CSE 396 – Spring 2025
Final Report-Documentation

SD Belt

(Scan & Detect Belt)

Group 15 a.k.a. “Belt Innovations”

01.06.2025

Belt Innovations	Image Processing & AI	Hardware	Mobile Application	Desktop Application	Database and Backend
Ahmet Abdulgaffar Fettahoğlu			+		+
Büşra Sarı	+				+
Cemil Fatih Yol	+		+		
Kağan Çakıroğlu		+		+	
Mehmet Hayrullah Özkul			+		+
Mert Emir Şeker		+		+	
Osmancan Bozali			+		+
Serkan Efe Çamoğlu	+			+	
Ziya Kadir Tokluoğlu	+	+			

Table of Contents

1. YOLO-Based Image Processing Module

1.1 Introduction

This section outlines the comprehensive pipeline developed for a fruit disease detection system, leveraging the YOLOv8 architecture. The process encompasses dataset acquisition and curation via Roboflow, meticulous dataset preprocessing, training a compact and efficient YOLOv8 model, and its subsequent conversion and optimization into a Hailo-8L compatible `.hef` binary. The primary objective is to deploy a fast and accurate detector on the Hailo AI Hat, integrated within a C++ application that employs intelligent image processing for real-time, on-device analysis. This system is designed for applications such as identifying healthy versus rotten fruit instances on a conveyor belt, forming a critical component of an automated quality control process.

1.2 Technology Stack & Methodology

The core technologies and methodologies underpinning this module include:

- **Dataset Management**: Roboflow for dataset collection, annotation, and versioning.
- **AI Framework**: Ultralytics YOLOv8 (specifically `yolov8n.pt` variant) for its optimal balance of speed, accuracy, and suitability for edge deployment.
- **Hardware Acceleration**: Hailo-8L AI Accelerator for low-latency inference on the edge device (Raspberry Pi).
- **Programming Language**: C++ for the runtime application, providing system control and efficient model execution. Python for dataset preprocessing scripts and model training.
- **Image Processing Library**: OpenCV for extensive image manipulation, runtime preprocessing, and visualization tasks within the C++ application.
- **Model Conversion**: ONNX for intermediate model representation, facilitating the transition to the Hailo-8L specific format.

The detection methodology involves training the YOLOv8 model to classify specific fruit types and their condition (e.g., "Apple_Healthy", "Apple_Rotten") based on visual features. At runtime, the C++ application further employs specific image processing techniques to optimize when and how the YOLOv8 model is invoked.

1.3 Dataset Collection and Preparation

A robust dataset is fundamental for training an effective object detection model. Our approach involved meticulous collection and preparation stages:

1.3.1 Dataset Collection from Roboflow

- **Project Setup**: A new object detection project was created on Roboflow. Raw images of various fruits, encompassing both healthy and rotten examples (Apples, Oranges, Potatoes), were imported.

- **Annotation**: Bounding boxes were drawn around each fruit instance in the images, and these were annotated with predefined classes indicating both the fruit type and its condition (e.g., Apple_Healthy, Apple_Rotten, Potato_Healthy, Potato_Rotten, Orange_Healthy, Orange_Rotten).
- **Version Configuration**: For dataset generation, an output resolution of 640x640 pixels was set. Data augmentation techniques, including mosaic augmentation, were enabled to increase dataset variability. The dataset was exported in the YOLOv8 format.
- **Download Structure**: The downloaded dataset was organized into train/, valid/, and test/ splits, each containing images/ and labels/ subdirectories.

1.3.2 Dataset Preprocessing Scripts

To ensure data quality and improve model training, several custom Python scripts were utilized:

- **Label Remapping (remap_labels.py)**: This script standardizes class labels across the dataset by adjusting class indices in the label files according to a predefined mapping (e.g., CLASS_MAP = {"0":"0", ..., "5":"5"}). This is crucial if initial annotations or different dataset versions use inconsistent class IDs.
- **Label Analysis (count_classes.py, tools/count_null.py)**: These tools were used to count per-class annotation instances and identify any images with empty or missing label files, ensuring data integrity.
- **Class Distribution Balancing (compute_prune_targets.py, balance_count.py)**: To prevent model bias towards over-represented classes, these scripts first analyze class distributions to suggest pruning targets and then selectively prune samples to achieve a more balanced dataset.
- **Addition of False Positive Samples (add_false_samples.py)**: To improve the model's ability to reject non-target objects and reduce false positives, a set of blank or irrelevant images (e.g., 300 images from a false_samples folder) were strategically added to the training split.

1.4 Model Training with YOLOv8

The YOLOv8 model was trained using the prepared dataset:

1. **Environment**: Python 3.11 and Ultralytics YOLOv8 version 8.3 were used for training.
2. **Data Configuration (data.yaml)**: A YAML file was created to specify paths to the training, validation, and test image sets, the number of classes (nc: 6), and the names of the classes.
3. **Training Command**: The model training was initiated using the yolo train command with yolov8n.pt as the base model. Key parameters included 50 epochs, an image size of 640x640, a batch size of 16, and enabling standard and RandAugment auto-augmentation techniques.

4. **Monitoring**: Training progress was monitored to ensure key metrics, such as mAP_{50} (mean Average Precision at IoU 0.50), exceeded 0.99 on both training and validation sets, with particular attention paid to the performance of under-represented or challenging classes. The best performing weights (best.pt) were saved for a subsequent export.

1.5 Model Export and Hailo-8L Transformation

For deployment on the Hailo-8L AI accelerator, the trained PyTorch model (best.pt) underwent a multi-step conversion and optimization process:

1.5.1 Export to ONNX

The best.pt model was exported to the ONNX (Open Neural Network Exchange) format using the YOLO export tool. This step included simplifying the model graph, specifying an opset version (e.g., 17), and excluding NMS (Non-Maximum Suppression) operations from the exported graph, as NMS is typically handled in post-processing on the host device. The image size was maintained at 640x640. This process generated a best.onnx file.

1.5.2 Hailo-8L Transformation Flow

The ONNX model was then processed using the Hailo AI Software Suite, typically within a Docker container environment:

1. **Parse ONNX to HAR (Hailo Archive)**: The hailo parser converted the best.onnx model into Hailo's internal HAR format, specifying the target hardware architecture (hailo8l) and input tensor shapes (e.g., [1,3,640,640]).

2. **Calibration Data Preparation**: A calibration dataset (calib_data.npy) was created using a representative set of images (e.g., 200 images from a calib_imgs folder). These images were resized to 640x640, converted from BGR to RGB, and normalized (pixel values to [0,1] float32). This dataset is crucial for the quantization step.

3. **Quantization (Optimize HAR)**: The hailo optimize tool performed post-training quantization on the best.har file using the prepared calibration dataset, producing a quantized model (best_q.har). This step significantly reduces model size and improves inference speed on integer-based AI hardware like the Hailo-8L, with minimal impact on accuracy.

4. **Compilation to HEF (Hailo Executable Format)**: Finally, the hailo compiler compiled the quantized best_q.har into a .hef binary file (best_q.hef), which is the executable format directly deployable on the Hailo-8L accelerator.

The resulting best_q.hef file is then ready to be copied to the Raspberry Pi for use by the C++ runtime application.

1.6 Runtime Image Processing and Inference Pipeline (C++ Application)

Once the .hef model is deployed on the Raspberry Pi with the Hailo AI Hat, the C++ application manages the real-time inference pipeline. This pipeline incorporates intelligent image processing to efficiently utilize the AI accelerator.

1.6.1 Intelligent Frame Triggering and Region of Interest (ROI) Management

To optimize resource usage, the system does not perform YOLO inference on every frame from each camera. Instead, a triggering mechanism implemented in each camera's dedicated grabLoop function intelligently selects frames:

- **Dynamic ROI Determination**: On startup, each camera uses functions like firstVerticalLineXsFromCenter (analyzing Hough Lines on Canny edges) to identify the conveyor belt area, which is then cropped using cropBetweenXs to define an active ROI.
- **Background Normalization & Object Isolation**: The initial cropped background ROI is processed with whiteOutSameTone (using meanCenterRGB for reference) to create a normalized, high-contrast background. Incoming frames are similarly processed. This isolates potential objects by making the empty belt appear uniformly white.
- **Change Detection & Object Presence**: The diffCentroidTol function compares the currently processed frame against a reference (previous frame or normalized background) to detect significant pixel changes, indicating object presence and movement.
- **Object Centering & Inference Trigger**: If a change is detected, isCenterBetweenPoints verifies if the object is sufficiently centered within the ROI before a frame is flagged for full YOLO inference.
- **Cooldown Mechanism**: A CAPTURE_COOLDOWN period prevents multiple rapid-fire inferences on the same object as it passes through the camera's view.

1.6.2 Preprocessing for YOLOv8n Inference (Runtime)

Frames selected by the triggering logic are then specifically prepared for the YOLOv8n model:

- **Resolution Adjustment**: Frames are resized to the model's required input (e.g., 640x640).
- **Normalization**: Pixel values are scaled (e.g., to [0,1]).
- **Channel Ordering & Tensor Formatting**: Color channels are ordered (BGR to RGB), and the data is converted to the tensor format expected by the Hailo runtime.

1.6.3 Real-Time Inference Pipeline Steps

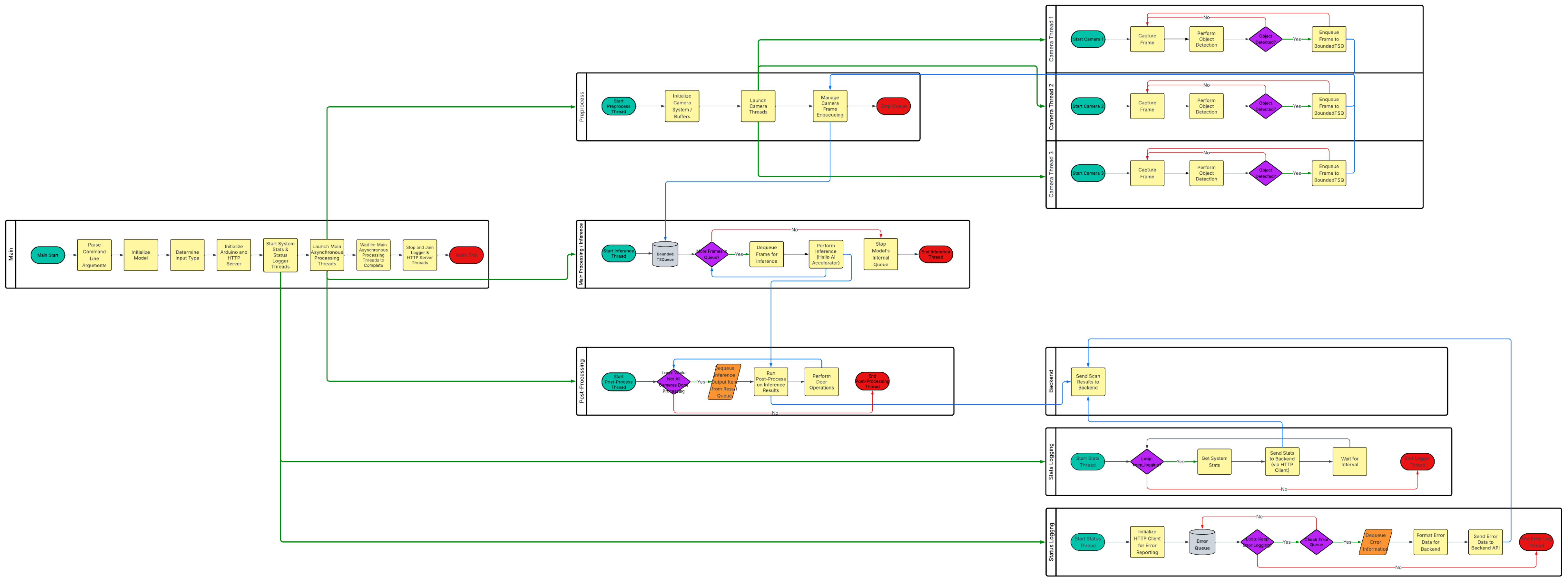
The overall runtime data flow is:

1. **Multi-Camera Frame Acquisition**: Dedicated threads capture frames from USB webcams.
2. **Intelligent Frame Preprocessing & Triggering**: Each camera thread applies the ROI, background normalization, change detection, and centering logic.
3. **Frame Enqueueing**: Triggered frames (containing potential objects) are added to a thread-safe queue (preprocessed_queue).
4. **YOLO-Specific Preprocessing & Inference**: An inference thread dequeues frames, performs final YOLO-specific preprocessing, and executes the YOLOv8n model using the Hailo AI accelerator.
5. **Post-processing**: Results (bounding boxes, class IDs, confidences) are passed to another queue (results_queue) for a post-processing thread. This thread applies NMS, filters by confidence, and interprets the results (e.g., identifying fruit type and condition).
6. **Action Trigger & Data Transmission**: Based on the final detection, actions like "Perform Door Operations" can be initiated, and/or detection data (e.g., ScanRequestDTO) is formatted and sent to a backend system via an HTTP client.

1.7 Conclusion

This comprehensive workflow, from dataset acquisition on Roboflow and targeted preprocessing scripts to YOLOv8n model training and its meticulous conversion for the Hailo-8L platform, culminates in an efficient .hef binary. When deployed within our C++ application, this model is augmented by intelligent runtime image processing techniques that optimize frame selection for inference. This multi-stage approach ensures a fast, accurate, and resource-conscious fruit disease detection system suitable for real-world deployment on edge devices like the Raspberry Pi equipped with a Hailo AI Hat.

1.8 Multi-Threaded Application Architecture for YOLOv8 Fruit Classifier on Hailo-8L


Main
Main Start
Parse Command Line Arguments
Initialize Model
Determine Input Type
Initialize Arduino and HTTP Server
Start System Stats & Status Logger Threads
Launch Main Asynchronous Processing Threads
Wait for Main Asynchronous Processing Threads to Complete
Stop and Join Logger & HTTP Server Threads
Main End
Preprocess
Start Preprocess Thread
Initialize Camera System / Buffers
Launch Camera Threads
Manage Camera Frame Enqueueing
Stop Queue
Camera Thread 1
Start Camera 1
Capture Frame
Perform Object Detection
Object Detected?
Yes
No
Enqueue Frame to BoundedTSQ
Camera Thread 2
Start Camera 2
Capture Frame
Perform Object Detection
Object Detected?
Yes
No
Enqueue Frame to BoundedTSQ
Camera Thread 3
Start Camera 3
Capture Frame
Perform Object Detection
Object Detected?
Yes
No
Enqueue Frame to BoundedTSQ
Main Processing / Inference
Start Inference Thread
Bounded TSQueue
More Frames in Queue?
Yes
No
Dequeue Frame for Inference
Perform Inference (Hailo AI Accelerator)
Stop Model's Internal Queue
End Inference Thread
Post-Processing
Start Post-Process Thread
Loop: While Not All Cameras Done Processing
Yes
No
Dequeue Inference Output Item from Result Queue
Run Post-Process on Inference Results
Perform Door Operations
End Post-Processing Thread
Backend
Send Scan Results to Backend
Stats Logging
Start Stats Thread
Loop: keep_logging?
Yes
No
Get System Stats
Send Stats to Backend (via HTTP Client)
Wait for Interval
End Logger Thread
Status Logging
Start Status Thread
Initialize HTTP Client for Error Reporting
Error Queue
Loop: Keep Error Logging?
Yes
No
Check Error Queue
Yes
No
Dequeue Error Information
Format Error Data for Backend
Send Error Data to Backend API
End Error Log Thread

1.9 Test Cases for YOLO-Based Image Processing Module

Feature	**Purpose**	**Test Input / Action**	**Expected Result**	**Result**
Fruit Detection	Verify that one fruit in a frame is correctly detected	A single image containing a fruit on a plain background	Exactly one bounding box is returned), with confidence ≥ C	**PASSED**
No-False-Positive Detection	Confirm module does not falsely flag pristine items	A blank image that doesn't contain any fruit	Zero "defect" boxes	**PASSED**
Multi-Camera Stream Handling	Validate that three camera feeds can be ingested and processed in parallel without dropping frames	Three simultaneous video streams (cam1.mp4, cam2.mp4, cam3.mp4) each containing n for each	All 3n frames (n per stream) processed; inference time per frame ≤ 0.04 s; no dropped frames	**PASSED**
Confidence Threshold Filtering	Test that detections below configured threshold are discarded	An image with one fruit detection at confidence = C (below threshold T)	The low-confidence detection is filtered out; no bounding box returned	**PASSED**
JSON Output & Schema Validation	Check that module outputs detection results in correct JSON format	Run detection on a test image and capture the module's JSON output	JSON contains keys: `product_id`, `timestamp`, `class_name`, `confidence`, `bbox` (with 4 coords); types and nesting match the API spec	**PASSED**
Real-Time Throughput	Verify pipeline meets minimum FPS requirement	Feed a 60-second 30 FPS test video to the module on Raspberry Pi	Sustained processing at ≥ 25 FPS over the entire video	**PASSED**

Low-Light / High-Glare Robustness	Ensure detection still works under challenging lighting	A pair of test images of fruits under dim light and under direct glare	Fruits still detected with confidence ≥ C;	**PASSED**
Background Separation	Ensure input frame preprocessed as removing background and handing more robust input to model	A pair of raw test images	Average background calculated and apply whiteout effect to areas other than object.	**PASSED**
Hailo model loading	Model is loaded with proper hailo8 light architecture	.hef model that compiled with hailortcli 4.20.0 (hailo8 light architecture)	Model is compatible w2ith raspberry pie's hailortcli version and architecture	**PASSED**
Object Centering & Timed Capture	Verify that an image is captured when thedetected object's center of mass aligns with the predefined central x-axis of the frame, after background normalization.	Move an object across the camera's field of view, ensuring it passes through the center.	An image of the object is captured precisely when its calculated center of mass is detected at the pre-set central x-axis trigger point	**PASSED**

2. Hardware Module

2.1 Overview

The hardware infrastructure of the SD Belt system consists of a conveyor belt powered by a 24V DC motor, controlled by an Arduino via a BTS7960B motor driver. The Arduino receives motion commands (start, stop, speed, direction) over a serial connection from a Raspberry Pi, which acts as the central controller.

A regulated voltage module converts 24V to 5V for logic components like the servo motor (TowerPro MG996R), which is used for sorting defective products based on classification results.

2.2 Camera and AI Integration

3 USB webcams are positioned around the belt, two on the sides that are not directly opposite each other and one mounted above, to provide multi-angle views of the products. These cameras are connected directly to the Raspberry Pi, which captures video streams, processes the frames using a YOLOv8n model, and determines in real time whether a product is defective.

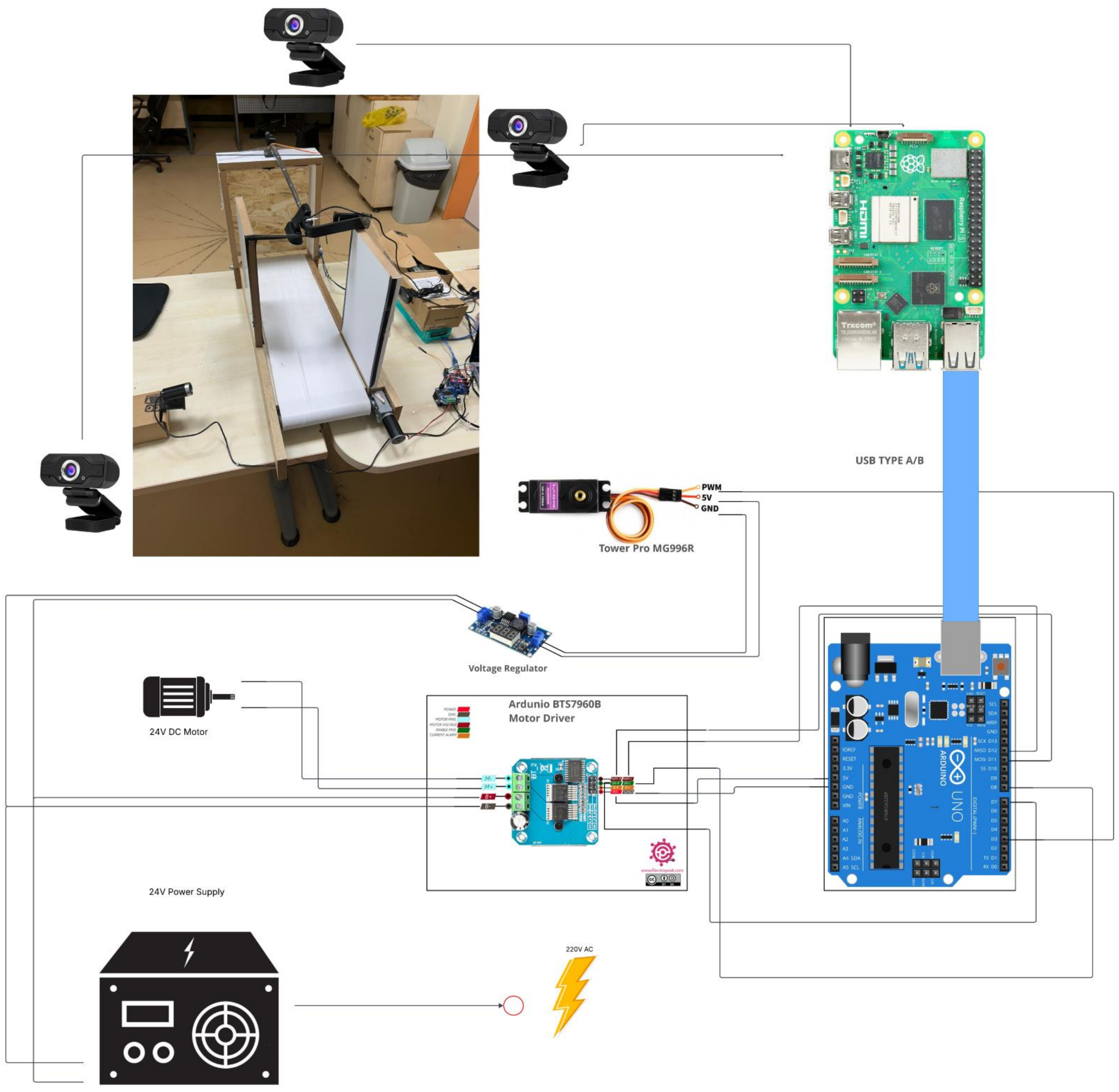
2.3 Arduino–Raspberry Pi Communication

A direct USB serial link is established between the Arduino and Raspberry Pi. The Pi sends text-based commands like PCT:50, STOP:0, or DIR:1, which the Arduino parses and converts into motor actions. This approach ensures real-time belt control with low latency.

2.4 Separation Mechanism

At the end of the conveyor belt, a gate mechanism controlled by a high-torque MG996R servo motor is used to sort products. Upon receiving classification results from the Raspberry Pi, the Arduino triggers the servo to open or close the gate, directing products to either the accepted or rejected side.

2.5 SD Belt Workflow



2.6 Test Cases for Hardware Module

Feature	Purpose	Test Input / Action	Expected Result	Result
SD Belt Motor Activation	Verify that the motor responds correctly to control signals from Arduino.	Power on system and send "start" command from Raspberry Pi to Arduino.	SD Belt begins moving smoothly and at consistent speed.	**PASSED**
Motor Stop Command	Ensure that the belt stops when a "stop" command is sent.	Send stop command from Raspberry Pi to Arduino while belt is in motion.	SD Belt stops within 2 seconds.	**PASSED**
Power Supply Check	Validate that the power supply provides stable 24V under load.	Measure voltage across motor terminals while the SD Belt is running.	Voltage remains within 23–25V during operation.	**PASSED**
Arduino-Pi Connection	Ensure serial connection between Raspberry Pi and Arduino is active.	Disconnect and reconnect the serial cable and observe command transmission.	Motor responds again after reconnection.	**PASSED**
SD Belt Frame Stability	Confirm that mechanical frame holds the system steady.	Run the belt at full speed with a standard load placed on it.	No wobble or excessive vibration observed.	**PASSED**
Camera Mounting Plan	Plan and verify suitable positions for multiple camera modules.	Attach mock-up cameras or placeholders at designated positions above the belt.	At least 3 camera viewpoints offer full coverage of product angles.	**PASSED**
Camera Wiring Plan	Ensure power and data paths for camera modules are feasible.	Plan wiring paths from Pi to 3-4 camera positions without overlap or signal interference.	Cabling layout is clean and Raspberry Pi ports are reachable for each camera.	**PASSED**

Raspberry Pi - Arduino Serial Communication	Verify that motor control commands are reliably transmitted from Raspberry Pi to Arduino via serial interface.	Send start/stop commands over serial and observe Arduino motor response.	Motor responds accordingly to the commands without delay.	**PASSED**
Product Transport by SD Belt	Confirm that the SD Belt can transport sample products without obstruction or slippage.	Place a sample product on the belt and activate motion.	Product is carried to the other end without misalignment or falling.	**PASSED**
Camera-Based Product Capture	Verify that the camera records passing products during belt movement for future processing.	Activate camera while the belt is moving with sample products.	Camera captures visual data correctly during motion.	**PASSED**
Webcam Fallback Test	Ensure that a USB webcam can be used as a backup in case of Raspberry Pi camera module failure.	Disconnect Pi camera and connect USB webcam, then run camera test application.	Video feed is displayed and functions equivalently to Pi camera module.	**PASSED**
Servo Motor Sorting Test	Ensure the servo motor correctly moves the wooden arm to sort products based on commands.	Send sort-left and sort-right commands to the Arduino and observe servo movement.	Servo rotates arm in correct direction and pushes product accordingly.	**PASSED**

3. Desktop Application Module

3.1. Overview

The SD-Belt system includes a Qt-based desktop application written in C++, serving as a control and monitoring hub for the entire conveyor infrastructure. The main class, MainWindow, orchestrates page switching between the Dashboard, Camera Views, and Logs using a stacked widget layout. It also dynamically loads and applies QSS-based styles from bundled files (MainWindow.qss, MenuButton.qss, Header.qss, etc.) to ensure a consistent and modern interface design.

3.1.1. Camera Receiver

Real-time visual inspection is handled by the Receiver class defined in CameraReceiver.h/cpp. This widget uses QUdpSocket to listen on port 5000 for incoming image data from the Raspberry Pi. Each camera sends its JPEG-encoded frames over UDP, prefixed with a camera ID. The system supports three cameras, and each frame is rendered in a QLabel, scaled to fit and displayed within a grid layout. The design is optimized for low latency and avoids blocking the UI thread.

3.1.2. Logs

The Logs class interacts with the backend REST API (/scans) at periodic intervals via QNetworkAccessManager. It parses the received JSON logs and displays product inspection results in a QListWidget. Each entry shows product ID, health ratio, pass/fail status, and error messages if any. The background color of list items reflects their success status—green for passed, red for defective. Additionally, total product count and processing speed (items per minute) are updated in real time using QLabel widgets.

3.1.3. System Information

The SystemInfoRetriever class is responsible for retrieving real-time diagnostics from the Raspberry Pi, including CPU temperature, RAM usage, and CPU load. These metrics are displayed directly on the dashboard and are updated periodically to allow operators to monitor the system's health. Color indicators reflect system status and warn in case of overheating or connection loss.

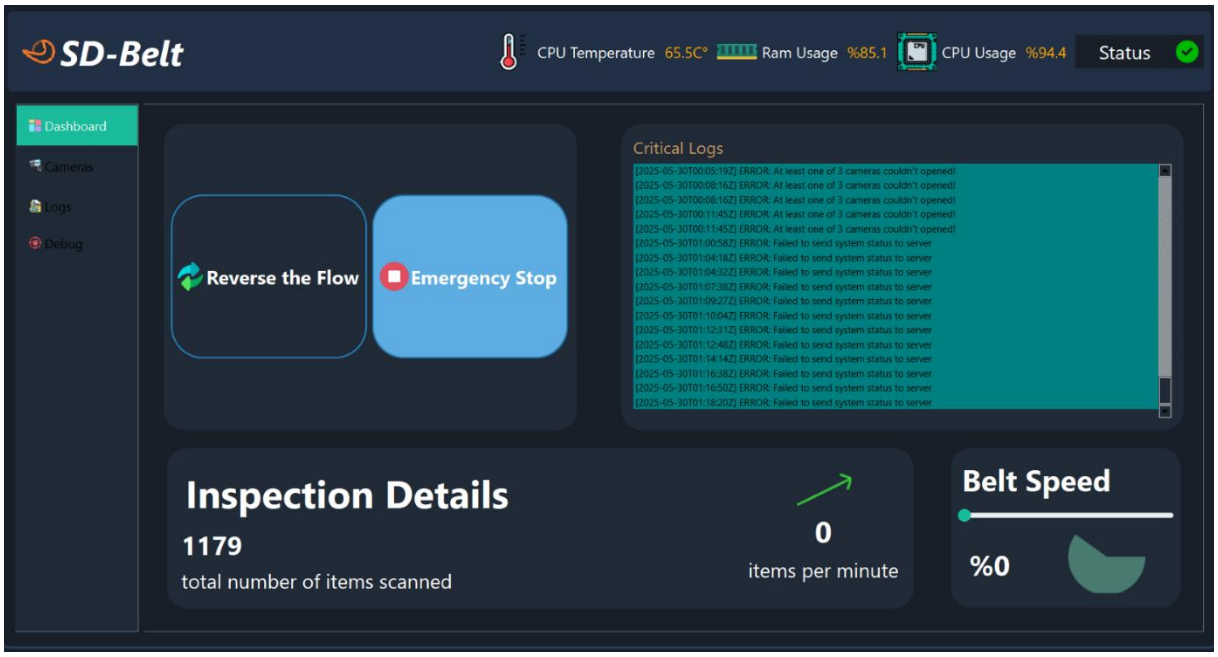
3.1.4. Backend Communication

All communication with the Raspberry Pi and backend server is handled asynchronously using Qt's networking module (QNetworkAccessManager, QNetworkRequest, QNetworkReply). API endpoints are defined in Globals.h, and token-based authentication is used. Control operations such as stopping the belt or reversing its direction are triggered through POST requests. Network connectivity is monitored to ensure robust operation.

3.1.5. Styles and UI Consistency

The application's look and feel are defined via modular QSS files. TextStyles.qss unifies typography across the application, while MenuButton.qss and MenuButtonSelected.qss control sidebar button states. Header.qss styles the header bar that includes system status indicators and application branding. These styles are loaded at runtime using QFile and applied programmatically to enhance visual consistency and maintainability.

3.2. Desktop App UI



```
[2025-05-30T00:05:19Z] ERROR: At least one of 3 cameras couldn't opened!
[2025-05-30T00:08:16Z] ERROR: At least one of 3 cameras couldn't opened!
[2025-05-30T00:08:16Z] ERROR: At least one of 3 cameras couldn't opened!
[2025-05-30T00:11:45Z] ERROR: At least one of 3 cameras couldn't opened!
[2025-05-30T00:11:45Z] ERROR: At least one of 3 cameras couldn't opened!
[2025-05-30T01:00:58Z] ERROR: Failed to send system status to server
[2025-05-30T01:04:18Z] ERROR: Failed to send system status to server
[2025-05-30T01:04:32Z] ERROR: Failed to send system status to server
[2025-05-30T01:07:38Z] ERROR: Failed to send system status to server
[2025-05-30T01:09:27Z] ERROR: Failed to send system status to server
[2025-05-30T01:10:04Z] ERROR: Failed to send system status to server
[2025-05-30T01:12:31Z] ERROR: Failed to send system status to server
[2025-05-30T01:12:48Z] ERROR: Failed to send system status to server
[2025-05-30T01:14:14Z] ERROR: Failed to send system status to server
[2025-05-30T01:16:38Z] ERROR: Failed to send system status to server
[2025-05-30T01:16:50Z] ERROR: Failed to send system status to server
[2025-05-30T01:18:20Z] ERROR: Failed to send system status to server
```

Inspection Details

1179

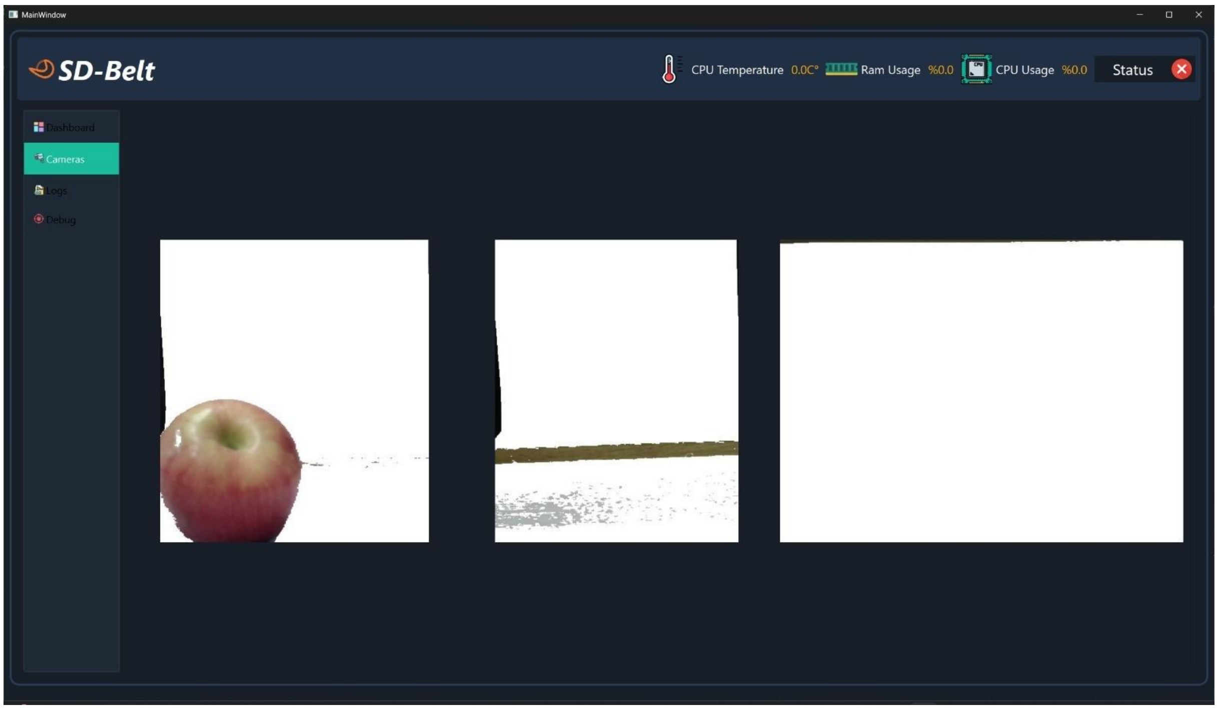
total number of items scanned

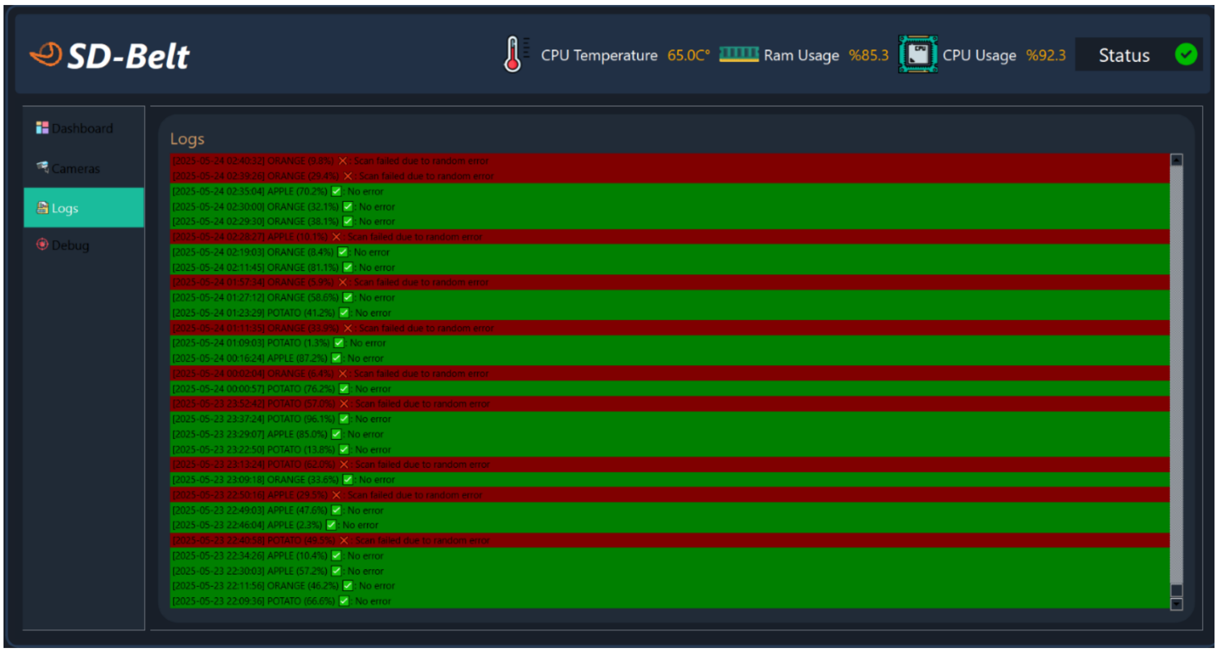
0

items per minute

Belt Speed

%0





```
[2025-05-24 02:40:32] ORANGE (9.8%) ✗: Scan failed due to random error
[2025-05-24 02:39:26] ORANGE (29.4%) ✗: Scan failed due to random error
[2025-05-24 02:35:04] APPLE (70.2%) ✓: No error
[2025-05-24 02:30:00] ORANGE (32.1%) ✓: No error
[2025-05-24 02:29:30] ORANGE (38.1%) ✓: No error
[2025-05-24 02:28:27] APPLE (10.1%) ✗: Scan failed due to random error
[2025-05-24 02:19:03] ORANGE (8.4%) ✓: No error
[2025-05-24 02:11:45] ORANGE (81.1%) ✓: No error
[2025-05-24 01:57:34] ORANGE (5.9%) ✗: Scan failed due to random error
[2025-05-24 01:27:12] ORANGE (58.6%) ✓: No error
[2025-05-24 01:23:29] POTATO (41.2%) ✓: No error
[2025-05-24 01:11:35] ORANGE (33.9%) ✗: Scan failed due to random error
[2025-05-24 01:09:03] POTATO (1.3%) ✓: No error
[2025-05-24 00:16:24] APPLE (87.2%) ✓: No error
[2025-05-24 00:02:04] ORANGE (6.4%) ✗: Scan failed due to random error
[2025-05-24 00:00:57] POTATO (76.2%) ✓: No error
[2025-05-23 23:52:42] POTATO (57.0%) ✗: Scan failed due to random error
[2025-05-23 23:37:24] POTATO (96.1%) ✓: No error
[2025-05-23 23:29:07] APPLE (85.0%) ✓: No error
[2025-05-23 23:22:50] POTATO (13.8%) ✓: No error
[2025-05-23 23:13:24] POTATO (62.0%) ✗: Scan failed due to random error
[2025-05-23 23:09:18] ORANGE (33.6%) ✓: No error
[2025-05-23 22:50:16] APPLE (29.5%) ✗: Scan failed due to random error
[2025-05-23 22:49:03] APPLE (47.6%) ✓: No error
[2025-05-23 22:46:04] APPLE (2.3%) ✓: No error
[2025-05-23 22:40:58] POTATO (49.5%) ✗: Scan failed due to random error
[2025-05-23 22:34:26] APPLE (10.4%) ✓: No error
[2025-05-23 22:30:03] APPLE (57.2%) ✓: No error
[2025-05-23 22:11:56] ORANGE (46.2%) ✓: No error
[2025-05-23 22:09:36] POTATO (66.6%) ✓: No error
```

3.3. Test Cases for Desktop Application Module

Feature	**Purpose**	**Input**	**Expected Result**	**Result**
Stop the Belt	Test whether the user can stop the SD-Belt via GUI and if the command is relayed through the backend to the Raspberry Pi.	`{ "type": "command", "action": "stop_belt", "source": "desktop", "target": "raspberry" }`	The belt stops within 2 seconds.	**PASSED**
Logs	Ensure that scanned product data is correctly fetched from backend logs and displayed in GUI.	Simulate product flow on the belt (with valid or defective items).	Inspection logs (ID, result, timestamp) are displayed in the GUI and match backend entries.	**PASSED**
System Health Monitoring	Validate that Raspberry Pi metrics are sent to	`{ "cpu_usage": "80.4", "cpu_temp": 61.2, "memory_usage":`	CPU temperature and memory usage are updated in the GUI panel.	**PASSED**

	the backend and displayed in GUI.	45.5 }		
Basic GUI Interaction with QT	Operating with interactive GUI elements such as (buttons, labels, input fields).	Verify that the user can interact with basic GUI components and see immediate visual feedback.	The GUI layout remains responsive and delivers the expected actions with visual feedbacks.	**PASSED**
Proper Layering of QT Widgets	Making sure of UI experience is flawless and clean	Verify that layers on top of each other acts as expected based on input.	Where Pointer(Mouse) is interacted on screen will get effected by input only.	**PASSED**
UDP-Based Camera Stream	Verify that JPEG-encoded image frames sent directly from the Raspberry Pi via UDP are correctly received and displayed on the desktop GUI.	Live image data sent over UDP port 5000 from Raspberry Pi (with camera_id headers)	Each camera's stream is decoded and displayed correctly in its corresponding GUI panel, with no freezing or delay.	**PASSED**
Reverse the Flow	Verify that the user can reverse the belt's movement direction via the desktop GUI and the command is transmitted through the backend to the Raspberry Pi.	{ "type": "command", "action": "reverse_belt", "source": "desktop", "target": "raspberry" }	The belt changes its direction within 2 seconds.	**PASSED**
Adjust Belt Speed	Verify that users can change the belt speed from the GUI and the new value is communicated through the backend to the Raspberry Pi.	{ "type": "command", "action": "set_speed", "value": 85, }	Belt speed changes to the specified value and GUI reflects the update.	**PASSED**

4. Mobile Application Module

4.1 Introduction

The SDBelt mobile application is a key component of the AI-powered quality control belt system. Its primary goal is to provide real-time access to operational insights and remote control capabilities from handheld devices. It enables production supervisors and operators to monitor system health, track product scan outcomes, and issue system-level commands such as start, stop, and restart, all from within a clean, intuitive mobile interface. This module improves accessibility, enhances operational flexibility, and allows for rapid intervention in case of production anomalies.

4.2. Technologies Used

The mobile application was developed using modern cross-platform technologies with performance and modularity in mind:

- **Framework:** Qt 6 and QML for declarative UI programming and rapid prototyping.
- **Programming Languages:**
    - **C++:** For backend integration, logic, and REST API communications.
    - **QML/JavaScript:** For UI layout, event handling, and page logic.
- **Target Platform:** Android
- **Backend Communication:** RESTful API interaction using QNetworkAccessManager, ensuring asynchronous and non-blocking communication.

4.3. Application Architecture and Components

The application is divided into two main layers: service logic written in C++ (ApiService) and a QML-based front-end composed of individual screens (pages) for different functionalities.

4.3.1. ApiService (C++ Network Logic)

This class is the communication backbone of the mobile app:

- Built on QObject, it exposes invokable methods and emits signals back to the QML layer.
- Includes request preparation, JSON serialization, and context-based result mapping.
- Synchronous user interactions like login, and asynchronous operations like fetching weekly scan stats are supported.
- Each request is uniquely identified using context strings to link API responses to UI logic.

4.3.2. QML UI Pages

- **LoginPage.qml:** Handles user authentication, provides immediate error feedback, and controls navigation to the home screen upon successful login.
- **HomePage.qml:** Displays an overview of system status, recent product scans, and a summary of today's total scans. Designed to be the landing dashboard.
- **OperationsPage.qml:** Presents live system metrics such as CPU temperature and uptime. Enables remote start, stop, and restart operations with confirmation.
- **ProductsPage.qml:** Lists all products and their associated success/failure statistics. Highlights top-performing items and recent trends.
- **ProductStatsPage.qml:** Dedicated analytics view for a selected product. Fetches and renders scan performance over the past 7 days in a responsive chart layout.

Each page is visually themed using a consistent Material palette and layout grid.

4.4. Workflow and Development Process

The mobile module followed an iterative and modular development strategy, which allowed continuous testing, user feedback incorporation, and rapid feature additions.

4.4.1. Design Phase

- Requirements were derived from the core SDBelt architecture and operator needs.
- UI/UX wireframes and navigation flows were designed using Figma before QML implementation.
- Special attention was paid to clarity, large touch targets, and real-time responsiveness.

4.4.2. Development Phase

- ApiService was implemented first to ensure communication infrastructure was solid.
- QML views were created in parallel and integrated progressively.
- Each page utilized reusable components (e.g., status cards, chart blocks).
- Color themes and typography were bound to root-level properties to support future dark/light mode themes.

4.4.3. Testing Phase

- Android emulator and physical device testing were both conducted.
- Manual test cases were executed for every major use case, including error simulation.
- Debug logs were used extensively to monitor signal-slot flows and trace API responses.

4.5. Features

The mobile app offers a robust set of features that align with the operational goals of the SDBelt platform:

Feature	Description
Authentication	Secure login system with input validation and token-based session tracking.
Real-Time Dashboard	Visual overview of current scans, system uptime, and alert indicators.
Remote System Control	Start, stop, and restart commands accessible with confirmation dialogs.
Per-Product Analytics	Weekly performance metrics, including success/failure breakdowns and trend charts.
Error Handling	Automatic fallback behaviors for network errors, including user alerts and logging.
Optimized UI/UX	Material-style interface with smooth transitions and feedback cues for user actions.

4.6. Interface Highlights

Here are key user interface components and their role in system operation:

- **LoginPage:** Clean interface with password masking, loading indicators, and animated error prompts.
- **HomePage:** Summary cards (scanned today, last scans) and system metrics displayed with iconography and color codes.
- **OperationsPage:** Real-time system info (status, CPU temp, accuracy setting) with toggles for command actions.
- **ProductsPage:** Scrollable list of products with per-item summary bars and entry points to detailed stats.
- **ProductStatsPage:** Chart visualization of daily scan results, fault/error tracking, and visual indicators for incomplete data.

4.7. Test Cases for Mobile Application Module

Comprehensive testing was carried out with an emphasis on both functionality and resilience:

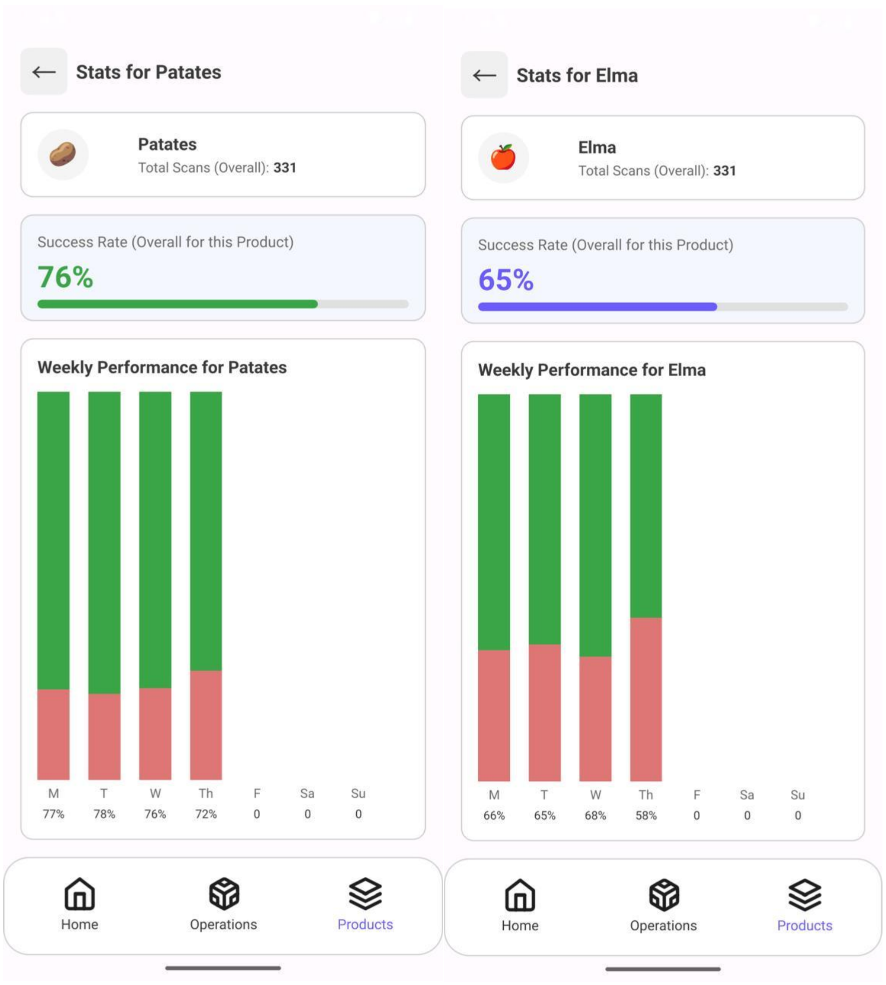
Feature	Purpose	Test Input / Action	Expected Result	Result
User Login	Verify login with credentials	Enter valid username and password	User is redirected to dashboard;	**PASSED**
Invalid Login	Prevent login with invalid credentials	Enter wrong password or username	Error message shown: “Invalid credentials”	**PASSED**
Daily Scan Statistics	Validate correct rendering of product scan statistics	Backend sends data: total: 10, valid: 8	App shows: 8 valid, 2 invalid	**PASSED**
Belt Accuracy Display	Ensure belt accuracy is correctly calculated and shown	Backend sends accuracy = 80%	App displays Belt Accuracy as 80% with correct formatting	**PASSED**
Belt Speed Visualization	Confirm belt speed updates correctly from backend	Speed rate = 3 sent from backend	App shows “Speed = 3” in the stats panel	**PASSED**
Real-Time Throughput	Verify pipeline sends live info	Backend constantly sends latest info	App updates display accordingly without needing manual refresh at a determined interval	**PASSED**
Belt Start Command	Ensure belt starts when command is triggered from UI	Tap “Start” button in control panel	Belt start command sent; UI reflects “Connected-Running”	**PASSED**

4.8. Challenges Faced

The mobile module development encountered several non-trivial challenges:

- **Contextual Asynchronous Responses:** The API design required unique context strings to differentiate between multiple pending requests. Handling this reliably in QML necessitated careful state tracking.
- **State Synchronization Across Pages:** Ensuring that updates in one page (e.g., product stats) were reflected across other pages without duplication or delay.
- **Error Tolerance:** Errors due to partial API responses, misformatted JSON, or dropped network connections needed robust UI fallback and logging strategies.

4.9 Screenshots


Stats for Patates

Patates
Total Scans (Overall): **331**

Success Rate (Overall for this Product)
76%

Weekly Performance for Patates

M	T	W	Th	F	Sa	Su
77%	78%	76%	72%	0	0	0

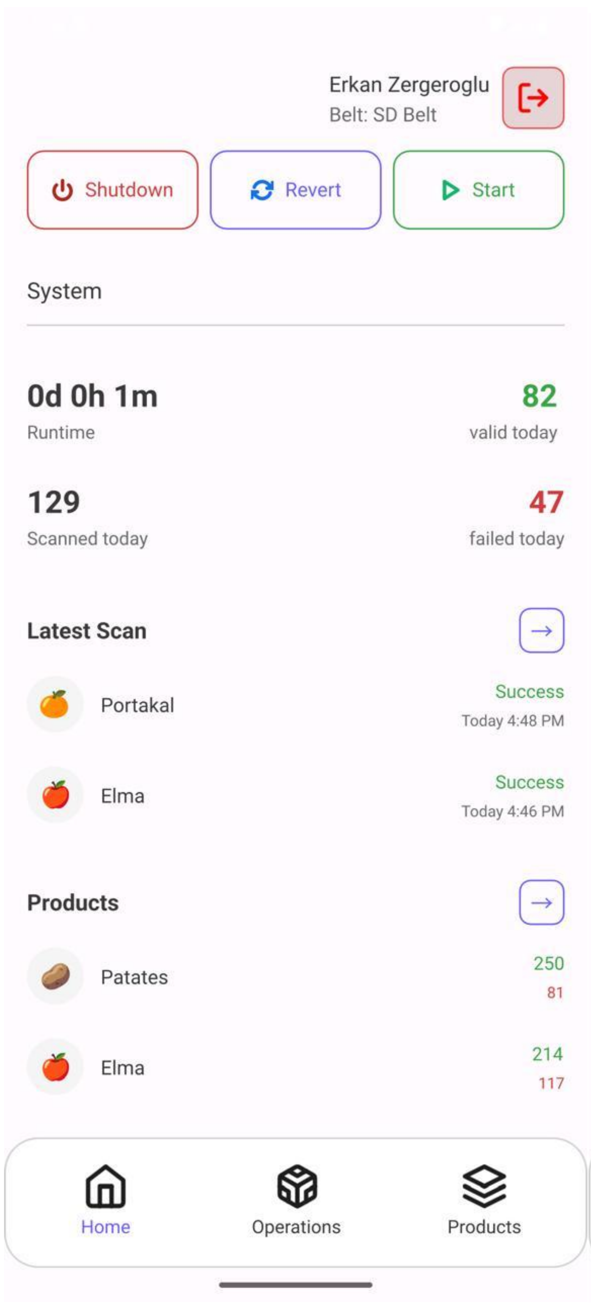
Stats for Elma

Elma
Total Scans (Overall): **331**

Success Rate (Overall for this Product)
65%

Weekly Performance for Elma

M	T	W	Th	F	Sa	Su
66%	65%	68%	58%	0	0	0



Erkan Zergeroglu
Belt: SD Belt

Shutdown | Revert | Start

System

0d 0h 1m
Runtime

82
valid today

129
Scanned today

47
failed today

Latest Scan

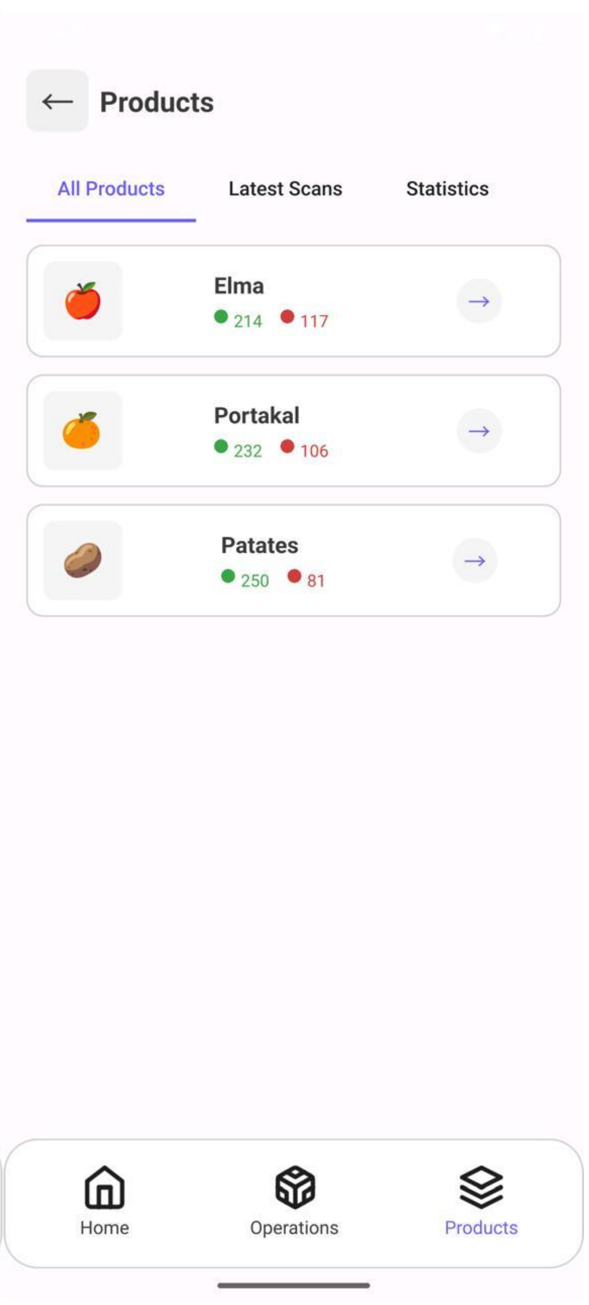
Portakal — Success — Today 4:48 PM

Elma — Success — Today 4:46 PM

Products

Patates — 250 — 81

Elma — 214 — 117



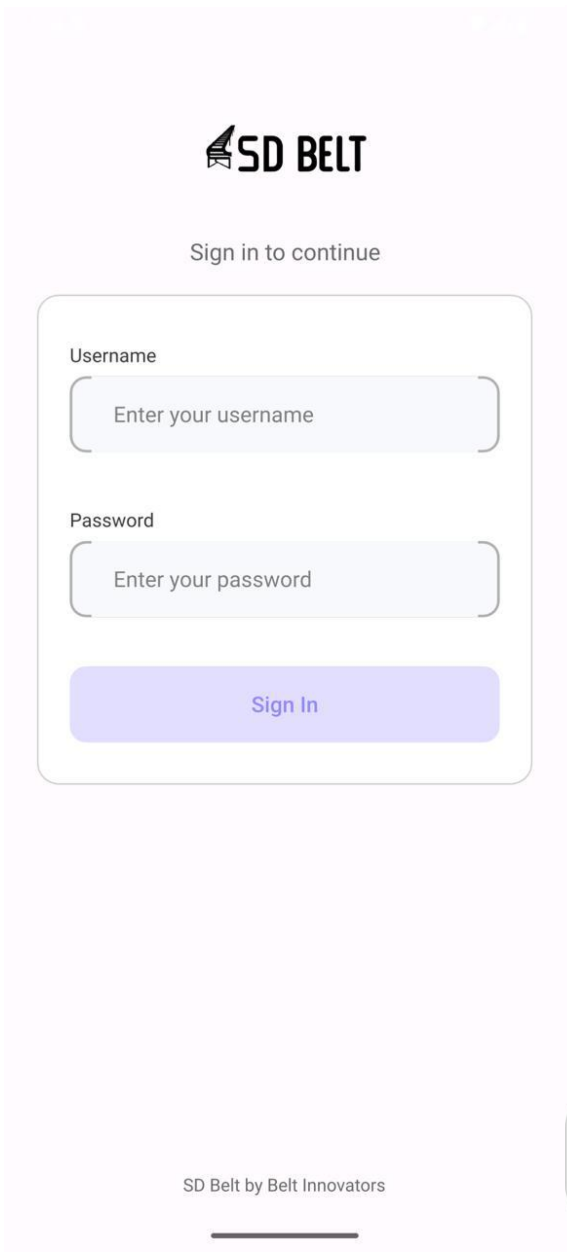
Products

All Products | Latest Scans | Statistics

Elma ● 214 ● 117

Portakal ● 232 ● 106

Patates ● 250 ● 81



SD BELT

Sign in to continue

Username

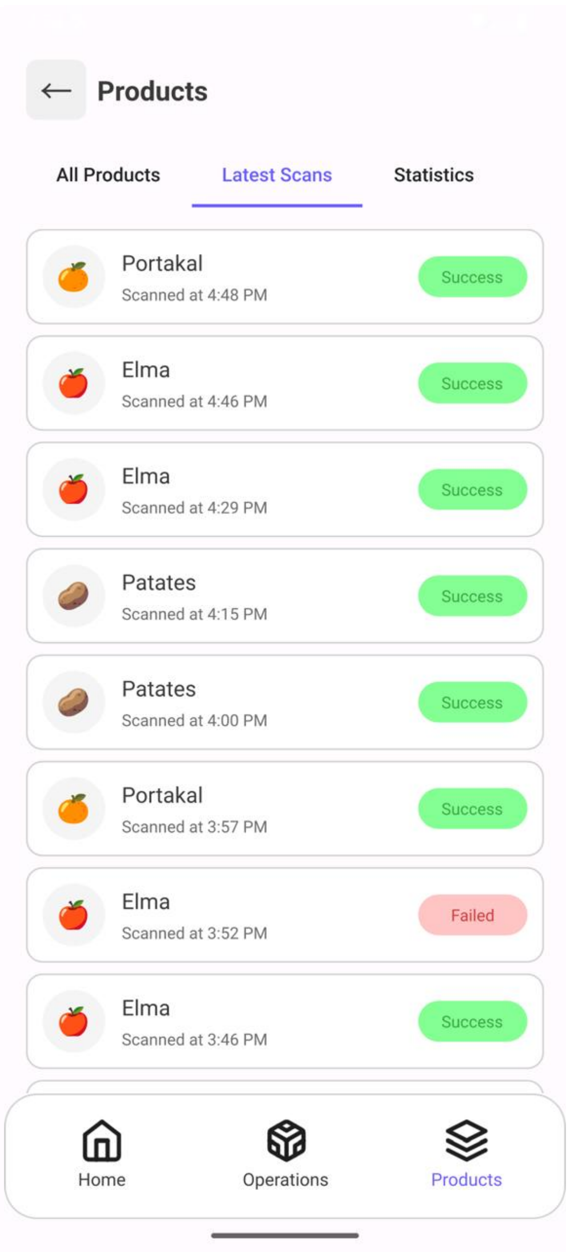
Enter your username

Password

Enter your password

Sign In

SD Belt by Belt Innovators

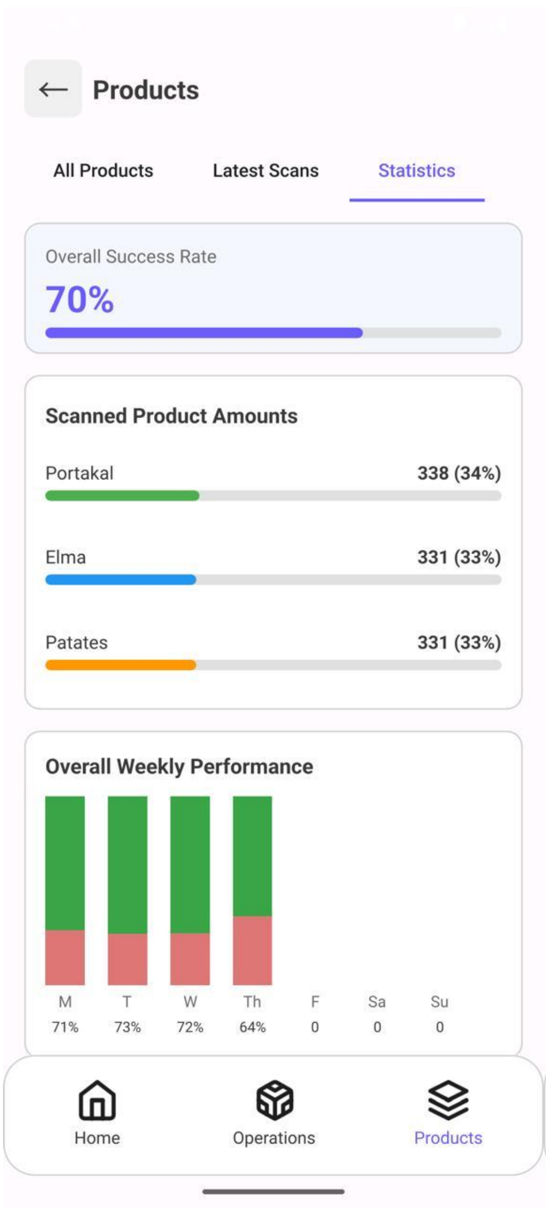


Products

All Products | Latest Scans | Statistics

Portakal	Scanned at 4:48 PM	Success
Elma	Scanned at 4:46 PM	Success
Elma	Scanned at 4:29 PM	Success
Patates	Scanned at 4:15 PM	Success
Patates	Scanned at 4:00 PM	Success
Portakal	Scanned at 3:57 PM	Success
Elma	Scanned at 3:52 PM	Failed
Elma	Scanned at 3:46 PM	Success

Home | Operations | Products



Products

All Products | Latest Scans | Statistics

Overall Success Rate

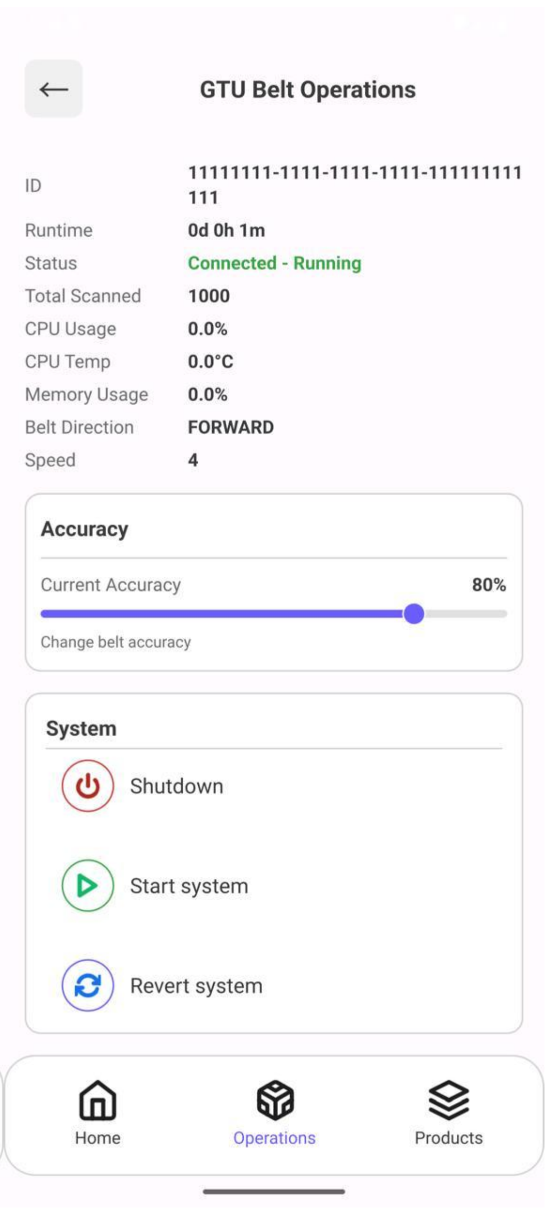
70%

Scanned Product Amounts

Portakal	338 (34%)
Elma	331 (33%)
Patates	331 (33%)

Overall Weekly Performance

M	T	W	Th	F	Sa	Su
71%	73%	72%	64%	0	0	0



GTU Belt Operations

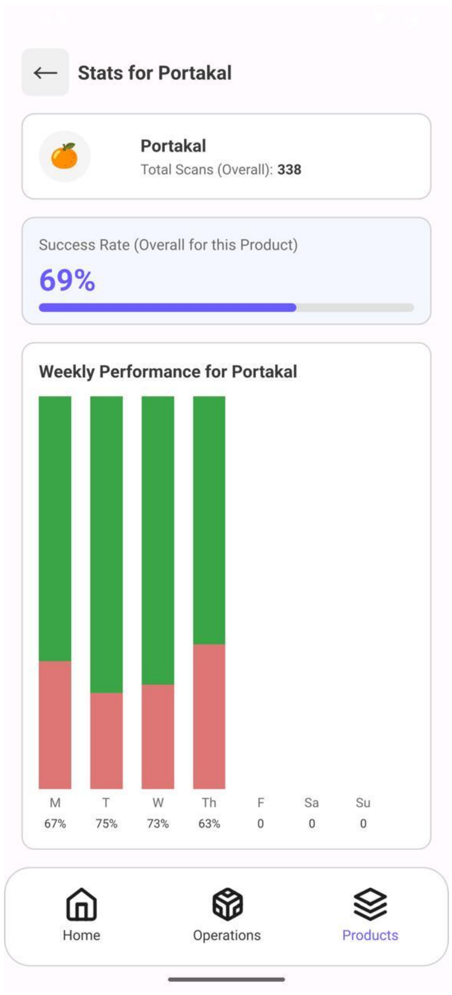
ID	11111111-1111-1111-1111-111111111111
Runtime	0d 0h 1m
Status	Connected - Running
Total Scanned	1000
CPU Usage	0.0%
CPU Temp	0.0°C
Memory Usage	0.0%
Belt Direction	FORWARD
Speed	4

Accuracy

Current Accuracy 80%

Change belt accuracy

System


Stats for Portakal

Portakal
Total Scans (Overall): **338**

Success Rate (Overall for this Product)
69%

Weekly Performance for Portakal

4.10 Conclusion

The SDBelt mobile application has evolved into a mature and essential part of the system ecosystem. With its C++ powered backend logic, responsive QML-based front end, and real-time integration with the core SDBelt platform, the app enables full-featured, portable management of production-line quality assurance.

5.Backend Module

5.1 Introduction

The SD Belt Backend Module serves as the central command and control center for the entire SD Belt system. Built using Spring Boot framework with PostgreSQL database, backend application manages bidirectional communication between desktop applications, mobile interfaces, and AI processing modules while keeping system in control and producing statistical analysis.

5.2. System Overview

5.2.1. Goal and Role

The SD Belt Backend operates as the communication hub and data management center. It functions as:

- **Control Center**: Manages and coordinates all system operations
- **Communication Gateway**: Facilitates real-time bidirectional data exchange
- **Data Processing Engine**: Generates statistics and analytics
- **Security Manager**: Implements user authentication and authorization
- **System Controller**: Provides real-time system status manipulation capabilities

5.2.2. Core Functionality

The backend application have four primary operational areas:

Real-Time Communication Management

- Bidirectional data streaming with desktop and mobile applications
- Command execution and status updates

Statistical Analysis and Reporting

- Real-time generation of quality control statistics
- Historical trend analysis based on scan results
- Comprehensive reporting capabilities for management oversight

System Status Control

- Dynamic handling of conveyor belt operations
- Real-time system parameter adjustments
- Emergency stop and safety protocol implementation

User Management and Security

- Secure authentication and authorization systems

5.3. Technical Architecture

5.3.1 Technology Stack

Framework and Language

- Spring Boot (Java-based framework)
- Java programming language

Database Management

- PostgreSQL relational database
- Advanced query optimization

Communication Protocols

- RESTful APIs for standard operations
- HTTP/HTTPS for secure data transmission

5.3.2. System Architecture Design

Multi-Tier Architecture The backend implements a architecture consisting of:

1. **Presentation Layer**: API endpoints and WebSocket connections
2. **Business Logic Layer**: Core processing and system control logic
3. **Data Access Layer**: Database operations and data management

5.4. Communication and Integration

5.4.1 Bidirectional Communication

Desktop Application Integration

- Comprehensive analytics dashboard data
- System control command processing

Mobile Application Connectivity

- Remote system monitoring capabilities
- Mobile control command execution

AI Module Integration

- Continuous data exchange for processing
- Real-time result integration and analysis
- Machine learning model performance monitoring
- Automated quality assessment feedback loops

5.4.2 API Design and Implementation

RESTful API Structure

- CRUD operations for all system entities
- Standardized response formats and error handling
- Comprehensive API documentation and versioning

WebSocket Implementation

- Real-time data streaming capabilities
- Low-latency communication protocols
- Automatic reconnection and error recovery

Control Command Interface

```
System Control Endpoints:
- /api/v1/system/stop - Emergency system shutdown
- /api/v1/system/start - System activation
- /api/v1/system/speed/{value} - Conveyor speed adjustment
- /api/v1/system/status - Current system status
- /api/v1/system/configure - System parameter configuration
```

5.5. Security and Authentication Framework

5.5.1 User Authentication System

Multi-Level Security Implementation

- Spring Security framework integration
- Password encryption and security policies

5.6. System Control and Manipulation Capabilities

5.6.1 Real-Time System Control

Command Center

- Live system status monitoring and display
- Real-time parameter adjustment capabilities
- Emergency response and safety protocol execution

Control Interface Features

- Conveyor belt speed regulation
- Quality threshold adjustments

5.6.2 Status Management System

Dynamic Status Tracking

- Real-time system health monitoring
- Component status visualization
- Performance metric tracking
- Predictive maintenance alerts

Configuration Management

- System parameter optimization
- Quality control standard adjustments
- User preference and profile management

5.7. Challenges Faced and Solutions Implemented

5.7.1 Technical Challenges

Real-Time Data Processing and Streaming

***Challenge*:** Managing high-volume, real-time data from AI module and hardware sensors while maintaining low latency and system responsiveness across desktop and mobile platforms.

***Solution*:** Implemented WebSocket and HTTP based communication architecture with message queuing systems.

System Integration and Synchronization

***Challenge*:** Maintaining data consistency and synchronization across multiple modules while handling network latency, connection failures.

***Solution*:** Implemented eventual consistency patterns with conflict resolution mechanisms.

5.8. Conclusion

The SD Belt Backend Module represents a solution that solves the requirements of modern quality control systems.

The system's ability to serve as a central control center while maintaining real-time bidirectional communication with desktop, mobile, and AI modules, combined with its powerful statistical analysis and system manipulation capabilities, makes it an essential component.

The implementation of Spring Boot framework with PostgreSQL database ensures that the SD Belt Backend Module delivers immediate operational benefits.

5.9 Test Cases for Backend Module

Feature	Purpose	Test Input / Action	Expected Result	Result
User Authentication	Verify login and token creation	Send valid username and password to `/api/v1/login`	API returns status 200 with a valid token in response	**PASSED**
Invalid Login Handling	Prevent login with incorrect credentials	Send wrong password to `/api/v1/login`	API returns status 401 with error message "Invalid credentials"	**PASSED**
Product Scan Insertion	Ensure product scandefect info is correctly stored	Send POST request with product scan (timestamp, defect flag etc.)	Data is stored in database's corresponding table	**PASSED**
Database Constraint Check	Validate rejection of invalid data formats	Send malformed data (e.g., missing `productResult`)	API returns 400 Bad Request; no insertion into DB	**PASSED**
Command Handling	Confirm backend processes control commands	Send an action for controlling the belt	Command is forwarded to hardware module and acknowledged	**PASSED**
Invalid API Endpoint	Test server response to invalid route	Send GET to a not existing endpoint	Returns 404 Not Found with a proper error message	**PASSED**
Unauthorized Access	Only authorized users can invoke endpoints	Send a request to the protected route	Returns 401 Unauthorized with a proper error message	**PASSED**

List Products	All products are listed	Send list products request	Returns list of products	**PASSED**