long to process an image, the whole app could freeze. We solve this by using background workers and Qt's signal-slot system.

3.4.1 VideoProcessorWorker

This is our background worker. It continuously grabs frames, runs the AI, and then sends the results back to the main screen. By doing this in the background, the UI stays smooth and responsive.

- Stereotype: «Control»
- Visibility: Public

Method Signature / Signal	Return Type	Vis.	Description and Contract
frame_processed(np.ndarray, dict)	pyqtSignal	+	Signal. Sent out every time an image is fully processed. It carries the image with drawn boxes and the detection stats.
error_occurred(str)	pyqtSignal	+	Signal. Safely sends error messages (like "Camera disconnected") to the main screen.
__init__(camera: CameraManager, engine: YOLOONNXEngine)	None	+	Post: Gives the worker access to the camera and the AI engine.
run_pipeline() -> None	None	+	Slot. The main loop. It reads a frame, runs the AI, draws the boxes, queues the image to be saved, and sends the frame_processed signal.
stop() -> None	None	+	Slot. Safely tells the worker to stop running its loop.

3.4.2 MainWindow

This is the main screen the user sees. It receives updates from the VideoProcessorWorker and updates the live video feed and the statistics counters.

- Stereotype: «Boundary»
- Visibility: Public

Method Signature	Return Type	Vis.	Description and Contract
__init__()	None	+	Post: Sets up the screen, connects to the database, starts the background worker, and links all the signals and slots together.
on_start_btn_clicked() -> None	None	+	Slot. Called when the user clicks "Start". It creates a new session, turns on the UV lights, and tells the worker to begin.
on_stop_btn_clicked() -> None	None	+	Slot. Called when the user clicks "Stop". It pauses the worker, turns off the lights, and updates the final session stats.
update_ui(frame: np.ndarray, stats: dict)	None	+	Slot. Receives the frame_processed signal. It updates the video player on the screen and changes the live numbers on the dashboard.

4. 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/

5. Python GUIs. (t.y.). Multithreading PyQt6 applications with QThreadPool. https://www.pythonguis.com/tutorials/multithreading-pyqt6-applications-qthreadpool/