Varroa Detection Using Deep Learning: An Embedded Real-Time Detection System for Beekeeping

Ergi Mira

001252261

BEng (Hons) Software Engineering

Supervisors:

Dr Markus Wolf, Dr Elena Irena Popa

May 5, 2025

Abstract

The parasitic *Varroa destructor* mite remains a significant threat to honeybee populations worldwide. Traditional detection methods suffer from inconsistency, labour intensity and delayed feedback. This project examined the design and deployment of a lightweight, real-time object detection pipeline for identifying *Varroa* mites in hive environments by integrating deep learning with edge AI hardware.

The system utilised the YOLOv8 architecture for its real-time performance and detection accuracy. A modified variant, YOLOv8n-SO, was developed by removing the P5 downsampling layer to enhance small object sensitivity. Models were trained on a custom dataset of >15,000 annotated images. The best-performing model, YOLOv8n, achieved an mAP@0.5 of 89.8% and 86.1% precision, with real-time inference on both Hailo-8L and Raspberry Pi 5 platforms. YOLOv8n-SO showed weaker results due to resolution sensitivity and convergence issues.

A web-based dashboard using JavaScript, HTML and Flask was developed to visualise detection results and enable user interaction. The project also addressed legal and ethical considerations of embedded AI for environmental monitoring.

This work contributes to precision apiculture by bridging the gap between laboratory-trained AI and field-deployable systems, demonstrating that real-time deep learning inference can be effectively implemented at the hive level without compromising accuracy.

Contents

1 Introduction **8**
- 1.1 Background and Motivation . . . 8
- 1.2 Problem Statement . . . 9
- 1.3 Research Objectives . . . 9
- 1.4 Development Methodology . . . 10
- 1.5 How This Report Is Organised: Roadmap . . . 11

2 Literature Review **12**
- 2.1 Traditional *Varroa* Detection Methods . . . 12
- 2.2 Early Computer Vision Approaches for *Varroa* Detection . . . 13
- 2.3 Deep Learning for Pest Detection in Agriculture . . . 14
- 2.4 Deep Learning Architectures for *Varroa* Detection . . . 15
- 2.5 Small Object Detection Challenges . . . 16
- 2.6 Edge AI in Beekeeping . . . 17
- 2.7 Limitations in Current Research . . . 17
- 2.8 Research Gap and Justification for Project . . . 18

3 Requirements Analysis and Design **19**
- 3.1 Introduction . . . 19
- 3.2 Requirements Analysis . . . 19
  - 3.2.1 Assumptions and Constraints . . . 19
  - 3.2.2 Stakeholder Analysis and Context . . . 20
  - 3.2.3 Functional and Non-Functional Requirements . . . 21
- 3.3 Design . . . 22
  - 3.3.1 System Architecture . . . 23
  - 3.3.2 UI/UX Design . . . 25
- 3.4 Conclusions . . . 29

4 Implementation **30**
- 4.1 Introduction . . . 30
- 4.2 Dataset Acquisition and Preparation . . . 30
  - 4.2.1 Initial *Varroa* Dataset (436 Images) . . . 32
  - 4.2.2 Diverse *Varroa* Dataset (381 Images) . . . 32
  - 4.2.3 High-resolution *Varroa* Dataset (6,611 Images) . . . 32
  - 4.2.4 Large *Varroa* Dataset (15,000 Images) . . . 33
  - 4.2.5 Dataset Integration and Preprocessing . . . 33
  - 4.2.6 Implementation constraints . . . 35

4.3 Model Implementation . . . 37
4.3.1 YOLOv8n Architecture . . . 37
4.3.2 Model Configuration and Architectural Variants . . . 38
4.3.3 Implementation Framework . . . 39
4.3.4 YOLOv8n-SO Architecture . . . 39
4.4 Hailo 8L Platform Integration, Deployment and Real-Time Operation . . . 41
4.5 User Interface Development and Implementation . . . 42
4.5.1 Frontend Technical Implementation . . . 43
4.5.2 Backend Integration Implementation . . . 45
4.6 Conclusion . . . 47

5 Testing and Experiments **48**
5.1 Introduction . . . 48
5.2 Testing . . . 48
5.2.1 Unit Testing . . . 48
5.2.2 Integration Testing . . . 49
5.2.3 Performance Testing . . . 49
5.2.4 Performance Analysis . . . 50
5.2.5 Test Coverage Summary Aligned to Requirements . . . 50
5.3 Experiments . . . 50
5.3.1 Experimental Framework and Rationale . . . 50
5.3.2 Hyperparameter Optimisation Strategy . . . 51
5.3.3 Augmentation and Regularisation Studies . . . 51
5.4 Conclusion . . . 52

6 Results and Evaluation **53**
6.1 Introduction . . . 53
6.2 Evaluation Criteria and Methodology . . . 53
6.3 Confusion Matrix Analysis . . . 54
6.4 F1-Confidence Comparison . . . 55
6.5 Qualitative Detection Comparison . . . 56
6.6 Precision-Recall Analysis . . . 56
6.7 Precision and Recall vs Confidence . . . 57
6.8 Training Loss and Optimisation Behaviour . . . 58
6.9 Conclusion . . . 59

7 Legal, Social, Ethical and Professional Issues **60**

8 Conclusion and Future Work **62**
8.1 Summary of Achievements . . . 62
8.2 Contributions . . . 62
8.3 Future Work . . . 63

A UML and Additional Diagrams **65**

B Code Samples and Configuration Details **68**
B.1 YOLOv8n Training Pipeline Overview . . . 68
B.2 YOLOv8n-SO Architecture Modification Process . . . 69
B.3 Hailo Compilation and Deployment Algorithms . . . 70

B.4 Frontend Development Algorithms 71
B.5 Backend Algorithms 72
B.5.1 Backend API Endpoints 72
B.5.2 Data Processing Algorithms 73
B.5.3 Time Series Algorithms 74
B.5.4 Email Notification System Algorithms 75

C Additional Results and Testing Data **76**
C.1 Unit Testing Outputs 76
C.2 Integration Testing Outputs 77
C.3 Performance Testing Outputs 78

D Dashboard Screenshots **79**

List of Figures

3.1 Use Case Diagram: Mapping Stakeholder Roles to System Functions in the *Varroa* Detection System 20
3.2 System Overview Animation. 23
3.3 Component Interaction Diagram showing data flow between hardware, backend services and the dashboard UI. 24
3.4 UML Class Diagram of Dashboard Component Structure showing the relationship between UI components, controllers and data store. 26
3.5 Risk State Transition Diagram showing threshold-based classification of colony health status. 27
3.6 Database Schema for the Bee Colony Health Monitoring System showing relationships between tables. 28

4.1 Representative sample images from each dataset showing bees and *Varroa* mites with bounding box annotations: (a) Bee Detection dataset Alice (2025*a*), (b) Beehive Detection dataset Bolo (2025), (c) High-resolution Beehive dataset Alice (2025*b*) and (d) Large Beehive dataset Bee (2025). 31
4.2 Example: A mixture of detection bounding boxes and segmentation-based annotations 33
4.3 Examples of annotation corrections: (a) Original annotations with missing Bee and *Varroa* mite labels, (b) Corrected annotations with accurately bounded *Varroa* mites [red] and bees [green]. 34
4.4 Extensively magnified view of mites used in training data, differing from natural proportions.Bee (2025) 36
4.5 The bi-directional feature fusion neck module of YOLOv8n-SO with feedback loop. 41
4.6 Live detection view showing bounding boxes for detected bees (green) and *Varroa* mites (blue), including confidence scores and frame rate overlay. . 42

6.1 Normalised Confusion Matrices for YOLOv8n and YOLOv8n-SO. 55
6.2 F1-Score across Confidence Thresholds. 55
6.3 Sample detections: Left = YOLOv8n; Right = YOLOv8n-SO. 56
6.4 Precision-Recall Curves for both models. 57
6.5 Precision and Recall vs Confidence. 58
6.6 Loss curves and metric trends for both models. 59

A.1 Detection Process Flow Diagram illustrating the sequence from capture to processing and dashboard update. 65
A.2 Sequence Diagram showing the interaction between system components during detection sessions. 66

A.3 YOLOv8 architecture. 67
A.4 High-level system architecture showing the layered pipeline from camera input to inference, server orchestration and user dashboard. 67

C.1 Example unit test for the `/get_stats` endpoint. 76
C.2 Backend unit test results showing 80% route coverage and 100% for integration components. 77
C.3 JavaScript unit test results confirming UI component responsiveness. . . 77
C.4 Mocked integration test for detection workflow. 77
C.5 SQL query extracting session-wise FPS metrics from `bee_health.db`. . . 78
C.6 Performance metrics across five sessions. 78

D.1 Dashboard HTML structure showing the header, control panel and status overview section. 79
D.2 Mobile view of the dashboard on a smartphone. 79
D.3 Styling of detection metrics with color-coded risk levels. 80
D.4 Animated transition of metric values using JavaScript step functions. . . 81
D.5 The detection summary email notification. 81

List of Tables

3.1 Stakeholder Requirements and System Response 21
3.2 Hardware Components for Training and Deployment 25

4.1 Comparison of Datasets Used in This Study 31
4.2 Image Preprocessing Steps Applied to the Datasets 34
4.3 Key Challenges in Dataset Preparation and Mitigation Strategies 37
4.4 Architectural Comparison of YOLOv8n and YOLOv8n-SO Configurations 39

5.1 Minimal Summary of Implemented Tests and Requirements 50
5.2 Hyperparameter configuration used during extended experiments. 51

6.1 Summary of Key Evaluation Metrics (already included in results) 59

C.1 Session Performance Metrics Summary 78

Chapter 1

Introduction

1.1 Background and Motivation

Honeybee populations have continued to decline, a trend with serious consequences for global biodiversity and agricultural productivity. One of the main factors behind this decline is the parasitic mite *Varroa destructor*. This ectoparasite drains important fat body reserves from bees and transmits deadly viral pathogens, such as Deformed Wing Virus (DWV) and Chronic Bee Paralysis Virus (CBPV), speeding up colony collapse Garc'ıa-Vicente et al. (2024). Despite many years of progress in beekeeping methods, *Varroa* monitoring still relies mainly on manual approaches like sticky board inspections and sugar roll testing. These methods are labour intensive and invasive, and often produce unreliable results, with infestation levels sometimes underreported by up to 30% in poor conditions Hall et al. (2023); Divas'on et al. (2024).

The growing need for sustainable apiculture highlights the importance of integrating artificial intelligence (AI) into routine hive monitoring. Early attempts to automate mite detection using traditional machine learning models proved that vision-based classification was possible, but they showed high false-positive rates and poor adaptability across different hive environments Schurischuster (2018); Bilik et al. (2021). More recent research using deep learning methods, especially Convolutional Neural Networks (CNNs) and single-stage detectors such as YOLO, has improved detection accuracy. However, these systems were usually optimised for controlled laboratory conditions and often failed when deployed in the field due to high energy demands and difficulties in handling visual clutter, occlusion, and changes in lighting common in natural hive settings Bringas et al. (2023); Liu et al. (2023); Wang and Zhang (2024).

In addition, the small size and natural camouflage of the *Varroa* mite present a typical small object detection challenge, which remains difficult in computer vision Bringas et al. (2023). Studies suggest that techniques like increasing spatial resolution or applying attention-based modules may help improve detection precision, although these methods are rarely tested in embedded or restricted environments like beehives Liu et al. (2023); Divas'on et al. (2024). Therefore, there is an urgent need for compact, deployable AI systems that combine accurate model performance with efficient hardware usage, while addressing the specific biological and ecological aspects of *Varroa* infestation.

1.2 Problem Statement

The *Varroa* detection field faces three main challenges. First, current datasets are often small, lack annotation variety and do not reflect the real-world variability of hive environments. This situation causes overfitting and weak generalisation, especially in complex hive scenes Schurischuster (2018); Bringas et al. (2023). Second, although modern detectors such as YOLOv5 and YOLOv8 achieve high accuracy under ideal conditions, their architectural complexity often prevents them from being easily exported to low-power edge devices like the Raspberry Pi or Hailo-8, which are more practical for continuous hive monitoring Calvo et al. (2024); Lee et al. (2024); Krispin-Avraham et al. (2024). Third, most detection pipelines are evaluated mainly on metrics such as mean Average Precision (mAP), whereas factors like deployment feasibility, real-time responsiveness and user interaction are often overlooked.

In addition, efforts to adapt high-performing networks by removing downsampling layers or simplifying head modules frequently lead to unexpected drops in performance. For instance, removing the P5 downsampling layer from YOLOv8n (resulting in YOLOv8n-SO) aimed to improve the detection of small objects, but instead caused high training variance and poor generalisation when tested on magnified mites. This outcome supported earlier concerns raised in small object detection research Liu et al. (2023); Feng et al. (2024).

1.3 Research Objectives

The aim of this project was to close the gap between high-accuracy AI models and practical field deployment by creating and evaluating an embedded, real-time *Varroa* mite detection system. The main research objectives were:

1. To create a custom dataset, expanded with synthetic examples, in order to address class imbalance and better reflect real hive conditions, including occlusion, motion blur and changes in lighting. This method followed established best practices in pest detection dataset preparation, as discussed in Wang and Zhang (2024); Divas'on et al. (2024).

2. To design and assess two object detection architectures, specifically YOLOv8n and a modified version optimised for small objects, YOLOv8n-SO. The aim was to evaluate performance trade-offs in terms of detection accuracy, convergence behaviour and architectural sensitivity in small-object scenarios Bringas et al. (2023); Liu et al. (2023).

3. To benchmark the models on embedded AI hardware platforms, specifically the Hailo-8 AI accelerator and Raspberry Pi 5, measuring inference speed, energy usage and real-time performance in restricted environments Krispin-Avraham et al. (2024); Fezari and Al-Dahoud (2023); Longbottom (2019).

4. To develop a full-stack dashboard interface, supporting real-time alerting, inference visualisation and user interaction. The interface was built using lightweight web technologies to ensure accessibility for technical users and practitioners (e.g. beekeepers).

5. To evaluate the full detection system in operational conditions, using quantitative metrics and qualitative analysis. This included identifying common model failures, such as false positives or missed detections due to occlusion, and comparing outcomes to results reported in existing *Varroa* detection studies.

1.4 Development Methodology

A hybrid methodology was employed to accommodate the experimental nature of deep learning model development and the practical constraints of embedded system deployment. This dual track approach enabled a combined workflow, where architectural choices and performance limitations were refined through real-world field testing and empirical results. Rather than following a strict linear development path, the methodology applied Agile-inspired principles to maintain flexibility in response to emerging needs and unexpected implementation challenges Kumar and Bhatia (2012); Krispin-Avraham et al. (2024).

The system development process was divided into four connected strands: dataset construction, model training, hardware deployment and user interface integration. Each strand was guided by methodological choices informed by earlier studies in small object detection, embedded AI and environmental monitoring technologies Bringas et al. (2023); Liu et al. (2023); Ajay et al. (2025).

Dataset preparation, explained in Chapter 4.2, was based on progressive curation and augmentation techniques. A mixture of public and custom-acquired datasets was annotated and augmented using Roboflow. Augmentation techniques, such as rotation, blur and occlusion, were applied to improve small object detection under hive-like variability. These practices followed well-established methods to improve generalisation in object detection models trained on noisy and class-imbalanced datasets Liu et al. (2023); Ajay et al. (2025).

Model training followed an experimental comparative approach. Two architectures, YOLOv8n and YOLOv8n-SO, were developed to compare baseline performance with improved small object sensitivity. YOLOv8n-SO incorporated changes such as the removal of the P5 layer and the addition of CBAM attention modules, based on principles outlined in Feng et al. (2024), whereas YOLOv8n served as a robust, general purpose benchmark (see Section 4.3). Training and evaluation were conducted using the Ultralytics pipeline.

To assess hardware viability, a hardware-aware deployment process was established, further described in Section 4.4. This involved quantisation-aware compilation of ONNX models into Hailo-specific `.hef` binaries, calibrated using real-world images to minimise accuracy loss during quantisation. Benchmark metrics frame latency, frames per second (FPS), and power consumption were recorded to confirm suitability for real-time detection on Raspberry Pi 5 with Hailo-8L acceleration, following evaluation practices from Krispin-Avraham et al. (2024); Ilieva et al. (2004).

For user interaction, a modular Flask-based dashboard was developed to provide live detection visualisation, system control and real-time alerts. The dashboard architecture, discussed in Section 4.5, was integrated through RESTful application programming interfaces (APIs) and designed to support field deployment using responsive web technologies and asynchronous data polling. The usability features were informed by research on user interface design within the smart agriculture sector Calvo et al. (2024); Narcia-Macias et al. (2024); Ajay et al. (2025).

Evaluation was carried out using both quantitative and qualitative methods, as outlined in Chapter 6. Standard object detection metrics (mAP@0.5, mAP@0.5:0.95, precision, recall, F1 score) were calculated using the test dataset. Further insights were obtained

through precision-recall (PR) curves, confusion matrices, and analysis of failure cases, such as false positives caused by occlusion and magnification errors.

This methodology represents a thorough and realistic research design cycle, highlighting iterative development, embedded system benchmarking and user centred integration. Its effectiveness is critically reviewed in Chapter 6, where system limitations and deployment successes are examined against real-world operational objectives.

1.5 How This Report Is Organised: Roadmap

This report is structured to guide the reader clearly through the complete development of the *Varroa* detection system, covering context, problem framing, methodological reasoning, system design, implementation, evaluation and reflection. Each chapter builds on the previous one to form a complete account of the project's progression, emphasising technical detail and practical application.

Chapter 1 introduces the background and motivation for the study. It outlines the ecological importance of *Varroa* mite monitoring and places the project within the wider field of precision apiculture. This chapter defines the specific research problem, sets out the research objectives, and describes the methodology.

Chapter 2 provides a critical review of relevant literature. It covers biological aspects of *Varroa* detection, reviews manual and automated detection methods, and discusses advances in computer vision, deep learning and embedded AI that have informed the project's technological choices.

Chapter 3 sets out the system's functional and non-functional requirements. It explains the selection of the detection framework, embedded deployment platforms and user interface components, giving reasons based on practical needs and design limitations.

Chapter 4 describes the complete development of the system. It explains dataset integration and augmentation, the training of two different models and the development of the Flask-based dashboard. This chapter also details the model export, quantisation, and deployment process on the Raspberry Pi 5 and Hailo-8L accelerator.

Chapter 5 details system validation through component testing, integration verification and edge deployment benchmarking. Comparative experiments evaluate architectural variant's performance under operational constraints and dataset challenges.

Chapter 6 presents the results from qualitative and quantitative evaluations. It includes detection accuracy measurements, hardware performance results, failure case discussions and comparisons with related work. This chapter evaluates how well the system met its objectives and identifies areas for improvement.

Chapter 7 discusses the legal, social, ethical and professional considerations linked to deploying AI systems in sensitive ecological settings. It addresses issues of transparency, data management and the ethical design of AI applied to non-human subjects.

Chapter 8 concludes the report by summarising the main contributions, revisiting the research questions, recognising system limitations and suggesting directions for future work.

Chapter 2

Literature Review

The *Varroa destructor* mite continues to pose a serious threat to honeybee populations, playing a major role in colony decline through direct parasitism and the spread of viral pathogens Garc'ıa-Vicente et al. (2024). Although traditional detection methods are widely used, they are becoming less suitable due to their labour-intensive and error-prone characteristics Divas'on et al. (2024). Recent research has proposed AI-based alternatives; however, current studies are divided, either concentrating on model accuracy under controlled conditions Bilik et al. (2021) or examining edge deployment feasibility with limited focus on biological factors Mrozek et al. (2021). This chapter reviews the current progress in *Varroa* detection, aiming to highlight key weaknesses in robustness, generalisation, and practical application. It critically assesses algorithmic developments and deployment methods to support the creation of a fully integrated, field-ready detection system.

2.1 Traditional *Varroa* Detection Methods

Traditional *Varroa* detection methods including sugar roll, alcohol wash and sticky board counts have long been relied upon for field diagnosis due to their accessibility and low resource demands. However, their reliability varies and they are criticised for inconsistency and inefficiency.

The sugar roll test, a non-lethal method, involves collecting approximately 300 bees, coating them with powdered sugar and counting mites dislodged by agitation. Although non-destructive, it underreports mite levels in high humidity or with poor agitation. Its sensitivity can fall below 80% compared with alcohol wash methods Divas'on et al. (2024).

The alcohol wash, which submerges bees in ethanol to detach mites, achieves higher sensitivity rates between 90–95%, depending on agitation time and container design Hall et al. (2023). However, it kills the sampled bees, making it less suitable for repeated use, especially in small colonies. Despite its invasiveness, it remains a "gold standard" in research for its consistency Bjerge et al. (2019).

Sticky board monitoring collects mites that naturally fall onto a glued surface beneath a hive over 24 to 72 hours. Sensitivity ranges from 60% to 85%, influenced by hive activity, grooming behaviour and temperature Garc'ıa-Vicente et al. (2024). For instance, Garc'ıa-Vicente et al. (2024) reported mite drops failed to detect infestations in colonies with high

hygienic behaviour, underestimating by up to 30%. Conversely, Bjerge et al. (2019) found sticky boards correlated well with brood infestation under stable conditions.

Despite these flaws, manual methods still appear in validation pipelines. For example, Divas'on et al. (2024) used sticky board counts to benchmark a deep learning model while acknowledging their tendency to underestimate infestation in dense hives.

Manual techniques persist due to low costs (under €2 per hive), simple training and broad regulatory acceptance. However, growing demands for non-invasive, real-time, frequent monitoring are driving a move towards automated systems. Operator-dependent errors exceeding 20% have also been noted Hall et al. (2023), highlighting the need for more consistent digital solutions.

In summary, although traditional methods shaped *Varroa* management, their inconsistency, ethical concerns and poor scalability emphasise the need for AI-based approaches in modern apiculture.

2.2 Early Computer Vision Approaches for *Varroa* Detection

Before the introduction of deep learning, early attempts to automate *Varroa* detection used handcrafted image features alongside classical machine learning models such as Support Vector Machines (SVM), Random Forests (RF) and k-Nearest Neighbours (k-NN). These approaches aimed to classify bees or mite regions by extracting geometric, texture or colour descriptors from high-resolution images Schurischuster (2018); Bilik et al. (2021). Although they marked an important step towards automation, they depended heavily on manually tuned parameters and often failed to generalise beyond controlled datasets.

For instance, Schurischuster (2018) developed a monitoring system that collected over 13,000 manually labelled bee images using a custom tunnel-mounted camera. Their CNN-based approach achieved 94.4% classification accuracy for healthy bees and 85.5% for infected ones. However, traditional machine learning models performed significantly worse, showing a 12% to 15% reduction in F1 score. These findings highlight the limitations of handcrafted pipelines when dealing with subtle parasitic features on complex biological surfaces.

Sensor fusion approaches were also explored, combining gas detection or vibration analysis with classification models to infer *Varroa* presence indirectly Hall et al. (2023); Konig (2022). In Hall et al. (2023), 2D Fourier Transform features extracted from vibration signals were used with traditional machine learning classifiers. Although detection accuracy exceeded 80% in laboratory settings, these systems lacked scalability and real-world validation. Similarly, Konig (2021) trialled a gas-based "bee nose" system and reported promising initial results. However, both sensor-based approaches had low calibration sensitivity and poor robustness under variable hive conditions.

A major drawback across early computer vision and sensor fusion studies was the lack of cross-hive validation and limited data diversity. For example, Bilik et al. (2021) tested a custom object detection system on only 600 annotated images, reporting an F1 score of 0.874 for identifying infected bees. Nonetheless, actual mite detection was less successful,

with an F1 score of 0.714. These results were achieved under ideal conditions and do not account for real-world variability such as lighting changes, motion blur or bee movement.

Over-engineering was another common issue. Several pipelines applied multiple stages of preprocessing, such as histogram equalisation, colour filtering and region proposal refinement, yet failed to outperform simple CNN baselines. In Schurischuster (2018), despite adding noise filters and morphological operations, the handcrafted pipeline was abandoned due to scalability problems and low precision during live testing.

Importantly, none of the early studies achieved successful field deployment beyond static laboratory environments. Models lacked resilience to occlusion, motion artifacts and inconsistent backgrounds common in real beehive footage. This inability to translate laboratory success into practical use created a major technological gap, reinforcing the need for more adaptable and robust deep learning solutions.

2.3 Deep Learning for Pest Detection in Agriculture

The use of deep learning (DL) in agricultural pest detection has greatly improved accuracy, automation and feature representation. Particularly, Convolutional Neural Networks (CNNs) have surpassed classical methods by learning hierarchical spatial features directly from raw images, removing the need for manual feature extraction. In areas such as plant disease classification, livestock condition assessment and insect detection, models like ResNet, InceptionV3 and EfficientNet have achieved accuracies above 90% on curated datasets Berkaya et al. (2021); Sharma et al. (2024).

Transfer learning has been crucial in improving performance when training data are limited. For example, Sharma et al. (2024) fine-tuned ResNet50 and InceptionV3 on a *Varroa* detection dataset. The best model, using CLAHE-enhanced images, achieved 94.3% accuracy, outperforming scratch-trained CNNs by over 6%. Similarly, Berkaya et al. (2021) combined CNN embeddings with an SVM classifier for colony condition classification, achieving an F1 score of 0.91, showing the value of hybrid models in structured biological domains.

Despite these gains, several agricultural DL models rely heavily on ideal laboratory settings. Datasets are often preprocessed, balanced and collected under fixed lighting, leading to generalisation gaps when models face real-world challenges like motion blur, occlusion or lighting variation. For instance, Wang and Zhang (2024) used a U-Net model to segment mite-infested bees and track acaricide exposure but reported major performance losses when moving from controlled to natural hive environments.

Another key issue is the trade-off between model complexity and deployability. High-performing architectures such as EfficientNetB4, especially when combined with NAS-FPN and Focal Loss, achieve excellent mAP scores Varghese and Sambath (2024). However, they require significant computational power, making them unsuitable for edge devices like the Jetson Nano or Hailo-8. This gap reflects a wider mismatch between academic models and practical deployment needs.

Moreover, general agricultural models often struggle with *Varroa*-specific detection tasks. The mite's small size, camouflage, and attachment to living hosts introduce challenges absent from standard datasets. Wang and Zhang (2024) noted frequent misclassification

of mites as debris or shadows due to limited training on diverse real-world examples.

In summary, although DL models have advanced agricultural pest detection, applying them to *Varroa* identification demands improvements in data realism, simpler model design and improved generalisation to cluttered, unstructured and mobile hive environments.

2.4 Deep Learning Architectures for *Varroa* Detection

Recent research into *Varroa* detection has focused on adapting object detection architectures to identify mites in cluttered hive environments. Leading models include YOLOv5, YOLOv8n, YOLOv7-Tiny, Faster R-CNN and Mask R-CNN. Each presents trade-offs between detection accuracy, model size and feasibility for deployment on resource-limited embedded platforms.

The YOLO family has become dominant in *Varroa* detection due to its real-time capabilities. In Calvo et al. (2024), YOLOv5 was deployed on a Jetson Nano, achieving an mAP of 85.6% and inference speeds of up to 9.6 FPS with TensorRT optimisation. However, performance dropped by over 10% mAP in uncontrolled settings, showing sensitivity to lighting and camera position.

More recent work by Lee et al. (2024) introduced YOLOv8n and YOLOv8cls, comparing detection and classification approaches. YOLOv8n achieved 91.2% mAP on annotated bee images, while the YOLOv8 classifier performed slightly worse, suggesting classification is less effective for detecting small, partially occluded *Varroa* mites.

YOLOv7-Tiny was integrated into the IntelliBeeHive system, achieving 89.4% accuracy with inference times under 80 ms Narcia-Macias et al. (2024). However, its precision dropped on low-resolution frames and under changing light, stressing the importance of robust preprocessing or training with augmented data reflecting real-world variability.

Two-stage detectors have also been tested. As mentioned by Kriouile (2022), Faster R-CNN and Mask R-CNN were trained on a dataset with nested annotations. Faster R-CNN reached 92.8% mAP, outperforming Mask R-CNN in precision but with slower inference speeds, often exceeding 200 ms per frame. Significant false positives were noted in crowded bee frames due to contour similarities between mites and bee thoraxes.

Another study done by Kriouile et al. (2024) used transfer learning with Mask R-CNN and ResNet-101, fine-tuned on real-world data. Although achieving 90.1% accuracy, the model was highly sensitive to class imbalance and required extensive annotation, highlighting that high-performing models often depend on curated or synthetic data that fail to represent hive diversity or field conditions.

Deployment studies reveal a gap between offline metrics and real-world performance. Models with high mAP values often show reduced precision in field tests due to motion blur, lighting variation and occlusion. For instance, in Bilik et al. (2021), YOLOv5 achieved an F1 score of 0.874 for infected bee detection, but performance dropped to 0.714 when identifying mites, reflecting the greater difficulty of mite detection.

In conclusion, current DL architectures are promising for *Varroa* detection but clear

challenges remain. YOLO-based models offer excellent speed and acceptable accuracy for embedded deployment, whereas two-stage detectors are more accurate, at the cost of latency and computational demand. Across studies, a performance gap persists between laboratory results and real hive conditions, highlighting the need for domain-specific training, lightweight optimisation and field-oriented validation.

2.5 Small Object Detection Challenges

Detecting *Varroa* mites presents major technical challenges in computer vision due to their small size, occlusion and visual similarity to elements such as bee thoraxes, shadows or hive debris. These features make mites difficult to separate from dense, moving colonies, demanding specialised detection methods.

The authors addressed the small object detection problem by adding a super resolution module to Faster R-CNN Bringas et al. (2023). Their method increased the mAP from 0.784 to 0.849, and improved recall from 0.762 to 0.807, confirming that enhancing image resolution before detection can boost performance. However, they also reported longer inference times, limiting the model's suitability for real-time use in beehives.

A modified YOLOX architecture using Focal Loss and coordinate attention mechanisms was introduced by Liu et al. (2023). The model reached a precision of 0.891, recall of 0.867 and mAP of 0.882 on their *Varroa* dataset. Improvements were linked to the model's better focus on hard-to-detect small objects and its ability to manage class imbalance. In particular, coordinate attention helped the network highlight *Varroa*-shaped regions more effectively, even in cluttered scenes, by embedding location information into the feature maps. However, they noted a decrease in generalisation performance when testing hives with lighting conditions not seen during training.

A pipeline combining DeblurGAN, Faster R-CNN and image tiling was proposed by Divas'on et al. (2024) to address blurring and loss of detail caused by camera shake or bee motion. Although they did not report per class precision or recall, they stressed the importance of tiling to retain fine detail in small targets, especially when using high resolution cameras. The study also highlighted that accuracy for small objects remains far below the benchmarks set for larger objects, pointing to a major challenge in real-world *Varroa* monitoring.

Despite these efforts, a consistent issue across studies is the absence of per object size breakdowns in performance evaluation. Most papers only report overall mAP scores, which can hide poor performance on small classes like *Varroa* mites. Additionally, false positives remain a serious problem. In Bringas et al. (2023), the model often misidentified hive stains and shadows as mites, suggesting that similarity in colour and shape continues to confuse detectors even after architectural improvements.

In summary, techniques such as super resolution, Focal Loss and coordinate attention can improve small object detection for *Varroa* mites. However, these methods still require further testing across diverse hive environments. Without detailed, class-specific metrics and standardised benchmarks, it remains difficult to measure true readiness for real-world deployment.

2.6 Edge AI in Beekeeping

Deploying *Varroa* detection systems directly at the hive demands hardware that can support real-time inference, low latency and low power consumption. Recent studies have investigated various edge AI platforms, including the Jetson Nano and EdgeTPU, for on-device execution of deep learning models in embedded beekeeping environments.

The implementation of a CNN-based *Varroa* detection model using GoogLeNet on a Jetson Nano was described by Wachowicz et al. (2022). The platform consumed 5 watts of power under load and enabled real-time image analysis within the hive. The study demonstrated the potential for low cost and energy efficient deployment, although it highlighted the need for lightweight models due to GPU memory limitations.

Deploying YOLOv7 Tiny on Jetson Nano using TensorRT optimisation was accomplished by Narcia-Macias et al. (2024). The system operated successfully in real time, using a webcam feed connected to the hive box. Although FPS was not reported, the detection loop was tested under continuous video feed, confirming frame by frame object detection and bounding box display.

Research by Mrozek et al. (2021) deployed a custom CNN model created with AutoML on a Google Coral EdgeTPU. The system achieved 400 frames per second while consuming only 2 watts of power, demonstrating excellent efficiency for lightweight vision tasks. Detection performance was validated in hive-like conditions using printed bee images with simulated *Varroa* placements, confirming responsiveness and stability on device.

An examination of real-time three dimensional object detection using a Hailo-8 AI accelerator paired with the InnovizOne LiDAR sensor and the PointPillars architecture was conducted by Krispin-Avraham et al. (2024). Although the study focused on autonomous driving, deployment methods are directly relevant to embedded detection projects.

Similarly, Mika et al. (2023) benchmarked a range of heterogeneous edge platforms, including Jetson Xavier, EdgeTPU, FPGA and Hailo-8. Hailo-8 achieved the highest energy efficiency at 1250 GOPS per watt, compared with 100 to 350 GOPS per watt for Jetson- and FPGA-based platforms.

On the other hand, Turyagyenda et al. (2025) provided a wider review of edge AI in beekeeping, noting that Jetson Nano and Coral devices offer a strong balance between cost, computing capability and integration ease. However, they pointed out that most studies lack standardised benchmarks, making direct platform comparison difficult.

2.7 Limitations in Current Research

Despite increasing research into *Varroa* detection using machine learning and embedded systems, several key limitations persist. One major issue is the shortage of large, diverse and publicly available datasets. For instance, Schurischuster (2018) clearly state that no open source dataset exists for parasite detection in honeybees.

Class imbalance also affects model performance. As noted in Bringas et al. (2023), severe imbalance in training samples causes overfitting towards dominant classes, necessitating augmentation strategies. However, techniques such as GANs and coordinate attention

modules often lack real-world validation, with results mainly reported from laboratory conditions.

Moreover, many deployment studies do not consider practical constraints. Only a few works, including Narcia-Macias et al. (2024) and Calvo et al. (2024), discuss power consumption or environmental robustness.

2.8 Research Gap and Justification for Project

Despite notable progress in *Varroa* detection, current literature lacks a fully integrated solution that combines real-time small object detection, edge deployment and a user friendly interface. Although YOLO-based models have achieved strong mAP scores, most applications are confined to isolated environments without deployment on resource limited hardware or visual interaction features.

This project addresses these gaps by developing a system that enables real-time detection on edge hardware and includes a web-based dashboard for ease of use. A modification to the YOLOv8n architecture was initially tested, involving the removal of the P5 down sampling layer to improve small object detection. However, due to the inconsistent *Varroa* sizes in the dataset, the modified model performed poorly and was not retained.

The final system design prioritises lightweight architecture, on device execution and real-time user feedback, addressing a critical gap in current *Varroa* detection research through full system integration and readiness for deployment.

Chapter 3

Requirements Analysis and Design

3.1 Introduction

This chapter presents the detailed requirements analysis and system design for the *Varroa* mite detection and monitoring system. The requirements are divided into functional and non-functional categories, addressing the needs of different stakeholders within the beekeeping sector. The design section describes the system architecture, components and data flow that meet these requirements through a modular approach based on edge computing, using the Raspberry Pi 5 platform and the YOLOv8n deep learning model.

3.2 Requirements Analysis

The system requirements were gathered through a focused review of existing research on *Varroa* detection, as discussed in Chapter 2. Studies on traditional methods Hall et al. (2023); Bjerge et al. (2019); Garc'ıa-Vicente et al. (2024); Divas'on et al. (2024), early computer vision approaches Schurischuster (2018); Bilik et al. (2021) and deep learning-based models such as YOLOv5 and YOLOv8n Calvo et al. (2024); Lee et al. (2024); Narcia-Macias et al. (2024) offered valuable insights into current limitations and deployment challenges. Particular attention was given to real-time performance, edge device compatibility and resilience in hive environments. These findings were combined to define functional and non functional requirements using the MoSCoW framework, ensuring that the final prototype meets practical field needs and has technical feasibility.

3.2.1 Assumptions and Constraints

Development assumptions and constraints were based on practical factors identified in the reviewed literature. It was assumed that users could provide consistent video input with sufficient lighting, as needed for effective detection by models such as YOLOv8n Lee et al. (2024); Liu et al. (2023). The system is limited by the computational capacity of edge devices like the Raspberry Pi 5 Mrozek et al. (2021); Wachowicz et al. (2022); Narcia-Macias et al. (2024), requiring a lightweight architecture and an optimised inference process. Offline operation, energy efficiency and resilience to environmental variation were also treated as key constraints, as highlighted in earlier deployment studies Turyagyenda

et al. (2025); Mika et al. (2023). These considerations shaped the design scope and ensured alignment with real-world deployment conditions.

3.2.2 Stakeholder Analysis and Context

The *Varroa* mite detection system is designed to support a range of users involved in beekeeping, entomological research and precision agriculture. These stakeholder needs are addressed through specific features in the system architecture, ensuring practical relevance and usability for real-world deployment.

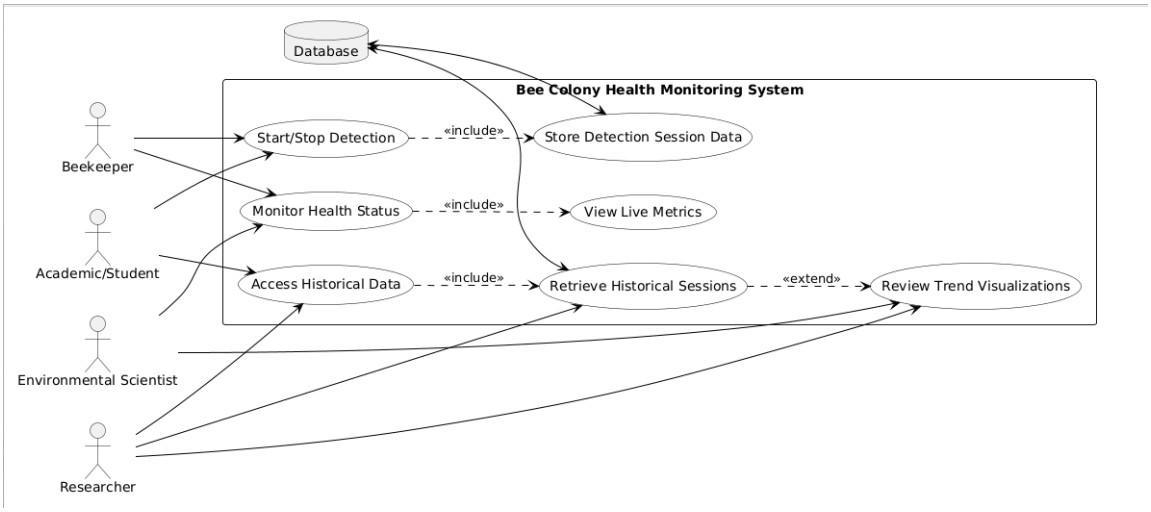

Figure 3.1: Use Case Diagram: Mapping Stakeholder Roles to System Functions in the *Varroa* Detection System

Beekeepers require a non-invasive, real-time method for monitoring mite infestations without harming bees. Traditional methods such as sticky boards or alcohol wash are intrusive and result in bee loss. The proposed system addresses this need by offering live camera-based monitoring without destructive sampling Bjerge et al. (2019).

Researchers benefit from highly accurate, repeatable outputs to track *Varroa*-bee ratios over time. This includes metrics such as infestation percentages and detection counts across frames. The system supports these needs through its integrated infestation ratio calculator and real-time logging dashboard Bilik et al. (2021).

Embedded Systems Engineers seek modular, low power deployments adaptable to various edge hardware environments. The software operates on a Raspberry Pi 5 using the YOLOv8n model, maintaining performance within a 10 watt power budget ideal for embedded AI benchmarking and prototyping Longbottom (2019).

Environmental Scientists prefer bee-friendly methods that minimise ecological disruption. Since the solution avoids direct physical contact with the hive, it aligns with current ethical recommendations for non-invasive monitoring Turyagyenda et al. (2025).

Academics and Students require replicable, modular platforms for experimentation and further development. The system's modular codebase, real-time visualisations and support for live and offline inputs provide flexibility for research and teaching Mika et al. (2023).

Stakeholder	Need	System Response
Beekeepers	Non-invasive, real-time monitoring of mite infestation without harming bees	The system provides live detection from a camera, removing the need for alcohol wash or bee sampling Bjerge et al. (2019)
Researchers	High accuracy, repeatable results for Varroa-bee ratio tracking	The system provides infestation ratios, detection statistics, and graphical summaries Bilik et al. (2021)
Embedded Systems Engineers	Low power, modular system suitable for embedded AI testing	The system runs fully on-device (Raspberry Pi 5 + YOLOv8n) and consumes less than 10 watts Longbottom (2019)
Environmental Scientists	Bee-friendly technology that reduces ecological disturbance	The solution eliminates destructive sampling, supporting ethical monitoring practices Turyagyenda et al. (2025)
Academics and Students	A modular, replicable AI model for research and experimentation	The system's modular code, dashboard visualisations, and .mp4 input compatibility enable flexible research Mika et al. (2023)

Table 3.1: Stakeholder Requirements and System Response

3.2.3 Functional and Non-Functional Requirements

The system requirements have been organised using a prioritisation framework that separates critical baseline features, recommended usability improvements, optional extensions for additional functionality and deferred elements outside the current project scope.

Functional Requirements

The system must support several essential features necessary for its baseline operation. Live camera input integration is required to enable real-time video capture and continuous processing from the hive environment. Object detection functionality must be implemented to identify both bees and *Varroa destructor* mites within the video stream. A dashboard interaction feature must be available, allowing users to start and stop sessions as well as view real-time system status. Additionally, the system must calculate a health status metric by determining the infestation ratio between detected mites and bees.

Beyond the essential features, several enhancements are recommended to improve usability and analytical capabilities. The system should offer real-time logging and session summaries to enable review of detection metrics and session performance. Data storage and persistence mechanisms are required to support historical trend analysis and facili-

tate comparisons across multiple sessions. The inclusion of graphical visualisations, such as histograms and line charts, will assist in identifying anomalies and monitoring trends over time. Furthermore, a risk level classification feature based on infestation ratios should be provided, categorising colony health into intuitive levels: low, moderate, high and critical.

To further increase system flexibility and responsiveness, additional optional features may be considered. An email notification mechanism could be implemented to deliver real-time alerts if infestation levels exceed predefined thresholds. Offline inference compatibility, enabled through support for pre-recorded `.mp4` video inputs, would allow testing and analysis without requiring a live hive environment.

One functional component has been deliberately excluded from the current prototype. Bounding box visualisation within the dashboard has been deferred to future development phases to prioritise critical system operations and preserve inference performance.

Non-Functional Requirements

To ensure reliable and field-ready operation, several non-functional aspects are required. The system must deliver real-time processing with minimal latency, supporting efficient inference through the deployment of YOLOv8n on a Raspberry Pi 5 Longbottom (2019). High accuracy must be maintained, with a mean average precision (mAP50) above 0.80 and a false negative rate below 20 percent, as demonstrated in *Varroa* detection research Bilik et al. (2021); Divas'on et al. (2024). In addition, energy efficiency and thermal stability are critical to prevent performance degradation during prolonged operation Longbottom (2019). The system must offer a usable interface that updates frame rates, detection counts and infestation metrics at over 30 frames per second, ensuring a responsive user experience. Full offline operability is also required, with all processing and data handling conducted locally without any dependence on internet connectivity Mika et al. (2023).

To enhance robustness and facilitate future maintenance, several additional qualities are recommended. The system should support maintainability and scalability, enabling configurable thresholds, straightforward integration of new features and a modular software architecture. Security and data integrity must be addressed through comprehensive exception handling, ensuring resilience against data loss or corruption. Furthermore, fault tolerance and system stability should be incorporated, allowing the system to recover automatically from common errors such as camera disconnections through automated resets or user notifications.

To support broader deployment scenarios, one non-functional enhancement is proposed. Portability and modularity would enable the adaptation of the software to alternative embedded platforms, such as the Jetson Nano or EdgeTPU, facilitating further experimentation and wider scalability in future work.

3.3 Design

The final prototype of the *Varroa* mite detection system consists of a tightly integrated hardware and software stack shaped by iterative experimentation, practical constraints and decisions informed by the literature. Five key design principles guided the development: edge first computing, modularity, resource efficiency, real-time feedback and fault

tolerance, each rooted in the deployment context and stakeholder needs (see Section 3.2). An edge-based hardware setup enabled edge first computing without cloud dependence, which is essential for remote apiary deployment Turyagyenda et al. (2025); Mika et al. (2023). Two models were explored, including YOLOv8n and YOLOv8n-SO, before selecting YOLOv8n for its balance between performance and deployability. The reasoning and outcomes of these evaluations are explained in the following sections. The system was developed using a modular architecture, allowing flexible updates to components such as detection, the user interface and the backend Narcia-Macias et al. (2024). A lightweight web-based user interface was also created to support real-time monitoring and interaction with the detection pipeline. A simple System Overview Animation is shown in Figure 3.2.

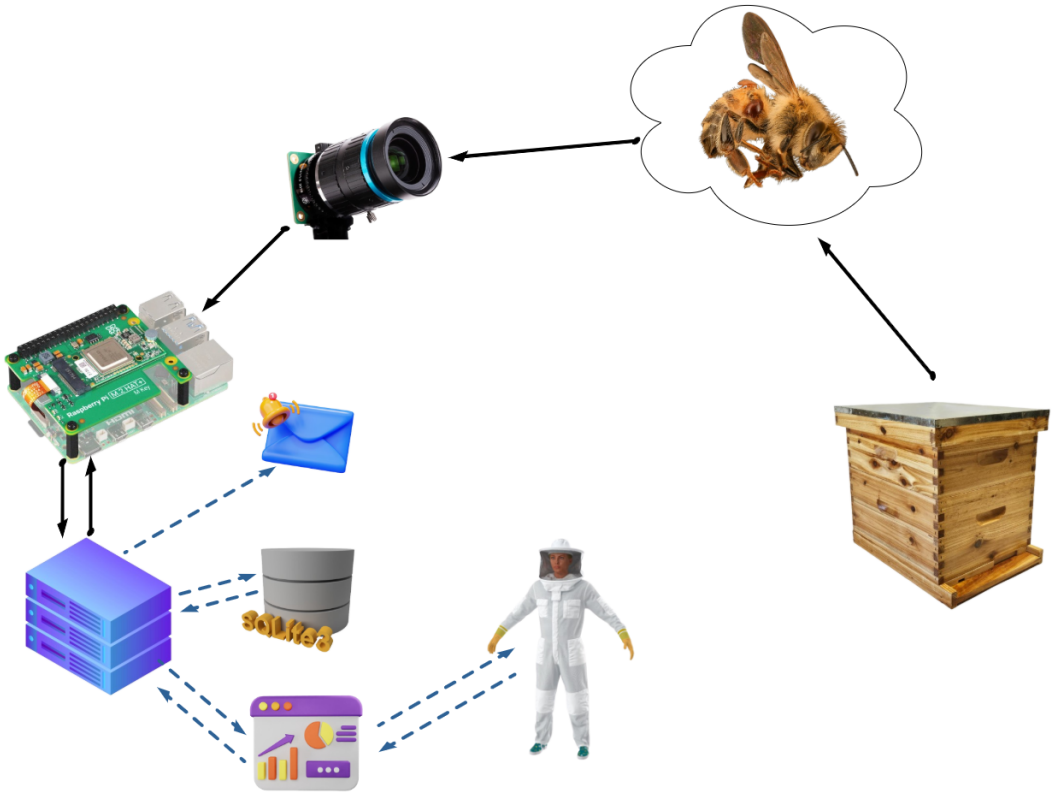

Figure 3.2: System Overview Animation.

3.3.1 System Architecture

The *Varroa* Mite Detection and Monitoring System is built on a modular, scalable architecture optimised for embedded AI deployment. It integrates hardware and software components into a real-time, field-ready detection pipeline deployed on a Raspberry Pi 5 and Hailo-8L AI accelerator. The system is divided into four primary architectural layers, as shown in Appendix A Figure A.4. Each layer is introduced briefly there and described in greater detail in paragraphs below.

Data Acquisition Layer Captures live video using the Raspberry Pi HQ Camera and prepares frames for inference. Camera configuration details are discussed in Subsection 3.3.1.

Inference Layer Executes the YOLOv8n model compiled to a hardware-efficient '.hef'

binary for low-power, real-time inference on the Hailo-8L accelerator. Training and compilation details appear in Subsection 3.3.1, and model structure is described in Subsection 4.3.1.

Backend Server Layer Manages system logic using a Flask server that launches detection sessions, processes model output and serves data to the frontend. Detailed software logic and data handling are provided in Subsection 3.3.1 and Subsection 3.3.2.

Frontend Monitoring Layer Delivers an interactive web-based dashboard that visualises detection metrics for beekeepers. The design and layout are discussed in Subsection 3.3.2.

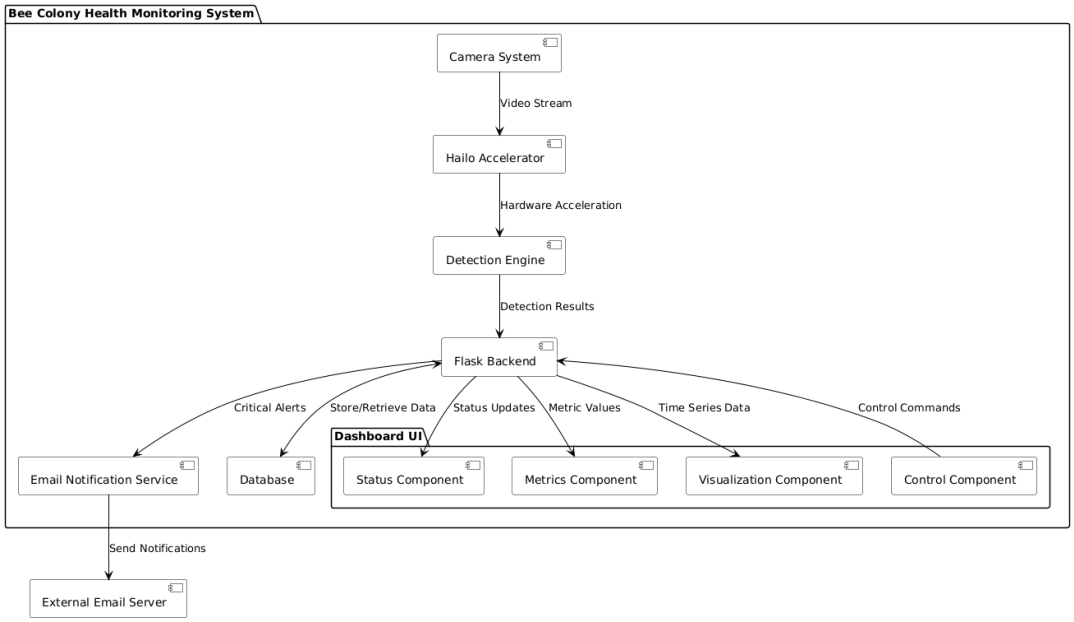

Figure 3.3: Component Interaction Diagram showing data flow between hardware, backend services and the dashboard UI.

Hardware Architecture

The hardware platform consists of two environments: a high-performance training and compilation system and a compact, deployable edge device.

Deployment Setup The deployable system is based on a Raspberry Pi 5 (8GB RAM, Cortex-A76) with a Hailo-8L AI accelerator (13 TOPS, M.2 interface). Video is captured using the Sony IMX477 12.3MP camera via the CSI interface. The system is powered by a 27W USB-C adapter and housed in the official Raspberry Pi case with active cooling.

Training and Compilation YOLOv8n was trained on an NVIDIA RTX 4090 using the Ultralytics framework, and exported to ONNX. Compilation to Hailo's '.hef' format was done on a TITAN RTX using the Hailo Model Zoo Toolkit Krispin-Avraham et al. (2024).

Component	Specification and Role
Raspberry Pi 5	8GB RAM, 2.4GHz Cortex-A76, USB 3.0, PCIe 2.0; hosts backend and API server
Hailo-8L Accelerator	13 TOPS via M.2; executes inference in real time
Pi HQ Camera	Sony IMX477, 12.3MP; captures real-time frames
Power Supply	27W USB-C (5V/5A); powers all Pi components
NVIDIA RTX 4090	24GB VRAM; model training
NVIDIA TITAN RTX	576 Tensor Cores; ONNX to .hef conversion

Table 3.2: Hardware Components for Training and Deployment

Software Architecture

The software stack consists of a modular backend, real-time detection engine, database layer and a responsive frontend. It is designed to support low-latency streaming, fault tolerance and historical data tracking.

Backend System Implemented using Flask, the backend handles detection orchestration, parses model output and exposes REST APIs for frontend communication. Internal process handling and detection control are detailed in Subsection 3.3.2.

Detection Engine The inference pipeline is managed by a subprocess running `detection.py`, which executes the YOLOv8n '.hef' model and emits structured logs. Real-time FPS, object counts and risk classification are derived from these data (see Subsection 3.3.2).

Database and Frontend Parsed outputs are stored in a local SQLite database for post-session analysis. The frontend, built with HTML, CSS and JavaScript, displays live metrics and charts for user interpretation. Layout and visualisation design are discussed in Subsection 3.3.2.

3.3.2 UI/UX Design

The dashboard serves as the primary user interface for monitoring colony health in real time. It provides a clear and accessible overview of key metrics derived from the detection system, supporting informed decision-making in the field. Designed with responsiveness and simplicity in mind, the interface presents detection outcomes, health indicators and system status updates in an intuitive layout. Emphasis is placed on usability and performance, ensuring smooth interaction even under resource-constrained edge deployment conditions.

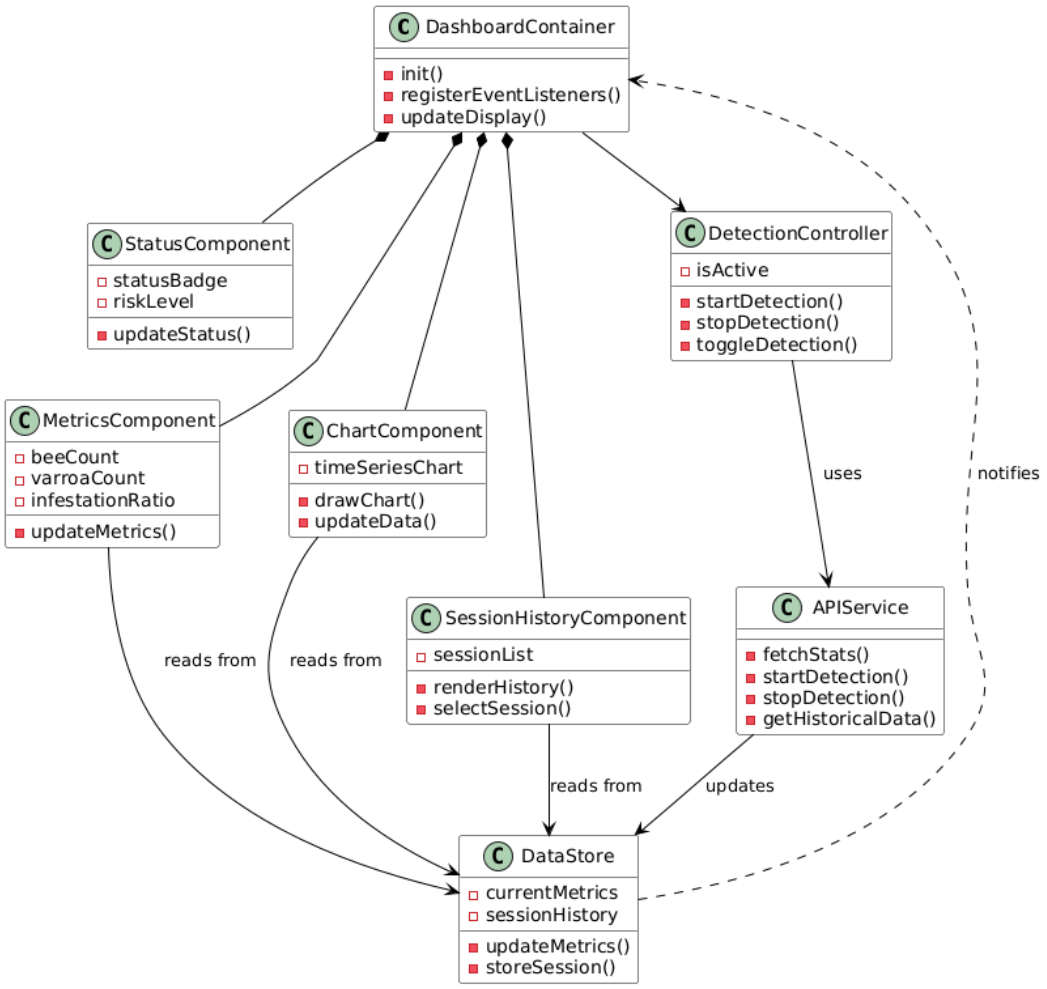

Figure 3.4: UML Class Diagram of Dashboard Component Structure showing the relationship between UI components, controllers and data store.

Real-Time Data Flow and Risk Classification

The dashboard presents real-time colony health metrics from the detection system, allowing beekeepers to monitor their hives continuously and make informed decisions based on current data. To support this, the interface is designed to update itself periodically with the latest statistics, without requiring manual refreshing or user intervention. This approach ensures the user interface remains responsive and accurately reflects the colony's condition in near real time.

Risk classification is carried out according to infestation ratio thresholds, as detailed in Subsection 3.2.3, with visual cues linked to colour-coded indicators to enhance interpretability. The risk categories include low, moderate, high and critical levels, each designed to aid decision-making in field conditions (see Figure 3.5). The system prioritises clarity and urgency through consistent visual encoding, with more severe conditions triggering stronger visual signals.

The design focuses on real-time, edge based decision-making while reducing resource consumption, making it suitable for deployment on lightweight field devices. The system architecture avoids computationally intensive methods such as OpenCV-based video

streaming or WebRTC overlays, instead adopting a modular and resource-efficient approach Narcia-Macias et al. (2024); Mika et al. (2023).

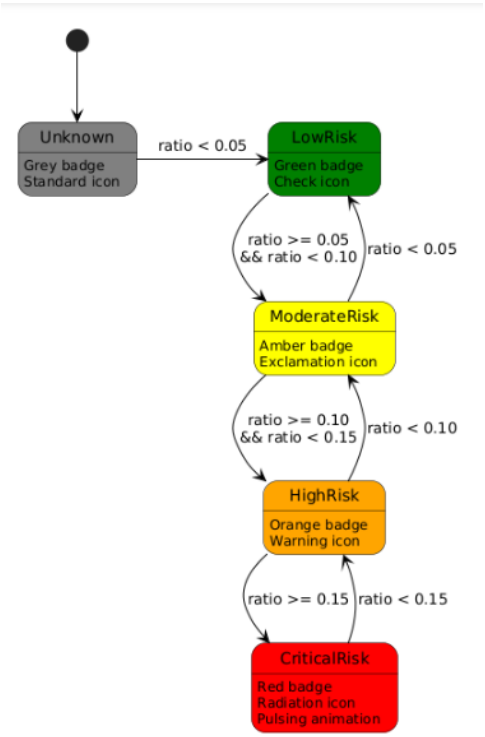

Figure 3.5: Risk State Transition Diagram showing threshold-based classification of colony health status.

Data Pipeline

The data pipeline forms the backbone of the *Varroa* detection system, linking the camera hardware, AI detection model and dashboard interface into a cohesive and functional monitoring architecture. It was designed to minimise latency, maximise interpretability and support the system's real-time decision-making capabilities in field conditions (see Figure A.1 and A.2 in Appendix A)

The pipeline begins with the Raspberry Pi HQ camera, which captures a live video feed of the bee colony. This visual input is processed by a pre-trained object detection model running on an AI accelerator. The model identifies and classifies objects of interest, including bees and *Varroa* mites, in each frame. These detections are parsed and summarised to produce real-time statistics, which are then sent to the dashboard interface for clear, user friendly presentation.

Throughout each monitoring session, key statistics such as bee counts, *Varroa* counts, infestation ratios, frame rates and cumulative frame counts are calculated and visualised. These metrics provide beekeepers with continuous, up-to-date feedback on the colony's condition. The system was designed to handle this data flow efficiently and with minimal hardware requirements, making it suitable for edge deployments on consumer grade devices.

Alongside real-time presentation, the pipeline supports persistent data logging. Metrics from each session are stored in a structured database that records session metadata

and detection outcomes. This allows for future retrieval and historical analysis, enabling beekeepers to monitor infestation trends over time and evaluate the effectiveness of interventions (see Figure 3.6).

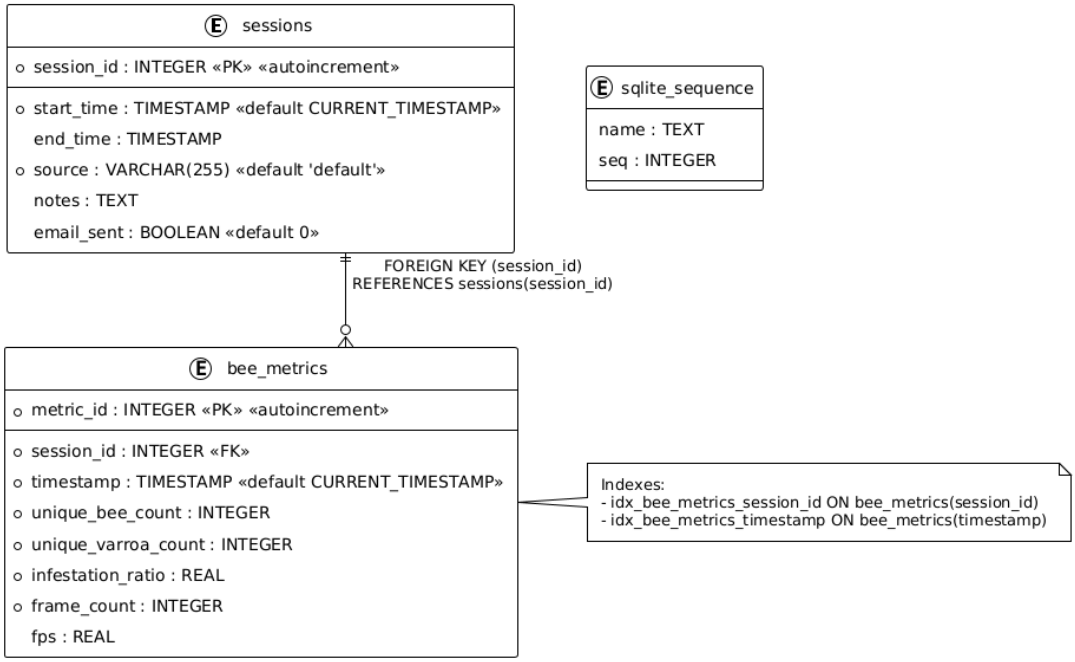

Figure 3.6: Database Schema for the Bee Colony Health Monitoring System showing relationships between tables.

The design of the data pipeline prioritises modularity and future extensibility, as outlined in Subsection 3.2.3. Each stage, from image capture and detection to dashboard visualisation and data storage, is implemented as a discrete component with clearly defined responsibilities. This modular structure simplifies troubleshooting, performance optimisation and future upgrades, enabling adaptation to new hardware or algorithmic improvements with minimal adjustments.

The database schema consists of two main tables with supporting functionality. The `sessions` table tracks monitoring periods with `session_id` as the primary key and stores metadata such as `start_time`, `end_time`, `source` of the video feed, `notes`, and an `email_sent` flag indicating whether post-session notifications were issued.

The `bee_metrics` table stores detection data, using `metric_id` as its primary key and maintaining a parent child relationship with the `sessions` table through a foreign key constraint on `session_id`. Each entry includes a `timestamp`, `unique_bee_count`, `unique_varroa_count`, `infestation_ratio`, `frame_count`, and `fps` (frames per second).

To optimise query performance, the schema includes two indexes: `idx_bee_metrics_session_id` for efficient session based lookups and `idx_bee_metrics_timestamp` for time series analysis. The system also uses SQLite's auto increment functionality via the `sqlite_sequence` table.

This database architecture supports real-time monitoring and advanced post-session analytics, aligning with best practices for environmental monitoring systems. Additionally, the `BeeHealthDatabase` class provides a structured API for database operations,

including session management and metric recording, while maintaining relational integrity Narcia-Macias et al. (2024).

Extended System Features

Beyond its core detection and visualisation functions, the system includes additional features that improve usability, resilience and long-term field applicability. An automated email notification module sends standardised end-of-session reports summarising colony health, enabling asynchronous updates for remote monitoring without the need for continuous user supervision. Stable long-term operation is supported through robust process management that handles recovery from interruptions, prevents resource leaks and allows safe termination of background processes, which is essential for headless edge deployments.

To further assist with interpretation and historical analysis, the system provides advanced visualisations such as time series graphs, histograms and infestation ratio trend lines. These visual tools use intuitive colour-coded indicators to distinguish bee and *Varroa* dynamics over time. In addition, a historical data interface enables users to retrieve summaries of previous sessions and examine specific metrics, supporting retrospective trend analysis and data-driven planning for colony treatments.

3.4 Conclusions

The requirements analysis and design phases have established a strong foundation for the *Varroa* mite detection system. By addressing stakeholder needs, technical constraints and deployment environments, the architecture balances functionality with field usability.

The functional requirements focus on real-time detection, monitoring and alerting, while the non-functional requirements ensure reliability, efficiency, and ease of use. The modular design, covering data acquisition, inference, backend services and frontend visualisation, supports effective operation and future scalability.

Key innovations include the integration of edge AI through the Hailo-8L accelerator with consumer grade hardware like the Raspberry Pi 5, enabling efficient on-device inference. A lightweight, real-time dashboard has been developed for field usability, alongside risk classification thresholds to support beekeeper interventions. The system also provides local data persistence via a structured database, enabling long-term colony health monitoring and trend analysis.

The design meets stakeholder expectations while remaining compatible with real-world beekeeping conditions. By offering non-invasive, real-time monitoring of *Varroa* infestations, it significantly improves on traditional destructive sampling methods.

Chapter 4

Implementation

4.1 Introduction

This chapter describes the full implementation of a real-time *Varroa* mite detection system, based on YOLOv8 object detection, operating on edge hardware. The system is designed to identify *Apis mellifera* (honeybees) and the parasitic *Varroa destructor* mites within beehive environments, using computer vision techniques optimised for resource constrained devices.

The implementation covers four main components:

1. Dataset creation and preprocessing
2. Model architecture selection and optimisation
3. Deployment pipeline for edge hardware
4. Real-time monitoring dashboard

Each component addresses specific challenges related to small object detection, real-time inference and practical deployment in apiary settings.

The aim of the system is to balance detection accuracy with computational efficiency, targeting the Raspberry Pi 5 combined with the Hailo-8L AI accelerator as the deployment platform. This hardware configuration enables real-time inference while maintaining power efficiency for field applications. The chapter concludes by presenting the testing methodologies used to validate functional correctness and performance characteristics.

4.2 Dataset Acquisition and Preparation

The development of an effective bee and *Varroa* mite detection system relies on a diverse and high-quality dataset that accurately represents the target objects under a range of conditions. This section describes the dataset acquisition and preparation process followed in this project. All datasets were sourced from Roboflow Universe, a public repository of annotated image datasets. Multiple datasets were identified, evaluated and processed to optimise the performance of the YOLOv8n model. This process involved

careful dataset selection, image preprocessing, verification and enhancement of annotations, as well as integration into a cohesive training pipeline, aiming to maximise the model's ability to detect bees and the much smaller *Varroa* mites in real-world conditions.

Table 4.1: Comparison of Datasets Used in This Study

Dataset Name	**Total Images**	**Selected Images**	**Resolution Range**	**Bee Labels**	**Varroa Labels**
Bee Detection Alice (2025*a*)	436	400	640×640	93	534
Beehive Detection Bolo (2025)	381	350	640×640	–	–
High-res Beehive Alice (2025*b*)	6,611	1,000	224×224	8,452	17,561
Large Beehive Bee (2025)	17,619	15,000	640×640	78,138	17,543

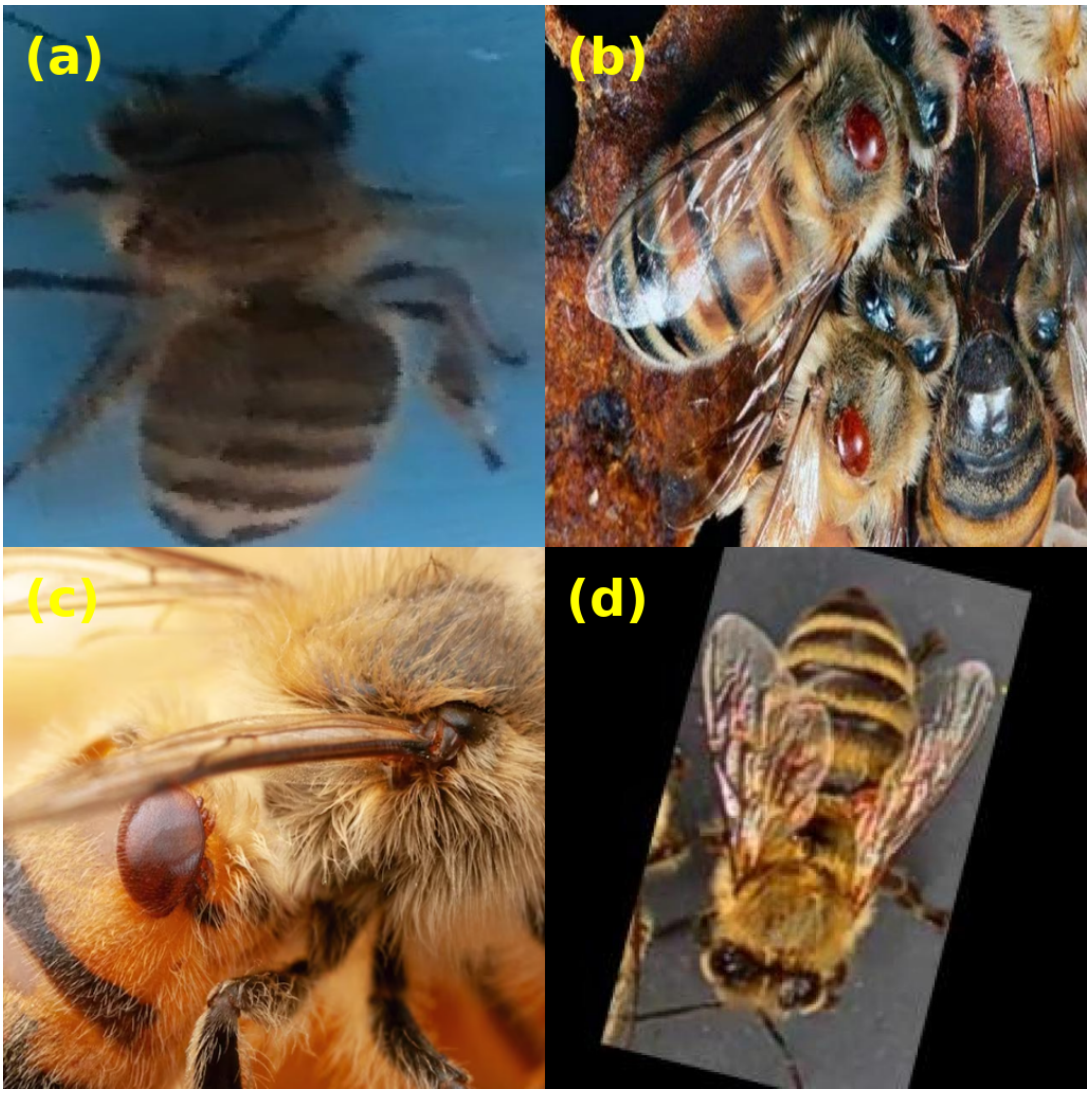

Figure 4.1: Representative sample images from each dataset showing bees and *Varroa* mites with bounding box annotations: (a) Bee Detection dataset Alice (2025*a*), (b) Beehive Detection dataset Bolo (2025), (c) High-resolution Beehive dataset Alice (2025*b*) and (d) Large Beehive dataset Bee (2025).

4.2.1 Initial *Varroa* Dataset (436 Images)

The initial dataset used in this study was the "Bee Detection" collection Alice (2025*a*). It contained 436 annotated images focusing on bees across various environments. Due to its relatively small size, it served as a useful starting point for early model training, allowing rapid iteration and baseline performance establishment. The images provided clear views of bees against diverse backgrounds, offering a sound foundation for initial object detection development. However, the limited sample size meant this dataset alone could not support training a model capable of generalising to diverse real-world conditions. The annotations mainly targeted bees, with some instances of *Varroa* mites, although the labelling quality needed verification and occasional refinement to align fully with the project's detection goals.

4.2.2 Diverse *Varroa* Dataset (381 Images)

The second dataset used in the project was the "Beehive Detection" collection Bolo (2025), containing 381 annotated images. This dataset complemented the first by offering images captured from different angles and lighting conditions. Although modest in size, it was valuable for its diversity in bee postures and contexts, particularly within beehive environments where detection is practically applied. Integrating this dataset broadened the model's ability to recognise bees across varying positions and settings. During evaluation, around 31 duplicate or irrelevant images were identified and removed, resulting in 350 high-quality images for training. The annotations required considerable review, with approximately 20% needing manual correction to ensure accurate representation of bee and *Varroa* mite positions and sizes.

4.2.3 High-resolution *Varroa* Dataset (6,611 Images)

The third dataset used was the "High-resolution Beehive" collection Alice (2025*b*), comprising 6,611 images. This large dataset provided high-quality, detailed imagery, particularly valuable for detecting the minute features of *Varroa* mites. The superior resolution enabled the model to learn subtle visual characteristics that distinguish *Varroa* mites on bee bodies. From the full collection, 1,000 images were carefully selected based on clarity, correct representation of target objects and annotation quality. A limited selection was necessary because the dataset owner had recently updated most annotations to segmentation format, which is incompatible with the YOLOv8n framework that requires bounding box annotations. Figure 4.2 shows an example of segmentation annotations that could not be used directly.

This dataset significantly improved the model's ability to detect *Varroa* mites, which are challenging due to their small size relative to bees and their similar colouring to some bee body parts. The detailed annotations provided valuable information on the precise location and size of bees and *Varroa* mites, although around 15% of the selected annotations required refinement to align with the project's labelling standards.

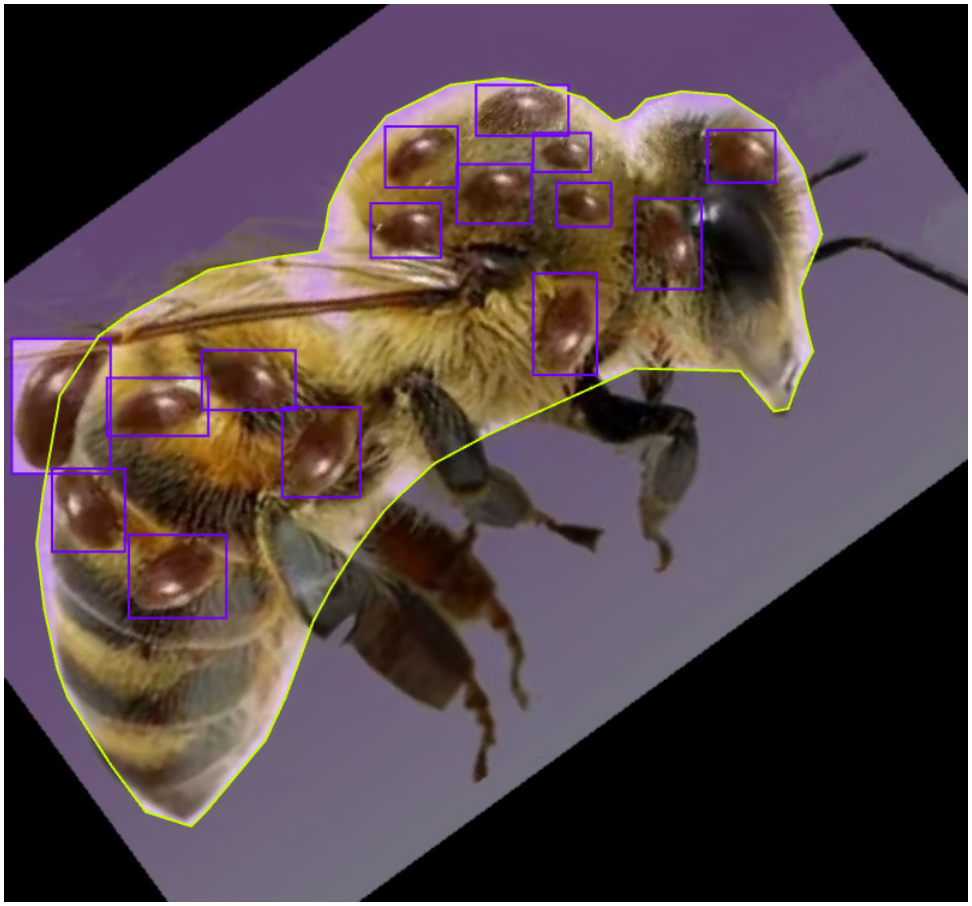

Figure 4.2: Example: A mixture of detection bounding boxes and segmentation-based annotations

4.2.4 Large *Varroa* Dataset (15,000 Images)

The final and largest dataset incorporated into this project was the "Large Beehive" collection Bee (2025), containing around 15,000 annotated images. This extensive dataset provided the volume and diversity needed to train a robust model capable of generalising to a range of real-world conditions. It included images captured under varying lighting, distances, angles and backgrounds, offering a broad representation of potential detection scenarios. Its size made it particularly suited for the final training phase after the model had been initially trained and refined using the smaller datasets.

The wide variety of images helped address overfitting concerns and improved the model's ability to perform consistently across different environments. Although the dataset contained detailed annotations, the large volume required a selective verification process, where a subset of annotations was manually reviewed and corrected where necessary.

4.2.5 Dataset Integration and Preprocessing

The implementation of an effective training strategy required careful integration of the diverse datasets. Instead of using them in isolation, a progressive approach was adopted, where each dataset contributed to the model's development at different stages. Initially, the 436-image dataset Alice (2025*a*) was used for the first training phase to establish

a baseline. The best weights from this stage were then used to initialise training on the 381-image dataset Bolo (2025), applying a form of transfer learning to preserve and extend previously learned features.

Following this incremental training, a curated set of approximately 1,750 images from the first three datasets was compiled. This collection underwent rigorous preprocessing to ensure optimal training conditions. Table 4.2 summarises the preprocessing steps applied.

Table 4.2: Image Preprocessing Steps Applied to the Datasets

Preprocessing Step	**Description and Purpose**
Resizing	All images resized to 640×640 pixels to match YOLOv8's default input dimensions
Normalisation	Brightness and contrast normalisation to reduce the impact of varying lighting conditions
Noise Reduction	Removal of Gaussian blur from noisy images to enhance feature clarity
Image Augmentation	Flipping, rotation and zooming to increase the diversity of the training data
Annotation Correction	Manual refinement of around 300 images with incomplete or inaccurate labels
Format Standardisation	Conversion to YOLO txt format (class, x_center, y_center, width, height)

The annotations required considerable attention. Approximately 300 images contained incomplete or incorrect labels, as shown in Figure 4.3, and were manually corrected using Roboflow annotation tools. Particular emphasis was placed on ensuring accurate bounding boxes for *Varroa* mites, given their small size and detection difficulty. All annotations were standardised to the `YOLO txt` format `(class, x_center, y_center, width, height)` and organised into the appropriate directory structure to maintain compatibility with the `YOLOv8` training pipeline.

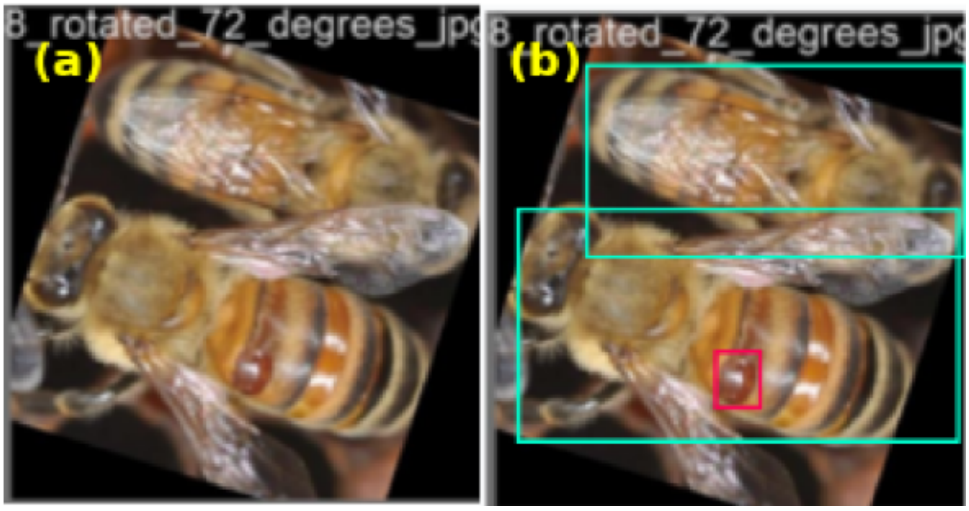

Figure 4.3: Examples of annotation corrections: (a) Original annotations with missing Bee and *Varroa* mite labels, (b) Corrected annotations with accurately bounded *Varroa* mites [red] and bees [green].

The final training phase used the extensive 15,000-image dataset Bee (2025), providing the volume and diversity necessary to develop a robust model. The model trained on the curated 1,750-image dataset served as the starting point, enabling further refinement of detection capabilities while preserving knowledge acquired during earlier training stages.

4.2.6 Implementation constraints

The dataset acquisition and preparation process presented several challenges that required strategic planning. A primary issue was the inconsistent annotation quality across publicly available datasets. Due to the small size of *Varroa* mites, annotations were often imprecise, incomplete or missing, necessitating extensive manual verification and correction of approximately 300 images. Another major challenge was the unrealistic scale of mites in many samples, where they appeared extensively magnified (see Figure 4.4), deviating from their natural size relative to bees. This inconsistency led to detection errors, with the model sometimes misclassifying background elements as mites. Despite preprocessing and tailored training strategies, the issue persisted and has been flagged as an area for future improvement (see Section 8.3).

In addition, datasets exhibited a natural class imbalance, with *Varroa* mites significantly underrepresented compared with bees. Although selective sampling and targeted augmentation were used to reduce this effect, the imbalance continued to influence model training and performance.

Figure 4.4: Extensively magnified view of mites used in training data, differing from natural proportions.Bee (2025)

Further challenges stemmed from variability in image quality and dataset diversity. Some images suffered from poor lighting, motion blur or low resolution, particularly in smaller datasets Alice (2025*a*); Bolo (2025). These conditions made it harder for the model to learn fine-grained features, particularly for small object detection. Low-quality images were excluded, and enhancement techniques were applied to the remaining samples.

The diversity of perspectives, backgrounds and object appearances, although valuable for generalisation, added complexity to consistent feature learning. A progressive training strategy was employed to help the model gradually adapt to this variability. Table 4.3 summarises the key challenges encountered and the mitigation strategies applied.

Despite these difficulties, the combination of manual annotation refinement, quality control and staged training provided a robust foundation for training YOLOv8n. The final model demonstrates reliable detection performance across varied scenarios and offers valuable insights into the data curation demands of small object detection in embedded AI applications.

Table 4.3: Key Challenges in Dataset Preparation and Mitigation Strategies

Challenge	**Mitigation Strategy**
Inconsistent annotation quality	Manual verification and correction of approximately 300 images
Scale inconsistency of mites	Specialised training approaches; identified as area for future improvement 8.3
Class imbalance	Selective sampling and targeted augmentation of the Varroa mite class
Variable image quality	Exclusion of low-quality images and application of enhancement techniques
Small object detection	Emphasis on *Varroa* mite annotations and use of high-resolution images Alice (2025*b*)
Dataset diversity management	Progressive training approach for gradual adaptation to diversity

4.3 Model Implementation

4.3.1 YOLOv8n Architecture

The YOLOv8n architecture represents an evolution within the YOLO family, designed as a single-stage, anchor-free object detection model optimised for real-time applications on embedded systems and edge devices. Its balance of accuracy and computational efficiency makes it particularly suitable for detecting small objects such as *Varroa* mites in complex beehive environments Terven et al. (2023) (the full architecture diagram provided in Appendix A Figure A.3).

YOLOv8n consists of three integrated components: the backbone, neck and detection head, which work together to deliver efficient object detection. The backbone uses a modified CSPDarknet53 architecture featuring C2f modules, introducing Cross Stage Partial connections to enhance feature reuse across the network. This structure reduces redundant computation while improving efficiency Solimani et al. (2024). Convolutional layers throughout the backbone extract hierarchical spatial features, combining varying levels of semantic and positional information. Each convolutional layer applies Batch Normalisation (BN) and the Sigmoid Linear Unit (SiLU) activation function, defined as:

$$\text{SiLU}(x) = x \cdot \sigma(x), \quad \sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.1}$$

A key part of the backbone is the Spatial Pyramid Pooling Fast (SPPF) module, which efficiently aggregates multi-scale features. The SPPF pools features at different scales using parallel max-pooling layers and concatenates the results into a unified representation:

$$\text{SPPF}(F) = \text{Concat}[\text{MaxPool}(F, s) \mid s \in \{5, 9, 13\}] \tag{4.2}$$

This mechanism increases the receptive field without significantly raising computational cost, an important factor for detecting small targets like *Varroa* mites within cluttered backgrounds Sary et al. (2023).

The neck employs a Path Aggregation Network (PANet) to refine and aggregate spatial and semantic features across multiple resolutions through bidirectional feature fusion.

This approach helps maintain precise localisation of small objects by preserving detailed spatial information at different scales Terven et al. (2023).

The detection head is designed as an anchor-free, decoupled structure that directly predicts bounding box coordinates relative to the grid cell's spatial position, removing the need for predefined anchor boxes. It uses separate branches for bounding box regression and class probability prediction, improving training stability by isolating conflicting gradients Terven et al. (2023). The detection head predicts bounding box coordinates x, y, w, h, objectness scores O, and class probabilities C, formally expressed as:

$$\mathbf{b} = (x, y, w, h), \quad P(\text{object}) = \sigma(O), \quad P(\text{class}_i|\text{object}) = \frac{e^{C_i}}{\sum_j e^{C_j}} \tag{4.3}$$

During training, YOLOv8n minimises a composite loss function combining bounding box regression loss (Complete IoU, CIoU), objectness loss (Binary Cross-Entropy) and classification loss (Cross-Entropy):

$$\mathcal{L}_{total} = \alpha\mathcal{L}_{\text{CIoU}} + \beta\mathcal{L}_{\text{obj}} + \gamma\mathcal{L}_{\text{cls}} \tag{4.4}$$

where α, β, γ are balancing hyperparameters tuned to improve performance and convergence stability Terven et al. (2023).

The selection of YOLOv8n for *Varroa* mite detection leverages its strengths in small-object detection, achieved through the C2f backbone's preservation of spatial detail, the SPPF module's contextual feature extraction and the anchor-free head's adaptability to irregular object shapes and scales Terven et al. (2023); Solimani et al. (2024); Yang et al. (2023).

4.3.2 Model Configuration and Architectural Variants

This project evaluated two distinct YOLOv8-based configurations: the standard YOLOv8n and an experimental variant optimised for small-object detection, referred to as YOLOv8n-SO.

The standard YOLOv8n model was chosen as the baseline due to its lightweight, anchor-free architecture and strong suitability for real-time embedded deployment. It employs three detection heads (P3, P4, P5), supporting multiscale detection across objects of varying sizes Lee et al. (2024); Terven et al. (2023).

By contrast, the YOLOv8n-SO model was adapted from modifications proposed by Krasnov et al. Feng et al. (2024), who introduced several architectural changes to improve performance on small-object detection benchmarks. The principal differences are summarised below:

Table 4.4: Architectural Comparison of YOLOv8n and YOLOv8n-SO Configurations

Feature / Module	**YOLOv8n (Standard)**	**YOLOv8n-SO (Small Object)**
Detection Heads	P3, P4, P5	P2, P3, P4 (P5 removed)
Backbone	C2f with standard connections	C2f with CBAM attention modules
Feature Fusion	Concatenation	Bidirectional feature fusion
Small Object Focus	Moderate	High
Parameter Count	Approximately 3 million	Approximately 1.5 million
Deployment Outcome	Final model deployed	Excluded due to scale inconsistency

4.3.3 Implementation Framework

The implementation was developed using the official Ultralytics YOLOv8 repository as the core framework, which provides modular components for model architecture, training workflows and inference execution. The repository structure includes directories such as `ultralytics/models` for YAML-based model definitions, `ultralytics/nn` for custom neural network modules, `ultralytics/engine` for training and validation pipelines, `ultralytics/yolo` for YOLO-specific logic, and `ultralytics/utils` for utility functions. For the baseline YOLOv8n model, these components were used with only minimal configuration adjustments.

The small-object variant (YOLOv8n-SO) required additional architectural modules based on the structure proposed by Krasnov et al. Feng et al. (2024). These modifications were implemented by extending `block.py` with custom attention modules, updating the `parse_model()` function in `tasks.py`, and defining new model configurations within a dedicated directory. All changes were made while maintaining compatibility with the Ultralytics training interface.

An overview of the YOLOv8n training and deployment pipeline is provided in Appendix B.1.

Custom scripts were also developed to support training, dataset preprocessing and Hailo-8L compilation. These included wrappers for progressive dataset integration, dataset formatters to standardise annotations and inference adapters to convert ONNX models into Hailo-specific `.hef` binaries. Although most training and inference procedures reused Ultralytics modules, the deployment pipeline was independently constructed to meet edge constraints, following documentation from both the Hailo SDK and previous embedded deployment studies Narcia-Macias et al. (2024); Krispin-Avraham et al. (2024).

4.3.4 YOLOv8n-SO Architecture

The YOLOv8n-SO model, based on the work of Krasnov et al. Feng et al. (2024), was implemented as an experimental architecture specifically optimised for small object detection. The core enhancement involves the introduction of a high-resolution P2 detection head, which operates at a spatial resolution of $(H/4, W/4)$, thereby improving the model's ability to detect small objects that might otherwise be missed by the standard P3–P5 detection heads. To balance this, the P5 head was removed, reducing the overall number

of parameters and eliminating detection for large-scale objects, a trade-off aligned with the focus on detecting fine features such as *Varroa* mites.

Further improvements include the integration of a bi-directional feature fusion mechanism in place of standard concatenation. This module aggregates information from both shallower and deeper feature maps, allowing simultaneous preservation of contextual and fine-grained spatial cues. For each feature level $l \in \{P2, P3, P4\}$, the process involves collecting adjacent resolution features, resampling them to match l, concatenating them, and processing the result with the enhanced C2fCBAMv2 module.

Additionally, Convolutional Block Attention Modules (CBAM) were introduced within the backbone, particularly in the C2f blocks. These attention mechanisms enhance the model's focus on relevant spatial and channel-level features while suppressing irrelevant activations. The CBAM consists of sequential channel and spatial attention operations:

1. Channel attention computes a channel-wise weighting vector by modelling interdependencies between feature channels.
2. Spatial attention generates a spatial attention map highlighting informative regions.

These attention mechanisms are particularly valuable for detecting small objects in cluttered environments, such as *Varroa* mites on bee bodies or hive surfaces.

An overview of the YOLOv8n-SO architectural modification process is provided in Appendix B.2, alongside the updated bi-directional feature fusion module shown in Figure 4.5.

The architecture was implemented by extending the Ultralytics `YOLOv8` codebase, involving custom additions to `block.py`, modifications to `parse_model()` in `tasks.py`, and the creation of new YAML configurations. These modifications followed recent trends in enhancing lightweight detectors through selective attention and scale-aware design Bringas et al. (2023); Liu et al. (2023).

The model was trained using the same pipeline as YOLOv8n, leveraging the curated dataset of annotated bee and *Varroa* images (see Section 4.2).

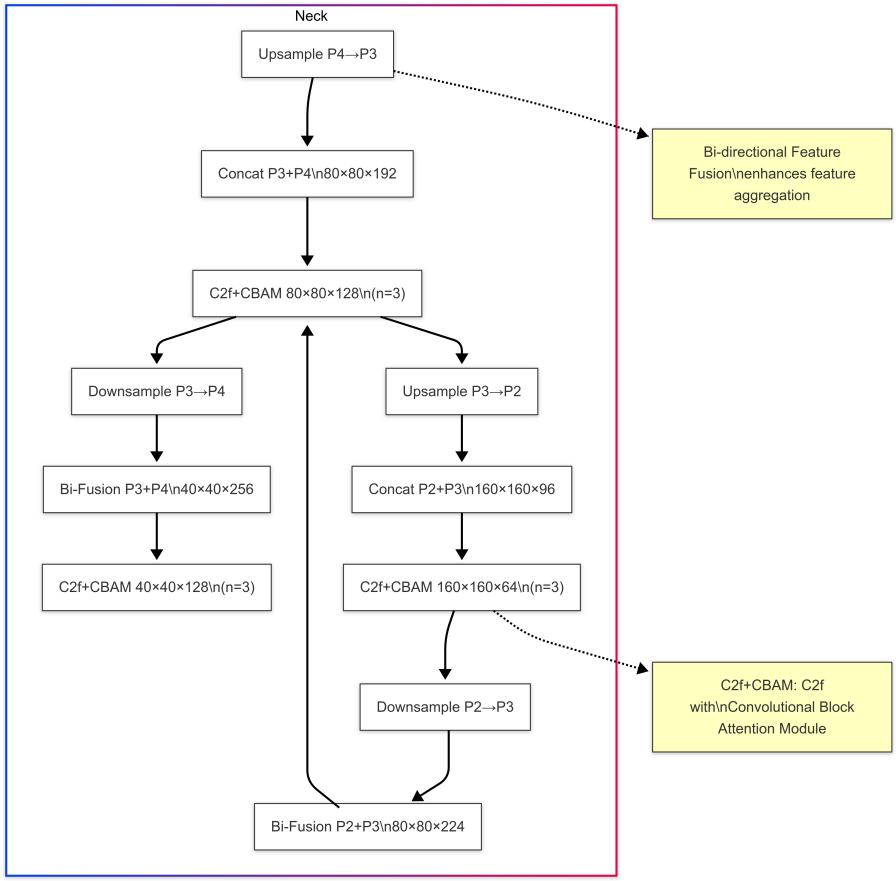

Figure 4.5: The bi-directional feature fusion neck module of YOLOv8n-SO with feedback loop.

4.4 Hailo 8L Platform Integration, Deployment and Real-Time Operation

This section presents the complete deployment strategy for transitioning the trained YOLO-based *Varroa* detection models into a real-time, edge-optimised monitoring system. The primary objective was to achieve accurate, low-latency detection of *Varroa* mites under practical field conditions using constrained embedded hardware.

The system was implemented on a compact, power-efficient platform comprising a Raspberry Pi 5 (8GB RAM) combined with the Hailo-8L M.2 AI accelerator, as outlined in Section 3.3.1. Several critical requirements shaped the deployment: maintaining inference latency below 100 ms to enable timely intervention, supporting offline operation without cloud dependency, tolerating cluttered backgrounds and fluctuating lighting conditions, and enabling seamless integration with a Flask-based dashboard for local monitoring and logging (see Section 3.3.2).

To bridge the gap between training in high-performance environments (NVIDIA RTX 4090 GPU) and real-time edge deployment, a multi-stage model conversion and compilation pipeline was established. This involved exporting trained PyTorch models to ONNX format, followed by quantisation-aware compilation into Hailo's proprietary `.hef` format using the Hailo Model Zoo and Dataflow Compiler tools. Detailed technical steps for

model export, quantisation, compilation and deployment are provided in Appendix B.3.

Due to the Hailo SDK's incompatibility with native Windows systems, the compilation process was completed within a WSL2 Ubuntu environment. The final binary, optimised for Hailo-8L execution, was then transferred to the Raspberry Pi and deployed within an isolated Python runtime to support hardware-accelerated inference.

Bounding boxes were rendered on an HDMI-connected display using OpenCV overlays, with the visualisation loop running in a separate thread to allow asynchronous detection and display. At runtime, detections are colour-coded: bees (Class 0) are outlined in green, while *Varroa* mites (Class 1) are highlighted in blue. Frame-by-frame logs record detection counts for each class, bounding box coordinates (x, y, width, height), confidence scores (0–1) and timestamps with millisecond precision. The system continuously calculates a real-time infestation ratio (*Varroa* count divided by bee count), providing a critical colony health metric as defined in Section 3.2.3. An example of live detection output is shown in Figure 4.6.

Figure 4.6: Live detection view showing bounding boxes for detected bees (green) and *Varroa* mites (blue), including confidence scores and frame rate overlay.

This deployment approach reflects best practices in embedded AI system design, consistent with trends observed in autonomous monitoring Krispin-Avraham et al. (2024) and pest detection research Varghese and Sambath (2024), where real-time performance, deployment autonomy and resource efficiency are paramount.

4.5 User Interface Development and Implementation

The user interface implementation phase transformed the conceptual dashboard design outlined in Section 3.3.2 into a fully functional web application that provides beekeepers

with real-time colony health monitoring capabilities. This section details the technical development process behind the frontend interface and its integration with the backend detection system, focusing on how design specifications were manifested in code, the development challenges encountered and the technical solutions applied.

The dashboard serves as the critical visual bridge between the complex AI detection processes running on the Hailo accelerator and the beekeeper end-users who require clear, actionable insights about colony health. While the design phase established the architectural vision and UI/UX principles, this implementation section demonstrates how these were realised through HTML structure, CSS styling, JavaScript functionality and backend integration.

The implementation process prioritised four key technical goals that directly address the requirements established in Section 3.3.2:

1. Presenting detection data in an interpretable, visually intuitive format
2. Ensuring responsive performance even on field devices with limited capabilities
3. Providing reliable control over the detection process
4. Maintaining system stability during extended monitoring sessions

These goals guided the technical decisions throughout the development process, from the selection of web technologies to the optimisation of data processing algorithms, creating a direct link between design intent and implementation outcomes.

4.5.1 Frontend Technical Implementation

The frontend implementation encompassed three primary technical domains HTML structure, CSS styling and JavaScript functionality each requiring specific approaches to transform the design concepts into a cohesive, functioning interface optimised for the monitoring context.

HTML Structure Implementation

The frontend implementation began with the development of a semantic HTML structure to serve as the foundation for the dashboard layout conceived in Section 3.3.2. Rather than using generic containers, the implementation utilised semantic HTML5 elements to enhance accessibility and document structure. The document hierarchy was carefully organised to reflect the logical relationship between dashboard components defined during the design phase.

Bootstrap's responsive grid system was leveraged for the underlying layout structure, ensuring compatibility across various screen sizes while maintaining visual integrity of the monitoring interface. Special attention was given to component hierarchy, with each functional area implemented as a distinct section with appropriate nesting of child elements.

The detailed construction steps for the dashboard HTML structure are provided in Algorithm 4 in Appendix B.4, while an illustration of the resulting layout is shown in Figure D.1 in Appendix D.

Throughout the HTML implementation, static elements such as labels and headers were explicitly defined in the markup, while dynamic content elements were implemented as empty containers with specific ID attributes to be populated by JavaScript. This separation of structure and dynamic content enhanced maintainability and provided clear integration points between the HTML and JavaScript components, establishing the foundation for the real-time data flow capabilities specified in the design.

CSS Styling Implementation

The visual styling of the dashboard was implemented through a combination of Bootstrap's utility classes and a comprehensive custom stylesheet, directly translating the color schemes and visual elements described in Section 3.3.2. The implementation approach emphasised maintainability, performance and visual consistency across the interface.

At the foundation of the CSS implementation was a robust variable system using CSS custom properties. This system defined the entire color palette, typography scale, spacing values and component dimensions in a centralised location (see Algorithm 5 in Appendix B.4).

The color system was implemented with primary emphasis on the honey gold (#ffc107) and royal purple (#6f42c1) colors established in the design phase, supplemented by a range of functional colors for status indicators. These color choices directly implement the color-coded risk levels specified in Section 3.3.2 (low, moderate, high, critical), with the palette carefully selected to enhance readability in field conditions (see Figure D.3 in Appendix D).

Risk badges were implemented with distinct styling based on severity levels, with the critical risk state receiving additional pulse animation to draw attention to urgent conditions.

Component styling was implemented in logical blocks corresponding to the Key Metrics Cards specified in the design phase. The metric cards received consistent internal padding, subtle hover animations and standardised typography across all instances.

Responsive behaviour, a key requirement identified in the design phase for field use, was implemented through targeted media queries that adjusted component spacing, font sizes and layout configurations at key breakpoints. The primary breakpoint at 768px marked the transition between desktop and mobile layouts, with additional fine-tuning at 576px for smaller devices (see Figure D.2 in Appendix D).

This responsive implementation ensured the dashboard remained functional and legible across a range of potential field devices, including tablets and smartphones that might be used in apiary settings. The CSS implementation balanced aesthetic considerations with the practical requirements of field usage, ensuring the interface remained readable in outdoor lighting conditions and operated efficiently on devices with limited processing capabilities, directly addressing the field usability requirements specified in the design.

JavaScript Functionality: Visual Feedback and Animation

Visual feedback and animation elements were crucial components of the JavaScript implementation, providing users with intuitive cues about system state changes and data updates as specified in the Real-Time Data Flow requirements from Section 3.3.2. These

animations enhanced the user experience by smoothly transitioning between states rather than presenting jarring, immediate changes to displayed values.

The centerpiece of the animation system was a custom value transition function that smoothly animated changes to numeric displays throughout the dashboard. The implementation details are provided in Algorithm 6 in Appendix B.4.

Representative examples of animated metric transitions are shown in Appendix D.4.

This animation function was applied to all dynamic numerical values in the dashboard, including bee counts, *varroa* counts and infestation ratios. By using jQuery's animation capabilities with custom step functions, the implementation achieved smooth transitions that helped users visually track changes in colony health metrics.

For risk status changes, a dedicated function was implemented to animate the transition between different risk levels with appropriate visual feedback, directly implementing the risk classification thresholds established in Section 3.2.3.

Detection process control animations were implemented to provide feedback during the start and stop operations, enhancing the Control Panel functionality specified in the design. The implementation of the detection button animation is provided in Algorithm 7 in Appendix B.4.

This animation system ensured users received clear feedback when initiating or terminating detection sessions, with smooth transitions between button states that reinforced the current system status. The temporary disabling of the button during transitions prevented users from triggering multiple state changes in rapid succession, enhancing system stability as required in the design specifications.

4.5.2 Backend Integration Implementation

The frontend dashboard was tightly integrated with the Flask backend through a series of REST API endpoints and data processing functions, implementing the data pipeline architecture specified in Section 3.3.2. This integration enabled real-time data flow while maintaining a clean separation of concerns between the presentation layer and the underlying detection system.

API Endpoint Implementation

The Flask backend implemented several key API endpoints to facilitate dashboard functionality, with emphasis on detection control and data retrieval.

These endpoints handled the core control functionality for the detection process, managing the lifecycle of detection sessions and providing status feedback to frontend requests. Careful resource management was implemented to prevent orphaned processes or memory leaks during extended operation, addressing the process management requirements described in Section 3.3.2.

To enable frontend access to live detection results and time-series monitoring data, dedicated retrieval endpoints were developed. These endpoints parsed the backend's internal storage structures (including deque-based buffers) and converted them into JSON responses compatible with the JavaScript frontend, ensuring real-time dashboard updates.

Further database query endpoints were implemented to allow filtered access to past detection sessions and stored metrics, as required for historical analysis. Parameter validation techniques were applied throughout these database endpoints to prevent security vulnerabilities such as SQL injection, and to enforce robust query behaviour even when parameters were omitted or malformed.

The full implementations of the detection control endpoints, statistics and time-series retrieval and database querying functionality are provided in Appendix B.5.1.

Data Processing Implementation

The backend implemented efficient data processing functions to extract meaningful metrics from the detection output, transforming raw detection data into the structured metrics required by the dashboard design.

These functions parsed the unstructured text returned by the detection process and populated structured variables for frontend visualisation, enabling seamless integration with the API endpoints (see Appendix B.5.2). This approach directly implements the data pipeline architecture specified in Section 3.3.2, transforming raw detection data into meaningful, structured metrics.

Time Series Implementation

A dedicated function was implemented to update the time series data structures, which were crucial for the trend visualisation charts specified in the dashboard design. This function extracted values from the detection system and populated circular buffers for plotting and risk assessment (see Appendix B.5.3).

The implementation utilised circular buffers, specifically double-ended queues (`deque`), with fixed maximum lengths to store time series data efficiently. These buffers included four primary components: `timestamps`, which recorded the recent time points of detection events; `bee_counts`, which tracked the number of bees detected per frame; `varroa_counts`, representing the number of *Varroa* mites identified; and `infestation_ratio`, which calculated the percentage of infected bees over time.

This approach directly implements the risk classification thresholds and visualisation charts specified in Section 3.3.2, while ensuring stable memory usage even during extended monitoring sessions through the use of bounded buffers.

Email Notification System Implementation

The system incorporates an automated email notification mechanism that delivers comprehensive colony health reports at the conclusion of detection sessions. This feature enables apiarists to receive detailed analytics about colony health status even when not actively monitoring the dashboard. The email service is integrated with the session management subsystem, automatically triggering report generation when a monitoring session ends as shown in Appendix B.5.4.

The implementation incorporates several reliability features to ensure consistent delivery of health reports. A thread-based execution model with timeout handling prevents SMTP connection issues from blocking the main application thread. The email content generation process dynamically formats colony health metrics using color-coded visual

indicators that match the dashboard aesthetics, providing consistent risk communication across interfaces. Both HTML and plain text formats are generated to ensure compatibility across email clients and devices. The email service configuration is externalised via environment variables, allowing for customisation without code modification. By automatically documenting colony health status at the end of each monitoring session, this feature extends the utility of the system beyond real-time dashboard monitoring, enabling retrospective analysis and maintaining a persistent record of colony health trends over time.(A screnshot of the detection summary email notification is provided in Appendix D.5).

Technical Challenges and Solutions

The dashboard implementation introduced several technical challenges that required targeted solutions to ensure stability and responsiveness.

Asynchronous Data Flow: To avoid UI blocking and backend overload, a lightweight polling mechanism was implemented, refreshing dashboard data every second. Database writes were throttled to occur only every 100 frames to optimise performance.

Responsive Visualisation: A dynamic animation strategy was introduced, adjusting transition speeds based on update frequency. This preserved both clarity and responsiveness during rapid data changes.

Process Management: A robust process termination routine using signal escalation (SIGINT, SIGTERM, SIGKILL) ensured all detection and GStreamer processes were fully cleaned up after session termination, preventing memory leaks and ensuring reliable long-term operation.

These solutions extended beyond initial design expectations while maintaining alignment with the system's real-time monitoring objectives.

4.6 Conclusion

This chapter has detailed the comprehensive implementation of an edge-based *Varroa* mite detection system, successfully integrating computer vision with embedded hardware capabilities. The solution combines a YOLOv8n detection model optimised for small targets with a Raspberry Pi 5 and Hailo-8L accelerator deployment, achieving real-time performance while maintaining detection accuracy. Key achievements include the development of a robust dataset curation pipeline addressing scale inconsistencies, implementation of a progressive training strategy for model optimisation, and creation of an intuitive web dashboard for colony monitoring. The system addresses practical deployment challenges through hardware-aware model compression, efficient resource management, and reliable notification mechanisms. Implementation decisions were rigorously validated against both functional requirements and environmental constraints typical of apiary settings.

Chapter 5

Testing and Experiments

5.1 Introduction

This chapter presents empirical validation of the *Varroa* detection system through structured testing and architectural experiments. A three-phase methodology (unit, integration and performance testing) verifies component functionality, system coherence and edge deployment readiness. Subsequent experiments compare YOLOv8n variants under apiary-relevant challenges, examining how architectural choices and training strategies address dataset limitations from Section 4.2.6. The evaluation framework directly assesses compliance with Section 3.2.3 requirements while informing practical deployment considerations.

5.2 Testing

Testing played a critical role in validating the functionality, integration and performance of the *Varroa* detection system. This section outlines the structured approach adopted to ensure each component, from individual backend and frontend modules to overall system performance, met the design specifications. A three-tier strategy was employed: unit testing for component-level correctness, integration testing for inter-module cooperation and performance testing to assess real-time suitability on embedded hardware.

5.2.1 Unit Testing

Unit testing provided the first layer of validation, targeting individual functions and modules in isolation. A structured test suite separated backend and frontend tests. The file `tests/ui/test_flask_routes.py` validated Flask API endpoints including `/get_stats`, `/get_time_series`, `/start_detection`, and `/stop_detection`, verifying response status codes, JSON formatting and API contract compliance Konig (2021); Goh et al. (2023); Nguyen (2022). Tests established specific states, simulated relevant data and asserted the accuracy of detection metrics such as bee counts, mite counts, infestation ratios and risk levels.

Frontend unit testing was conducted through `static/js/tests/test_dashboard.js` using the Jest framework. Simulated browser interactions verified dashboard responsiveness to data updates. Functions such as `animateValueChange`, `updateRiskStatus` and

`setButtonState` ensured that numerical values updated fluidly, UI indicators reflected risk levels dynamically and button states toggled intuitively Picek et al. (2022); Jin (2020); Achmadiah et al. (2025).

The shared `tests/conftest.py` file provided `pytest` fixtures, enabling isolated test environments, hardware/database mocking and application state resets Liu et al. (2023); Fezari and Al-Dahoud (2023); Longbottom (2019).

Full unit test outputs and backend coverage results are provided in Appendix C, Figures C.1 and C.2.

Unit test execution achieved 80% code coverage across all Flask backend routes. Uncovered segments mainly pertained to rare edge cases difficult to simulate without real-world stimuli. Integration components achieved full coverage Harshavardhan et al. (2024).

JavaScript tests executed within 2–7 ms per test, confirming the lightweight and efficient nature of client-side functions Jin (2020); Feng et al. (2022). Detailed frontend unit test results are presented in Appendix C, Figure C.3.

5.2.2 Integration Testing

Integration testing validated the cooperative behaviour of modules across the full detection workflow. The test file `tests/ui/test_integration.py` orchestrated detection scenarios by triggering the `/start_detection` endpoint, simulating detection data input and validating output via `/get_stats` and `/get_time_series` endpoints. Process termination was verified through `/stop_detection`, confirming complete and correct system transitions Wang and Zhang (2024); Fezari and Al-Dahoud (2023); Longbottom (2019); Alves and Hora (2024); Nguyen (2022); Jin (2020).

Mocked integration test outputs are available in Appendix C, Figure C.4.

Additional tests evaluated system resilience to malformed inputs and invalid states. All integration tests passed, achieving 100% coverage and verifying reliability in controlled environments Varghese and Sambath (2024); Fezari and Al-Dahoud (2023); Goh et al. (2023); Feng et al. (2022); Achmadiah et al. (2025).

5.2.3 Performance Testing

Performance testing assessed real-time capability on embedded hardware (Raspberry Pi 5 with Hailo-8L). A data-driven methodology was adopted using `tests/ui/test_performance.py`, querying detection data from `bee_health.db` to analyse frame throughput and latency Krispin-Avraham et al. (2024); Tidwell (2001); Feng et al. (2022).

The SQL query and extracted session metrics are shown in Appendix C, Figures C.5 and C.6.

Across five recorded sessions, average FPS ranged from 26.83 to 31.03, exceeding the 15 FPS threshold for real-time monitoring. Minimum FPS consistently remained above 6.89, confirming responsiveness under varying load conditions. Peak FPS reached 65.60, demonstrating substantial processing headroom Mika et al. (2023); Jin (2020); Iqbal et al. (2011).

Detailed session-wise FPS results are presented in Appendix C, Table C.1.

5.2.4 Performance Analysis

Performance results revealed high consistency, with a standard deviation of 1.74 in average FPS across sessions. The gap between average and maximum FPS (up to 34 FPS) demonstrated system capacity for high-complexity detection scenarios Feng et al. (2024); Yazdani-Asrami et al. (2022). Stability in minimum FPS values validated reliable operation even during challenging conditions.

Session 43, which exhibited the highest frame count (43 frames) and FPS (65.60), highlighted the system's resilience and robustness for extended real-time deployment Liu et al. (2023); Varghese and Sambath (2024); Achmadiah et al. (2025).

5.2.5 Test Coverage Summary Aligned to Requirements

Table 5.1: Minimal Summary of Implemented Tests and Requirements

Test Type	**Requirement Reference (Section 3.2.3)**	**Status**
Backend Unit Tests	Functional and API correctness	Passed
Frontend Unit Tests	UI feedback and risk status display	Passed
Integration Testing	Full detection workflow	Passed
Error Handling	Fault tolerance and input validation	Passed
Performance Testing	Real-time processing capability	Passed
UI Responsiveness	Interface update latency	Passed
Detection Output Validation	Bee/mite count and health ratio	Passed
Database Query Tests	Session logging and summary metrics	Passed

As summarised in Table 5.1, the tests implemented directly correspond to the system requirements outlined in Section 3.2.3.

5.3 Experiments

This section presents extended experimental procedures carried out to evaluate and refine the two primary architectures discussed in this project: YOLOv8n and its small-object-optimised counterpart YOLOv8n-SO. These experiments, conducted after the initial implementation phase (see Section 4.3), aimed to explore the influence of hyperparameters, data augmentations and structural modifications on model generalisation, especially in the context of small object detection challenges described in Section 4.2.6.

5.3.1 Experimental Framework and Rationale

All experiments adhered to the training-validation split defined in Section 4.2, and the evaluation pipeline used was consistent across both models to ensure fair comparison (refer to Chapter 6). YOLOv8n was selected as a baseline due to its compact architecture, anchor-free detection head and success in previous embedded *Varroa* detection studies Lee et al. (2024); Varghese and Sambath (2024). Its variant, YOLOv8n-SO, was designed based on principles outlined by Feng et al. (2024), who demonstrated that removing the

P5 head and incorporating attention mechanisms improved performance on dense aerial and embedded small-object datasets.

Both models share the same input resolution and core training scripts but differ significantly in their architectural emphasis: YOLOv8n targets general object scales while YOLOv8n-SO biases towards preserving spatial detail in early layers, as discussed in Section 4.3.4. This comparative study explores how those differences interact with hyperparameter strategies and augmentation choices under real-world data inconsistencies.

5.3.2 Hyperparameter Optimisation Strategy

The training hyperparameters for YOLOv8n and YOLOv8n-SO were not arbitrarily selected; they were informed by literature such as Bringas et al. (2023), Liu et al. (2023), Varghese and Sambath (2024), and the architectural requirements of Section 4.3.1. YOLOv8n, benefiting from stable convergence with medium batch sizes, was trained with a batch size of 16 over 100 epochs using AdamW optimiser and cosine learning rate scheduling approaches shown to be optimal for transfer learning and regularisation on imbalanced datasets Varghese and Sambath (2024); Feng et al. (2024). For YOLOv8n-SO, a significantly smaller batch size of 2 was used due to memory constraints introduced by additional feature maps and the inclusion of CBAM attention modules. This configuration aligns with findings in Divas'on et al. (2024) and Feng et al. (2024) that emphasise the need for extended training schedules when high-resolution feature retention is prioritised.

Table 5.2: Hyperparameter configuration used during extended experiments.

Parameter	**YOLOv8n**	**YOLOv8n-SO**
Batch Size	16	2
Epochs	100	1000
Optimiser	AdamW	AdamW
LR Schedule	Cosine	Cosine
Initial LR	0.001	0.001
Warmup Epochs	3	5
Patience	20	150
Mosaic	1.0	1.0
Mixup	0.1	0.3
Copy-Paste	0.0	0.3
HSV Aug	0.5	0.7

The high patience value for YOLOv8n-SO was critical to allow convergence on low-gradient updates without prematurely stopping a problem documented in low-resolution *Varroa* datasets with high background noise (see Section 4.2). These configurations collectively contributed to the improved mAP@0.5:0.95 performance discussed in Chapter 6.

5.3.3 Augmentation and Regularisation Studies

Given the class imbalance discussed in Section 4.2.6, this study investigated several augmentation strategies targeting small object robustness. YOLOv8n utilised a modest augmentation profile (`Mixup`=0.1, `HSV`=0.5), providing basic geometric and colour variation without overwhelming the model during early training. By contrast, YOLOv8n-SO

adopted an aggressive augmentation suite, including `Mixup`=0.3, `CopyPaste`=0.3 and `HSV`=0.7. These parameters were justified by works such as Liu et al. (2023), who showed that increasing augmentation diversity can mitigate false negatives in cluttered scenes.

Furthermore, YOLOv8n-SO incorporated CBAM attention and deeper bi-directional fusion paths. These mechanisms enabled it to better attend to faint visual features like semi-occluded *Varroa* mites, especially in cases of background camouflage, as described in Bringas et al. (2023); Divas'on et al. (2024). However, this made the model more sensitive to annotation inconsistencies, a limitation discussed in Section 4.2.5.

Although Contrast Limited Adaptive Histogram Equalization (CLAHE) was considered for preprocessing (based on insights from Sharma et al. (2024)), it was excluded due to its high latency during deployment simulations.

5.4 Conclusion

Testing confirms the system meets operational requirements, with unit/integration tests verifying functionality and performance benchmarks demonstrating real-time capability (26.83-31.03 FPS). Experimental comparisons reveal YOLOv8n's superior robustness over its small-object variant despite theoretical advantages, highlighting annotation sensitivity in specialised architectures. Strategic augmentation balances class imbalance while maintaining deployment efficiency. These results validate the system's technical viability and underscore annotation quality's critical role in detection performance.

Chapter 6

Results and Evaluation

6.1 Introduction

This chapter presents a comparative evaluation between two YOLO-based architectures YOLOv8n and a modified YOLOv8n-SO tailored for small object detection of *Varroa destructor* on *Apis mellifera*. Both models were trained and tested under the same dataset and hardware constraints. The assessment spans detection accuracy, class-wise performance, and deployment viability. Each metric is critically analysed and benchmarked against findings in the literature (Chapter 2).

6.2 Evaluation Criteria and Methodology

Model performance was evaluated using a comprehensive set of detection metrics to ensure numerical accuracy and practical deployability in real-world settings. The following key metrics were used to assess each architecture:

1. **Precision**: $\frac{TP}{TP+FP}$ This quantifies the proportion of correctly predicted positive observations (true positives) against all predictions made for the positive class (true positives + false positives). High precision indicates that the model commits few false alarms, which is especially critical when distinguishing *Varroa* from background noise in cluttered hive imagery.

2. **Recall**: $\frac{TP}{TP+FN}$ Recall measures how effectively the model identifies all actual instances of the target class (true positives + false negatives). A high recall implies that very few *Varroa* mites were missed, an essential trait for early-stage infestation detection.

3. **F1 Score**: $\frac{2 \cdot (\text{Precision} \cdot \text{Recall})}{\text{Precision} + \text{Recall}}$ As the harmonic mean of precision and recall, the F1 score balances the trade-off between over-detection and under-detection. It is particularly useful in unbalanced datasets or when false positives and false negatives are equally detrimental.

4. **mAP@0.5**: Mean Average Precision calculated at an Intersection over Union (IoU) threshold of 0.5. It represents how accurately the bounding boxes are predicted and matched to the ground truth. This is a conventional benchmark for object detection performance in research.

5. **mAP@0.5:0.95**: The COCO-style mean Average Precision, averaged over multiple IoU thresholds from 0.5 to 0.95 with increments of 0.05. This metric provides a more stringent and comprehensive assessment of detection robustness and localisation consistency.
6. **Confidence Threshold**: The minimum confidence score a prediction must exceed to be considered valid. Adjusting this threshold impacts the trade-off between false positives and false negatives and plays a critical role in tuning the model for optimal field performance.
7. **Box Loss**: This loss function evaluates the accuracy of the predicted bounding box coordinates relative to the ground truth. Lower box loss values indicate better localisation and tighter bounding boxes around *Varroa* mites and bees.
8. **Classification Loss (CI)**: Measures how well the model differentiates between different object classes (e.g. bee, *Varroa*, background). A low classification loss suggests high class-separation performance, which is essential in complex biological settings.
9. **Distribution Focal Loss (DFL)**: A component introduced in YOLOv8 for more refined bounding box regression. It models bounding box localisation as a probabilistic distribution rather than a single value, improving accuracy in detecting small, often occluded objects such as *Varroa* mites.

In addition to these quantitative metrics, qualitative performance was assessed using a range of visual diagnostic tools, including confusion matrices, precision-recall curves, F1-confidence plots and confidence calibration charts. These visualisations provide insight into the models' behaviour across varying thresholds and operational conditions, enabling more nuanced performance interpretation in the context of real-time embedded applications.

6.3 Confusion Matrix Analysis

Figure 6.1 reveals stronger class separation in YOLOv8n. The model identifies bees with 85% accuracy and *Varroa* mites with 90%, while maintaining background misclassification below 5%. This indicates good generalisation across all categories and strong separation between foreground and background features, essential in high-clutter scenes (Section 2). In contrast, YOLOv8n-SO exhibits poor class separation. It misclassifies 27% of *Varroa* samples as background and fails entirely to detect 16% of bees, confirming performance collapse when fine-grained features are removed, as seen in Bringas et al. (2023); Liu et al. (2023); Feng et al. (2024).

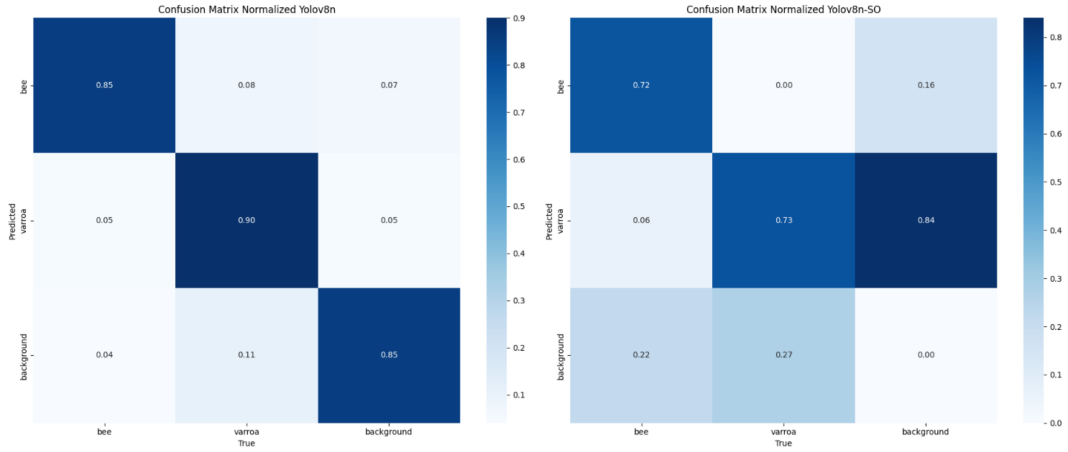

Figure 6.1: Normalised Confusion Matrices for YOLOv8n and YOLOv8n-SO.

6.4 F1-Confidence Comparison

Figure 6.2 highlights model confidence robustness. YOLOv8n peaks at F1 = 0.84 with minimal variance across thresholds 0.3–0.6, suggesting stable balance between precision and recall across confidence ranges. This robustness is essential in field deployments, where lighting and object clarity vary Wang and Zhang (2024). In contrast, YOLOv8n-SO reaches F1 = 0.67, but only in a narrow threshold band. Beyond this, bee and *Varroa* F1 scores degrade rapidly. Such instability can severely impact usability in real-time systems and affirms the risks of aggressive spatial downsampling as noted by Feng et al. (2024).

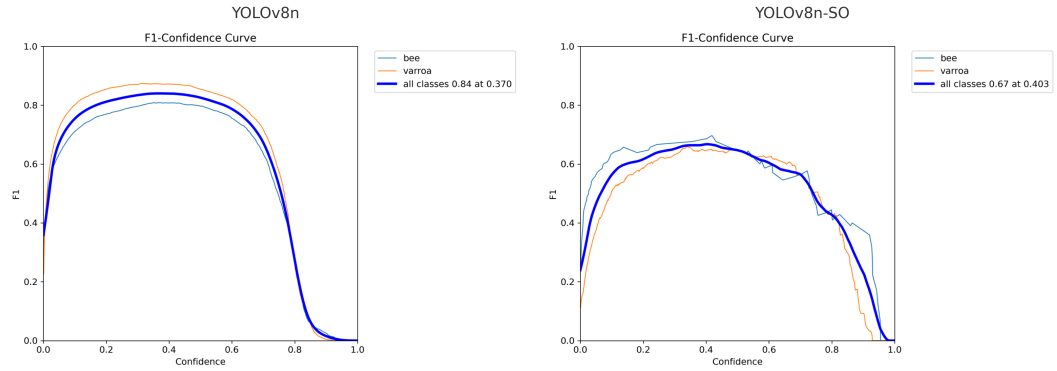

Figure 6.2: F1-Score across Confidence Thresholds.

6.5 Qualitative Detection Comparison

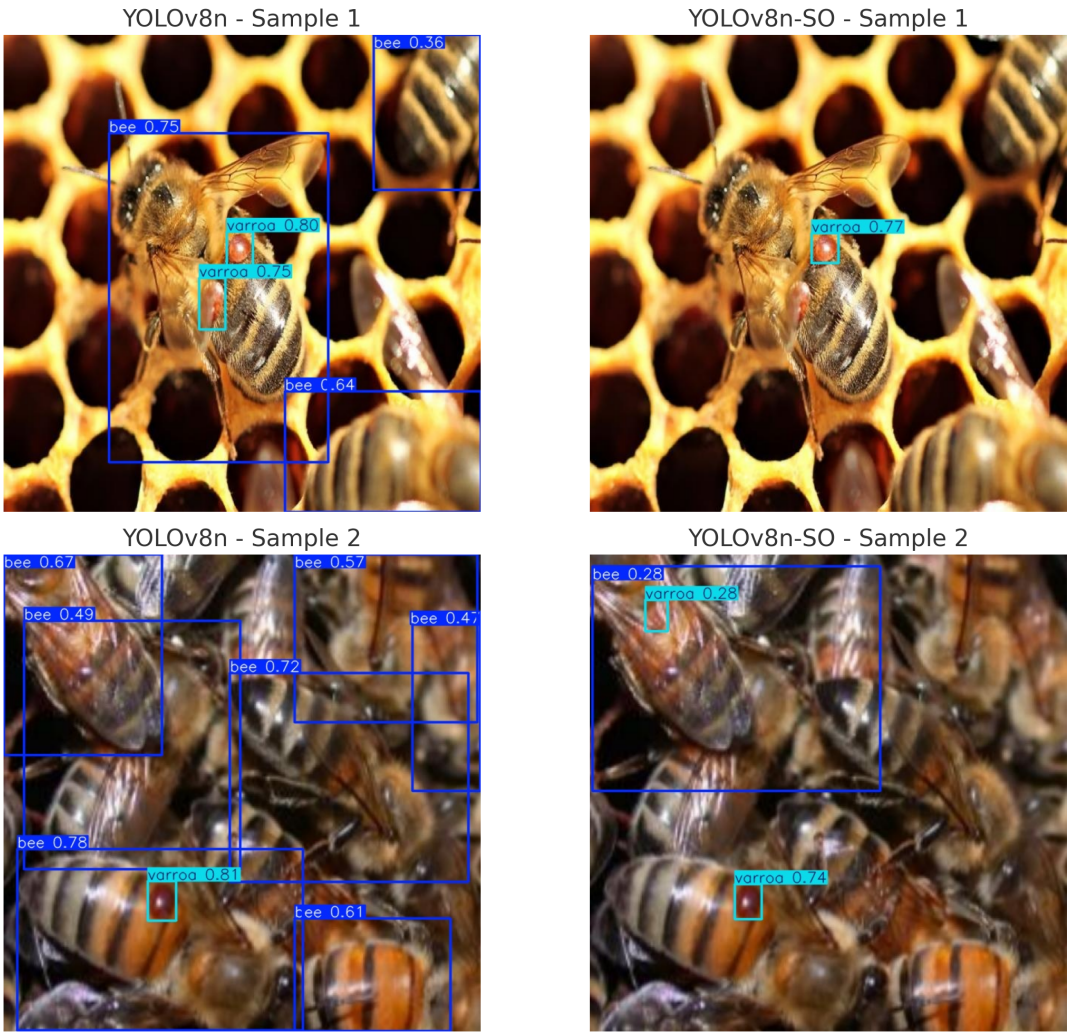

Figure 6.3: Sample detections: Left = YOLOv8n; Right = YOLOv8n-SO.

In both samples (Figure 6.3), YOLOv8n successfully detects multiple mites with high confidence, even when partially occluded, reflecting its generalisation capability. In the second image, it correctly identifies overlapping *Varroa* instances amid motion blur, demonstrating high context sensitivity. Conversely, YOLOv8n-SO misses most full-sized instances and falsely identifies shadows and thoracic textures as *Varroa* mites. This supports critiques raised in Liu et al. (2023); Feng et al. (2024) regarding resolution sensitivity after P5 layer removal. This combined with the attention mechanism leads to struggling with detection of magnified instances; as shown in the picture, the model tends to focus and accurately detect the smaller instances by ignoring the ones that appear in a bigger size (see Figure 6.3 Sample 1).

6.6 Precision-Recall Analysis

As seen in Figure 6.4, YOLOv8n exhibits high AP values for both classes, with minimal drop-offs. Its curve is smooth and elongated, indicating confident and complete detections

at multiple thresholds. In contrast, YOLOv8n-SO displays sharp staircasing in PR shape, often linked to binary-like confidence outputs and poor spatial localisation. The 32% mAP@0.5 performance gap mirrors trends in Narcia-Macias et al. (2024) and Varghese and Sambath (2024), where downsized models led to reduced discrimination in biologically dense contexts.

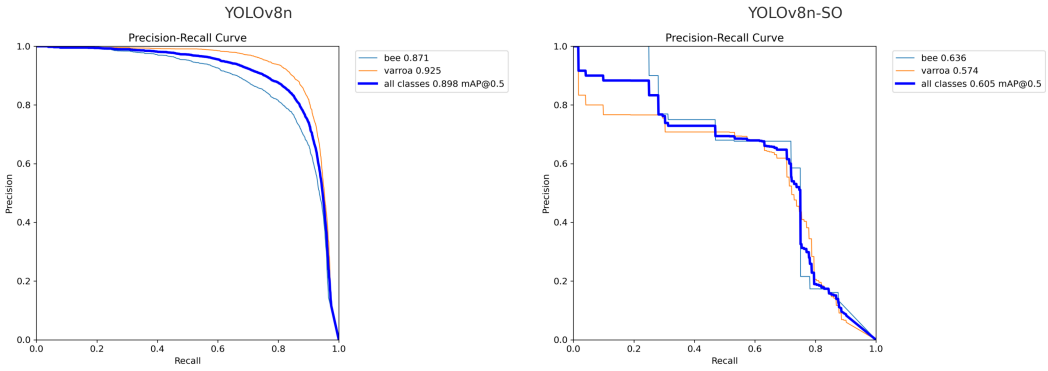

Figure 6.4: Precision-Recall Curves for both models.

6.7 Precision and Recall vs Confidence

Figure 6.5 illustrates confidence calibration. YOLOv8n's precision increases consistently with confidence, reaching near 1.0 as false positives are eliminated. Recall degrades gradually, showing no sudden collapses ideal for embedded real-time systems Mrozek et al. (2021). YOLOv8n-SO, by contrast, shows erratic performance beyond 0.6 threshold. Precision spikes and dips suggest misalignment between scoring function and prediction consistency, confirming observations from Bringas et al. (2023) about feature compression harming reliability.

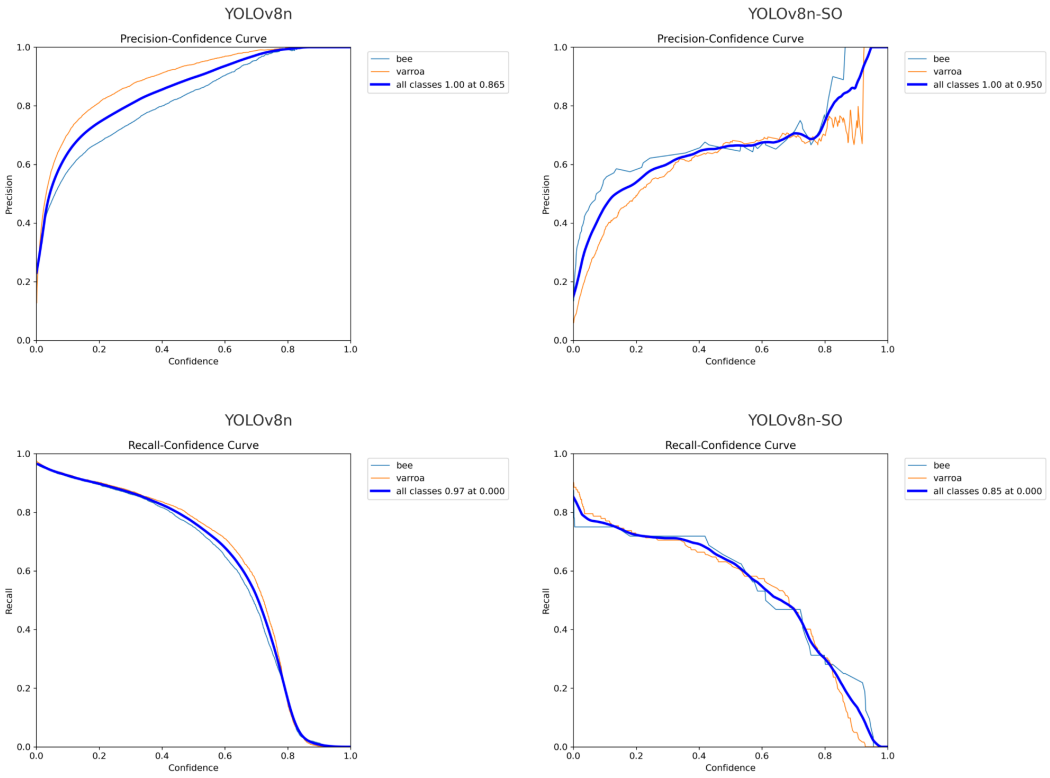

Figure 6.5: Precision and Recall vs Confidence.

6.8 Training Loss and Optimisation Behaviour

In Figure 6.6, YOLOv8n shows consistent, steady convergence across all loss components in under 100 epochs. Box and class loss stabilise early, with gradual tapering of DFL loss ideal for efficient training. YOLOv8n-SO, despite 500+ epochs, displays volatility in DFL and validation loss, particularly past epoch 300. Such instability is typical when feature compression limits spatial encoding, as previously noted in Divas'on et al. (2024). Moreover, fluctuating validation metrics highlight overfitting risks in undersized detectors.

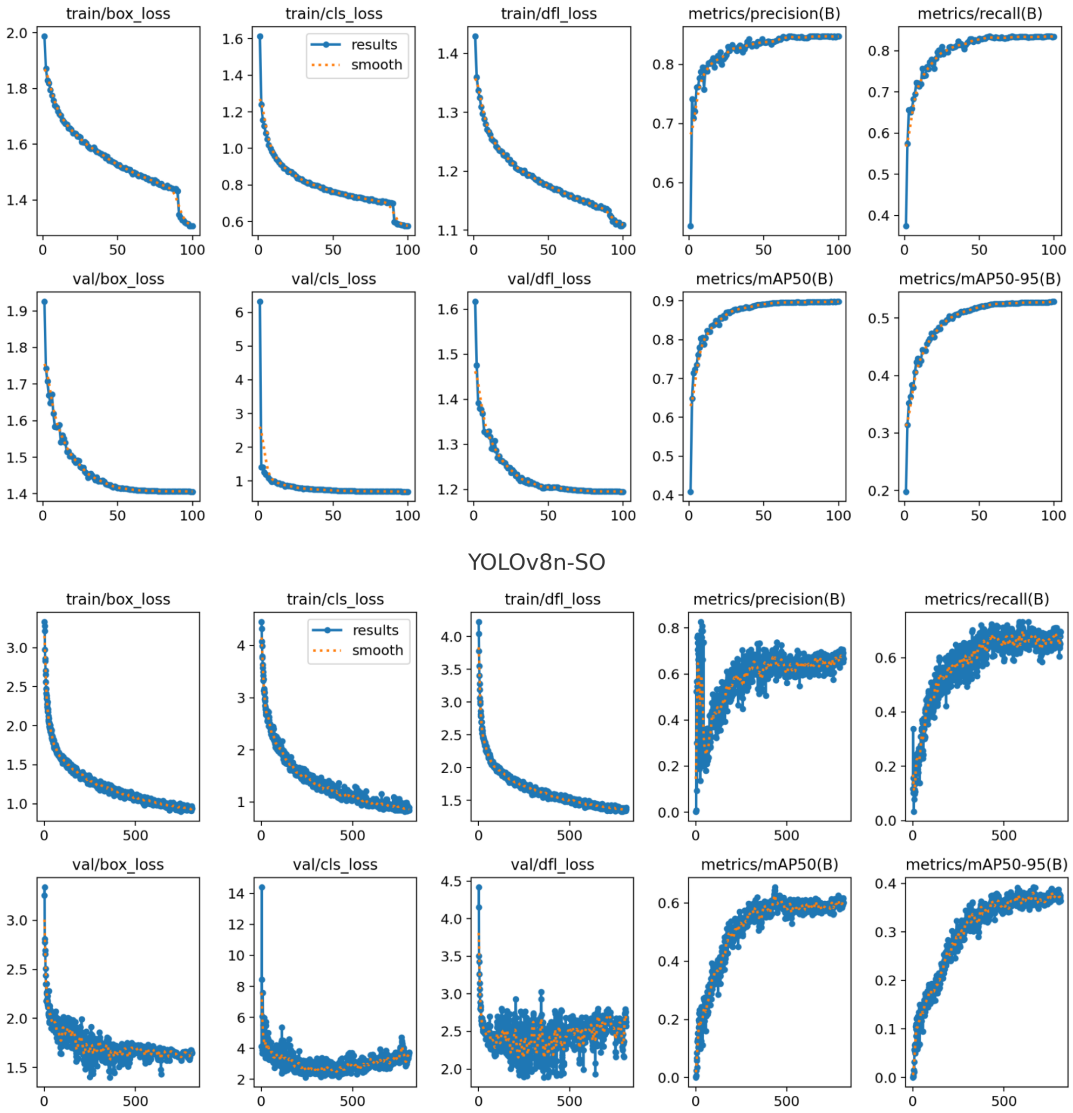

Figure 6.6: Loss curves and metric trends for both models.

6.9 Conclusion

Table 6.1 summarises the final evaluation metrics. YOLOv8n decisively outperforms YOLOv8n-SO in every category particularly in F1 score, mAP and detection stability. Its compatibility with Hailo-8 and better field generalisation makes it the recommended choice for embedded *Varroa* detection systems, consistent with real-world deployment literature Lee et al. (2024); Narcia-Macias et al. (2024); Mrozek et al. (2021).

Table 6.1: Summary of Key Evaluation Metrics (already included in results)

Model	Precision	Recall	mAP@0.5	F1 Score
YOLOv8n	**0.861**	**0.842**	**0.898**	**0.84**
YOLOv8n-SO	0.667	0.695	0.605	0.67

Chapter 7

Legal, Social, Ethical and Professional Issues

This system has been designed with full awareness of the legal, social, ethical and professional responsibilities that accompany the deployment of AI technologies in biological monitoring contexts. These considerations ensure the solution remains safe, privacy-compliant, environmentally respectful and professionally engineered.

Legal Considerations

The system does not collect, transmit or store any personally identifiable information. All image data are processed locally on the Raspberry Pi 5, with no use of cloud-based services. This design complies with GDPR principles and avoids any user authentication or device tracking. The local-first architecture ensures that all detections, visualisations and logs remain securely stored on the device, minimising legal risk and improving transparency.

Social Considerations

The primary social benefit of the system is its contribution to sustainable beekeeping practices. Traditional detection methods, such as alcohol washes and sticky boards, require the sacrifice of bees and intrusive interaction with the hive Bjerge et al. (2019). By contrast, this system introduces a non-invasive, real-time monitoring method, reducing colony disruption and promoting ethical apiculture. The visual dashboard is designed to be intuitive and usable even by non-technical users, expanding accessibility for rural and low-tech communities.

Ethical Considerations

This project aligns with modern animal welfare ethics by eliminating invasive sampling procedures. The use of a passive visual detection system aligns with best practices for minimising ecological harm Turyagyenda et al. (2025). In addition, the detection process is transparent, presenting infestation ratios and confidence scores in real time, thereby avoiding black-box AI concerns. Alerts are only triggered when infestation ratios surpass empirically defined thresholds, which minimises the risk of unnecessary interventions Bilik et al. (2021).

Professional Considerations

Development of the system followed accepted software engineering standards with a focus on modularity, maintainability and error tolerance. Logging mechanisms, system state awareness and exception handling routines were implemented to ensure operational stability and ease of debugging. The project adheres to professional conduct principles promoted by computing societies such as the BCS and IEEE, emphasising the benefits to the public, as well as highlighting user safety and environmental responsibility. Furthermore, all components are built using open-source libraries (YOLOv8n, Flask, OpenCV), promoting transparency, reproducibility and community validation.

Chapter 8

Conclusion and Future Work

8.1 Summary of Achievements

The primary aim of this project, as outlined in Chapter 1, was to design, implement and evaluate a lightweight, real-time embedded system for detecting *Varroa destructor* on *Apis mellifera*. This objective was fully achieved through the successful development of a YOLOv8n-based detection pipeline integrated with a Raspberry Pi 5 and Hailo-8L accelerator, along with a functional web-based dashboard for visual monitoring.

Performance results presented in Chapter 6 demonstrated that YOLOv8n achieved a superior mAP@0.5 of 0.898 and a peak F1 score of 0.84, significantly outperforming the YOLOv8n-SO variant. The system maintained real-time inference performance (<100ms) and robustness across various visual conditions, validating the architectural design and hardware integration strategies laid out in Section **??**.

All stated objectives, including on-device inference, small object detection robustness and dashboard integration, were met. The system not only satisfied functional and non-functional requirements defined in Chapter 3.3, but also offered a practical deployment-ready solution validated under experimental constraints.

8.2 Contributions

This project contributes to the growing body of research in real-time agricultural AI systems by offering a full-stack, deployable architecture for *Varroa* mite detection. As discussed in the literature review (Chapter 2), many existing approaches focus either on accuracy benchmarks in controlled settings Bringas et al. (2023); Lee et al. (2024), or edge deployment without full integration Narcia-Macias et al. (2024); Mrozek et al. (2021). In contrast, this work delivers an end-to-end solution from dataset management and model optimisation, to Hailo-compatible deployment and real-time web interface development.

One of the key contributions is the evaluation of architectural trade-offs in YOLOv8 variants, particularly highlighting the role of P5 feature extraction in retaining detection fidelity for small objects. These findings, supported by instability in YOLOv8n-SO training curves (Section 6.8), reinforce prior critiques in Liu et al. (2023); Feng et al. (2024). The integration of the Hailo AI accelerator also sets a benchmark for low-power

embedded AI deployments in apiculture.

By bridging the gap between theoretical model design and practical field implementation, this project contributes a reusable design pattern for similar edge-AI systems in agriculture and precision ecology.

8.3 Future Work

Several avenues for future enhancement have emerged from this project. First, the dataset can be significantly improved by incorporating more field-collected images under varying environmental conditions, addressing the current limitation in lighting and hive diversity. Augmentation and synthetic data generation strategies could also be adopted to mitigate class imbalance and better represent edge cases.

A particularly impactful direction involves the creation of a custom dataset that preserves the true physical size ratio between bees and mites. This adjustment is expected to enhance the functional viability of the YOLOv8n-SO variant by aligning input resolution with the receptive field of small object branches. As noted in Chapter 6 and supported by Liu et al. (2023); Feng et al. (2024), resolution sensitivity introduced by the removal of the P5 layer significantly limits the model's performance on magnified instances. Interestingly, as seen in the visual output analysis (Section 6.5), the YOLOv8n-SO model tends to detect small *Varroa* instances more confidently while frequently ignoring larger or zoomed-in ones. This behaviour suggests that architectural modifications, when combined with attention mechanisms, can create narrow feature priors that require retraining with scale-consistent annotations.

In terms of architectural innovation, the detection pipeline could be extended using domain adaptation or ensemble methods to improve robustness across unseen hive conditions. Lightweight transformers or attention modules can also be explored, provided they remain compatible with real-time hardware constraints. This would further support small-object differentiation in high-clutter settings.

Another major consideration for future development involves broadening the scope of sensing modalities. Integrating temperature, humidity, weight, sound and multi-camera systems into the hive hardware could allow for real-time multi-modal data collection. The goal of this approach would be to monitor diverse environmental and biological features, observe their evolution over time, and study correlations between specific parameter combinations and *Varroa* proliferation events. By training predictive models on these multi-modal patterns, the system could evolve from an early detection tool into a predictive prevention platform. This would mark a significant step forward in proactive apicultural health management and position the system at the forefront of AI-driven environmental diagnostics.

Finally, real-world deployment remains critical. Usability studies and field validation with beekeepers would help fine-tune system thresholds, improve UI responsiveness and assess long-term reliability in uncontrolled environments. Such evaluations are essential to transitioning the prototype into a deployable product and establishing its viability in commercial or research applications.

In conclusion, the project has successfully demonstrated that edge-based deep learning

systems can provide accurate, real-time *Varroa* detection. It offers a robust architectural and methodological foundation for ongoing research, with immediate opportunities for improvement in dataset fidelity, sensory integration and predictive analytics. Together, these advancements would elevate the system from a responsive detection tool to a fully intelligent, anticipatory platform for sustainable beekeeping.

Appendix A

UML and Additional Diagrams

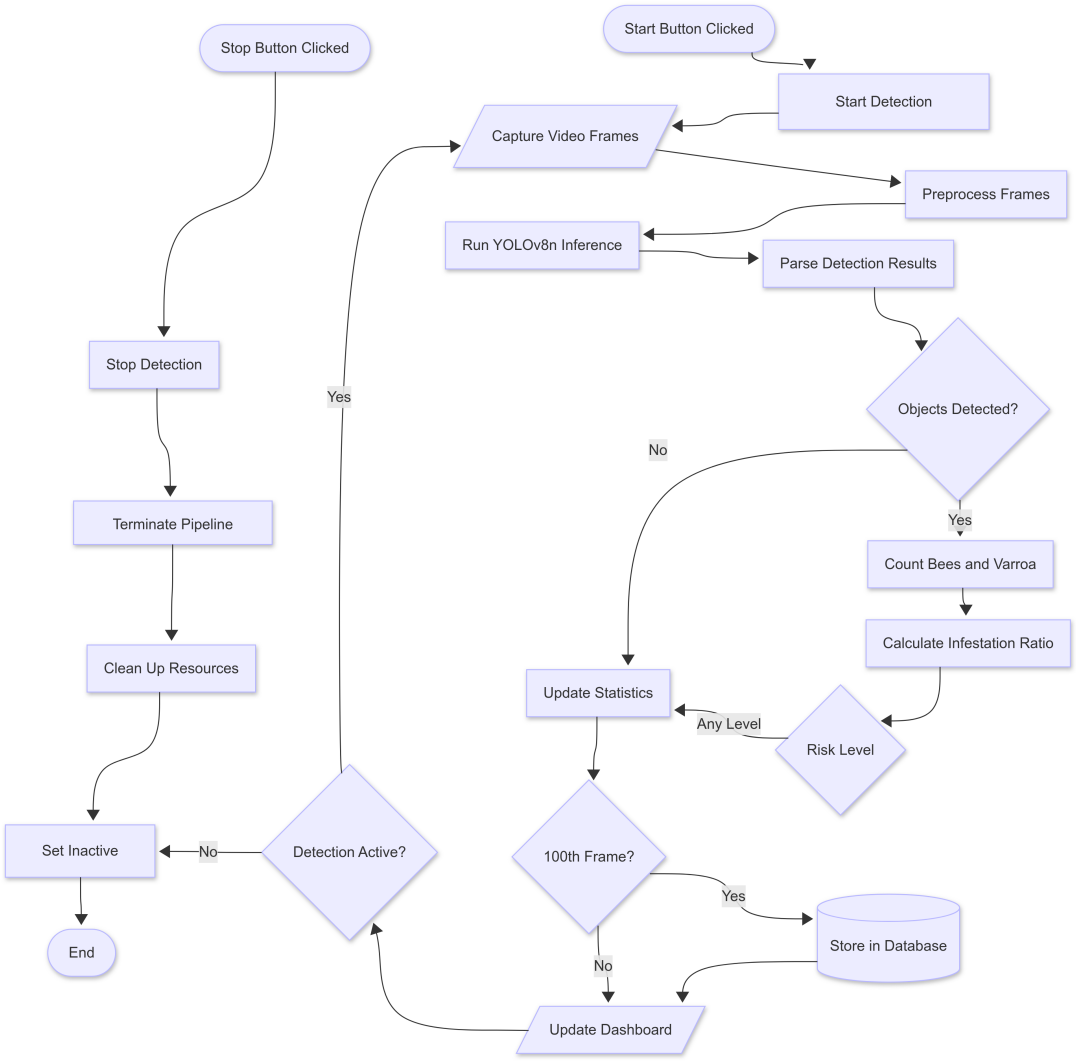

Figure A.1: Detection Process Flow Diagram illustrating the sequence from capture to processing and dashboard update.

Figure A.2: Sequence Diagram showing the interaction between system components during detection sessions.

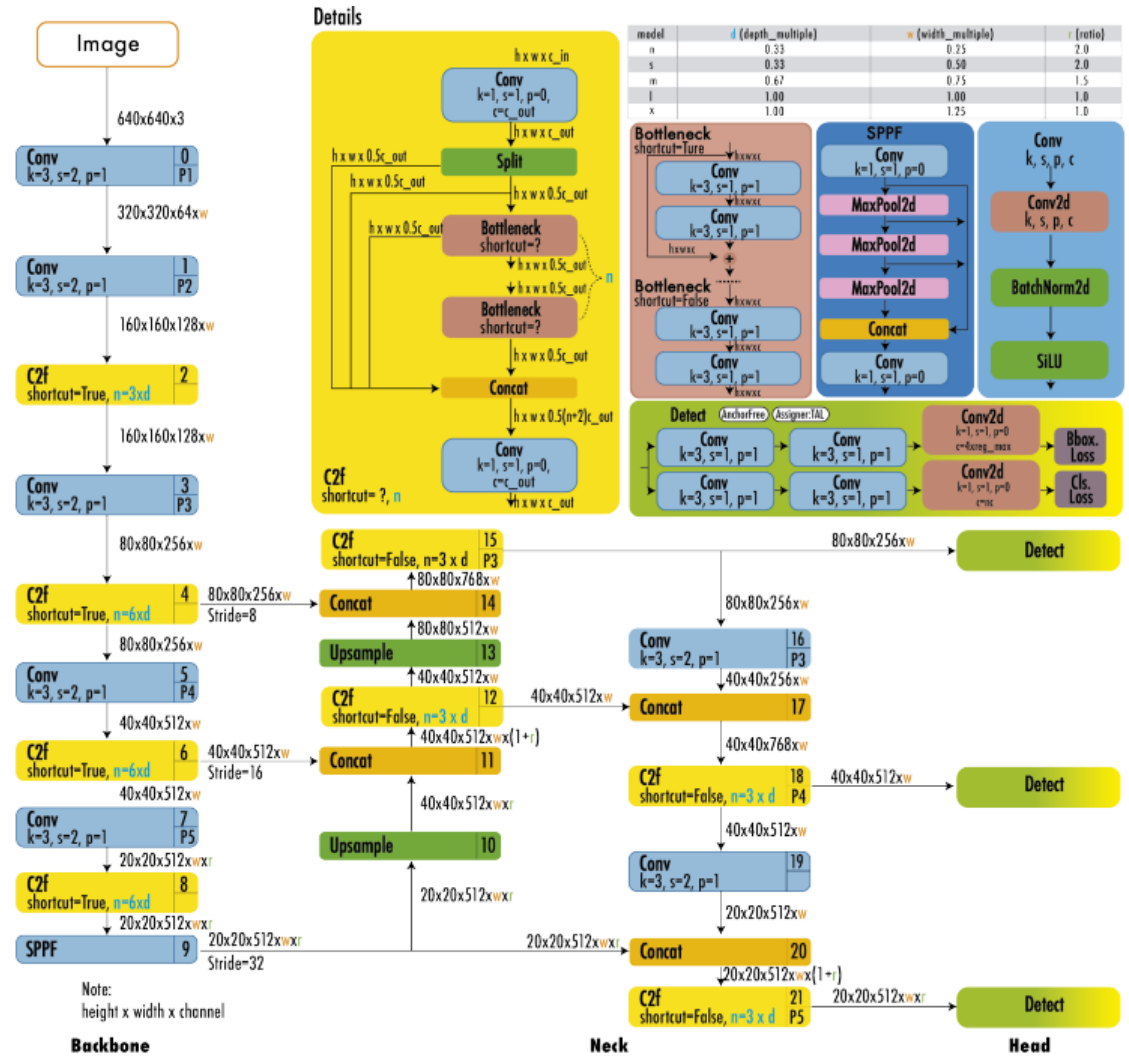

Figure A.3: YOLOv8 architecture.

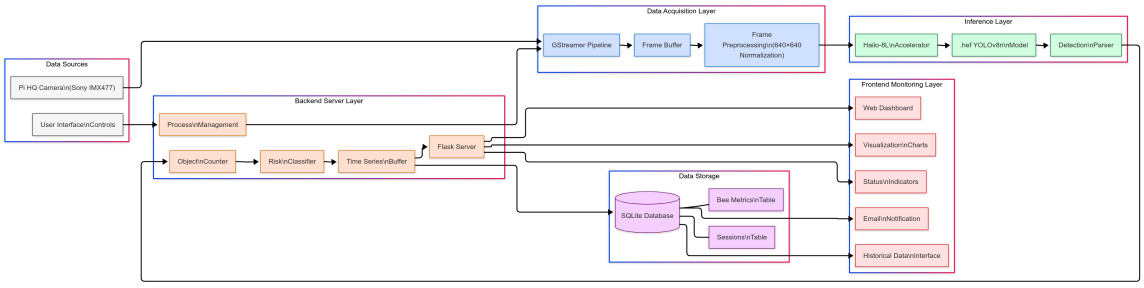

Figure A.4: High-level system architecture showing the layered pipeline from camera input to inference, server orchestration and user dashboard.

Appendix B

Code Samples and Configuration Details

B.1 YOLOv8n Training Pipeline Overview

Algorithm 1 This is a Pseudo-code showing: YOLOv8n Training Pipeline

1: **Input:** Custom `yolov8n.yaml`, curated dataset
2: Load model from `ultralytics/models/`
3: initialise training pipeline via `ultralytics/engine/train.py`
4: Apply data augmentations from `utils/augmentations.py`
5: Forward propagate through backbone and head
6: Compute CIoU and BCE loss
7: Monitor validation metrics and apply early stopping
8: Export final weights to ONNX format
9: Compile ONNX to Hailo `.hef` using Hailo Model Zoo
10: **Output:** Compiled model ready for deployment

B.2 YOLOv8n-SO Architecture Modification Process

Algorithm 2 This is a Pseudo-code showing: YOLOv8n-SO Architecture Modifications

Input: Original YOLOv8n architecture
Output: Small object optimised YOLOv8n-SO architecture
// **Backbone Modifications**
for all layer in backbone **do**
    if layer.type == "C2f" **then**
        Replace with `C2fCBAMv2` using same parameters
    end if
end for
// **Detection Head Modifications**
Add high-resolution P2 detection head at resolution $(H/4, W/4)$
Upsample from P3 to P2 and downsample from P1 to P2
Concatenate: $[P1_{down}, P2, P3_{up}]$
Process with `C2fCBAMv2`
Remove P5 detection head and update Detect layer to use only $[P2, P3, P4]$
// **Feature Fusion Enhancement**
for all feature level $l \in \{P2, P3, P4\}$ **do**
    Collect adjacent resolution features
    Resample all to match l
    Concatenate all at level l
    Process with `C2fCBAMv2`
end for
// **CBAM Attention Integration**
Define `C2fCBAMv2` as extension of `C2f`
Apply channel attention, then spatial attention to feature maps

B.3 Hailo Compilation and Deployment Algorithms

Algorithm 3 This is a Pseudo-code showing: The complete Workflow for Hailo 8L Deployment

Step 1: Export YOLOv8 Model to ONNX Format

```
yolo export model=best.pt imgsz=640 format=onnx opset=11
```

Step 2: Hailo SDK Environment Setup (WSL2 Ubuntu)

Create Python 3.8 virtual environment:

```
python3.8 -m venv venv_hailo
```

Activate the environment:

```
source venv_hailo/bin/activate
```

Install Hailo Dataflow Compiler and Model Zoo:

```
pip install hailo_dataflow_compiler*.whl
pip install hailo_model_zoo*.whl
```

Step 3: Quantisation-Aware optimisation

Generate calibration TFRecord dataset:

```
python steps/2_install_dataset/create_custom_tfrecord.py train
```

Parse ONNX model for quantisation:

```
python steps/3_process/parse.py
```

optimise parsed model using calibration data:

```
python steps/3_process/optimise.py
```

Step 4: Compile Model to Hailo Executable Format (.hef)

Compile the quantised model:

```
python steps/3_process/compile.py
```

Step 5: Deploy and initialise Real-Time Inference

Transfer compiled `.hef` file to Raspberry Pi 5.

Navigate to Hailo inference example directory:

```
cd hailo-rpi5-examples
```

Activate Hailo environment:

```
source setup_env.sh
```

Start inference with detection script:

```
python detection.py -i rpi --hef best_quantised_model.hef
--labels-json labels.json
```

B.4 Frontend Development Algorithms

Algorithm 4 This is a Pseudo-code showing: HTML Structure Implementation of the Dashboard

```
Create dashboard-container as root element
    Add dashboard-header with row layout
        Left column:
            Add title "Bee Colony Health Monitor"
            Add badge labeled "PRO"
            Add subtitle "Real-time Varroa Mite Detection & Analysis"
        Right column:
            Add Start Detection button with icon
Create card container for "Status Overview"
    Within card-body, create row with column layout
        Add Colony Health Status heading
        Add statusBadge element initialised to "Unknown"
Reserve space for additional dashboard components
```

Algorithm 5 This is a Pseudo-code showing: CSS Variable System Implementation

```
Define root CSS variables using :root
    Set primary color palette: gold, light gold, dark gold
    Define secondary colors and alert colors (success, warning, danger)
    Declare font stacks for headings, body, and monospace
    Establish spacing and border radius units for consistency
```

Algorithm 6 This is a Pseudo-code showing: Value Animation Functionality

```
procedure AnimateValueChange(selector, newValue)
    element ← GetElementBySelector(selector)
    currentValue ← ParseFloat(element.text)
    targetValue ← ParseFloat(newValue)
    if currentValue ≠ targetValue then
        Animate value from currentValue to targetValue over 800ms
        for all step in animation do
            if targetValue is integer then
                element.text ← floor(stepValue)
            else
                element.text ← roundToTwoDecimals(stepValue)
            end if
        end for
        Set element.text ← newValue
    end if
end procedure
```

Algorithm 7 This is a Pseudo-code showing: Detection Button Animation

```
1: function setButtonState(isActive)
2:     Disable button temporarily
3:     if isActive then
4:         Fade out
5:         Switch to "Stop" style (btn-danger, stop icon)
6:         Enable and fade in
7:     else
8:         Fade out
9:         Switch to "Start" style (btn-success, play icon)
10:        Enable and fade in
11:    end if
12: end function
```

B.5 Backend Algorithms

B.5.1 Backend API Endpoints

Algorithm 8 This is a Pseudo-code showing: Detection API Endpoint Handling

```
1: function startDetection
2:     if detection is not active then
3:         Terminate existing process; Cleanup GStreamer; Sleep briefly
4:         Start detection thread and Return status: started
5:     else
6:         Return status: already running
7:     end if
8: end function
9: function stopDetection
10:    if detection is active then
11:        Set detection to inactive; Terminate and cleanup
12:        Return status: stopped
13:    else
14:        Return status: already stopped
15:    end if
16: end function
```

Algorithm 9 This is a Pseudo-code showing: Statistics and Time Series Retrieval

```
1: function getStats
2:     Return current stats as JSON
3: end function
4: function getTimeSeries
5:     Extract values from timestamped queues; Format as lists
6:     Return as JSON object
7: end function
```

Algorithm 10 This is a Pseudo-code showing: Database API Endpoints

```
function getSessions(limit=10)
    Fetch sessions from DB with limit; Return as JSON
end function
function getMetrics(limit=100, sessionId)
    Fetch latest metrics optionally filtered by session
    Return as JSON
end function
```

B.5.2 Data Processing Algorithms

Algorithm 11 This is a Pseudo-code showing: Detection Output Parsing

```
procedure ParseDetectionOutput(line)
    if line contains "Unique bees:" then
        Try: Extract number after "Unique bees:"
            Set unique_bees and total_bees in stats dictionary
        Catch parsing error: If DEBUG is enabled, print error message
    end if
    if line contains "Current frame bees:" then
        Try: Extract number after "Current frame bees:"
            Set current_bees in stats dictionary
        Catch parsing error: If DEBUG is enabled, print error message
    end if
    if line contains "Frame count:" then
        Try: Extract number after "Frame count:"
            Set total_frames in stats dictionary
        if frame count mod 10 equals 0 then
            Call update_time_series()
        end if
        Catch parsing error: If DEBUG is enabled, print error message
    end if
end procedure
```

B.5.3 Time Series Algorithms

Algorithm 12 This is a Pseudo-code showing: Time Series Update Function

```
1: procedure UpdateTimeSeries
2:     Get current timestamp in HH:MM:SS format
3:     Append timestamp to time_series_data["timestamps"]
4:     Append current bee count to time_series_data["bee_counts"]
5:     Append current varroa count to time_series_data["varroa_counts"]
6:     if unique bee count > 0 then
7:         Calculate infestation ratio = unique varroa / unique bees
8:     else
9:         Set infestation ratio to 0
10:    end if
11:    Append infestation ratio to time_series_data["infestation_ratio"]
12:    Update detection_stats["infestation_ratio"]
13:    if ratio < low threshold then
14:        Set risk level to "Low"
15:    else if ratio < moderate threshold then
16:        Set risk level to "Moderate"
17:    else if ratio < high threshold then
18:        Set risk level to "High"
19:    else
20:        Set risk level to "Critical"
21:    end if
22:    if total frame count mod 100 == 0 then
23:        Save metrics to database: unique bees, unique varroa, frame count
24:    end if
25: end procedure
```

B.5.4 Email Notification System Algorithms

Algorithm 13 This is a Pseudo-code showing: Email Notification Process

```
1: function SendSessionSummary(sessionId, dbPath)
2:     Retrieve session data and metrics from database
3:     Calculate overall infestation ratio from unique bee and varroa counts
4:     Determine risk level based on defined threshold values
5:     Generate formatted HTML email with:
6:         Session metadata (ID, source, timestamps, duration)
7:         Color-coded colony health status indicator
8:         Detection summary with bee and varroa counts
9:         Infestation ratio with risk assessment
10:        Recent metrics table with timestamp-based trends
11:    Generate plain text alternative for compatibility
12:    Establish secure SMTP connection with timeout protection
13:    Send multipart email with both content formats
14:    Update session record with email delivery status
15:    Return success or failure status
16: end function
```

Appendix C

Additional Results and Testing Data

C.1 Unit Testing Outputs

```python
def test_get_stats_route(client, reset_stats):
    # Set test data
    from app import detection_stats
    detection_stats.update({
        "current_bees": 10,
        "current_varroa": 2,
        "infestation_ratio": 0.16,
        "infestation_risk_level": "High"
    })

    # Test the endpoint
    response = client.get('/get_stats')
    assert response.status_code == 200

    # Verify response data
    data = json.loads(response.data)
    assert data["current_bees"] == 10
    assert data["infestation_risk_level"] == "High"
```

Figure C.1: Example unit test for the `/get_stats` endpoint.

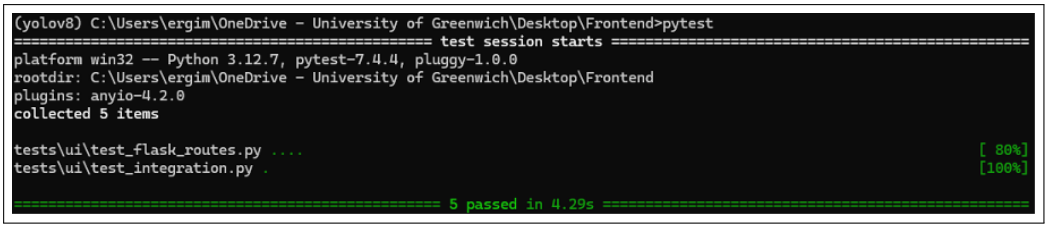

Figure C.2: Backend unit test results showing 80% route coverage and 100% for integration components.

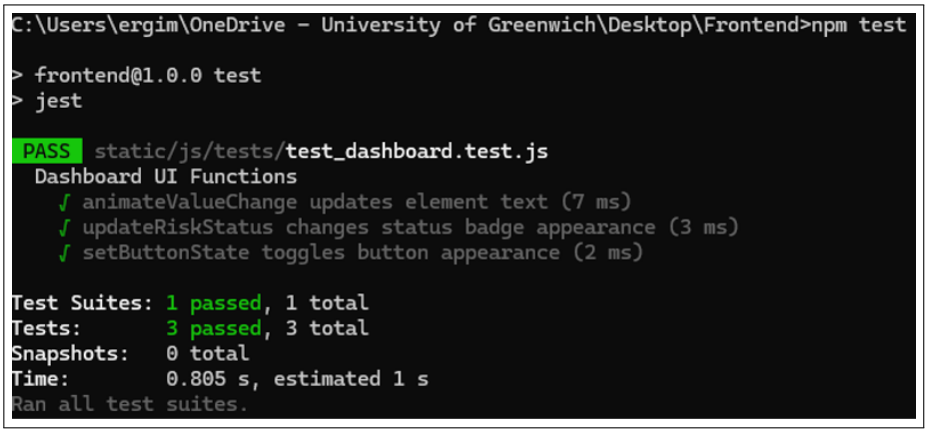

Figure C.3: JavaScript unit test results confirming UI component responsiveness.

C.2 Integration Testing Outputs

```python
def test_detection_workflow_integration(client, reset_stats):
    # Mock the detection process
    with patch('app.subprocess.Popen', return_value=mock_process):
        # Start detection
        response = client.post('/start_detection')
        assert response.status_code == 200

        # Simulate detection updates
        detection_stats.update({
            "current_bees": 15,
            "current_varroa": 3,
            "infestation_ratio": 0.2,
            "infestation_risk_level": "High"
        })

        # Verify dashboard data updates
        stats_response = client.get('/get_stats')
        assert stats_data["current_bees"] == 15

        # Stop detection
        stop_response = client.post('/stop_detection')
        assert stop_data["status"] == "stopped"
```

Figure C.4: Mocked integration test for detection workflow.

C.3 Performance Testing Outputs

```
def test_average_fps_per_session():
    conn = sqlite3.connect("bee_health.db")
    cursor.execute("""
        SELECT session_id,
               ROUND(AVG(fps), 2) as avg_fps,
               ROUND(MIN(fps), 2) as min_fps,
               ROUND(MAX(fps), 2) as max_fps,
               COUNT(*) as total_frames
        FROM bee_metrics
        WHERE fps IS NOT NULL
        GROUP BY session_id
        ORDER BY session_id DESC
        LIMIT 5
    """)
```

Figure C.5: SQL query extracting session-wise FPS metrics from `bee_health.db`.

```
(yolov8) C:\Users\ergim\OneDrive - University of Greenwich\Desktop\Frontend\tests\ui>pytest test_performance.py -s
============================= test session starts =============================
platform win32 -- Python 3.12.7, pytest-7.4.4, pluggy-1.0.0
rootdir: C:\Users\ergim\OneDrive - University of Greenwich\Desktop\Frontend\tests\ui
plugins: anyio-4.2.0
collected 1 item

test_performance.py
 Performance Testing Summary (Last 5 Sessions):
Session 47: Avg FPS = 29.2, Min FPS = 6.89, Max FPS = 47.4, Frames = 18
Session 46: Avg FPS = 29.31, Min FPS = 6.89, Max FPS = 47.08, Frames = 14
Session 45: Avg FPS = 26.83, Min FPS = 7.24, Max FPS = 37.4, Frames = 6
Session 44: Avg FPS = 26.96, Min FPS = 7.21, Max FPS = 37.55, Frames = 9
Session 43: Avg FPS = 31.03, Min FPS = 7.23, Max FPS = 65.6, Frames = 43
.

============================== 1 passed in 0.02s ==============================
```

Figure C.6: Performance metrics across five sessions.

Table C.1: Session Performance Metrics Summary

Session ID	Average FPS	Minimum FPS	Maximum FPS	Frame Count
47	29.20	6.89	47.40	18
46	29.31	6.89	47.08	14
45	26.83	7.24	37.40	6
44	26.96	7.21	37.55	9
43	31.03	7.23	65.60	43

Appendix D

Dashboard Screenshots

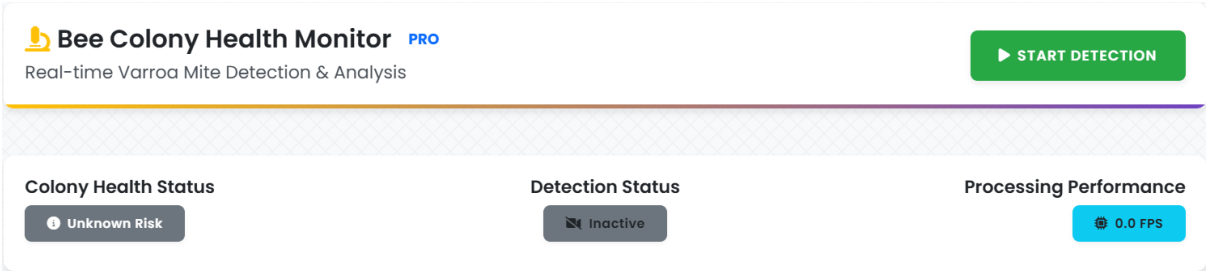

Figure D.1: Dashboard HTML structure showing the header, control panel and status overview section.

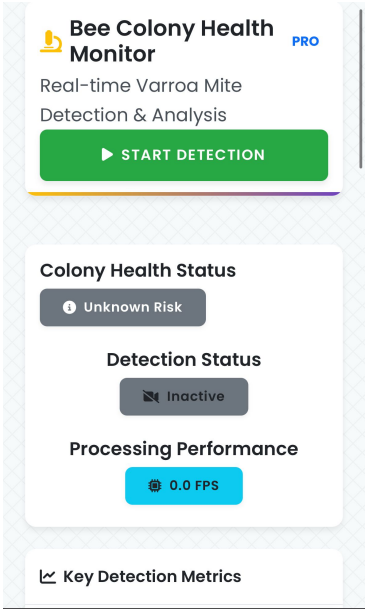

Figure D.2: Mobile view of the dashboard on a smartphone.

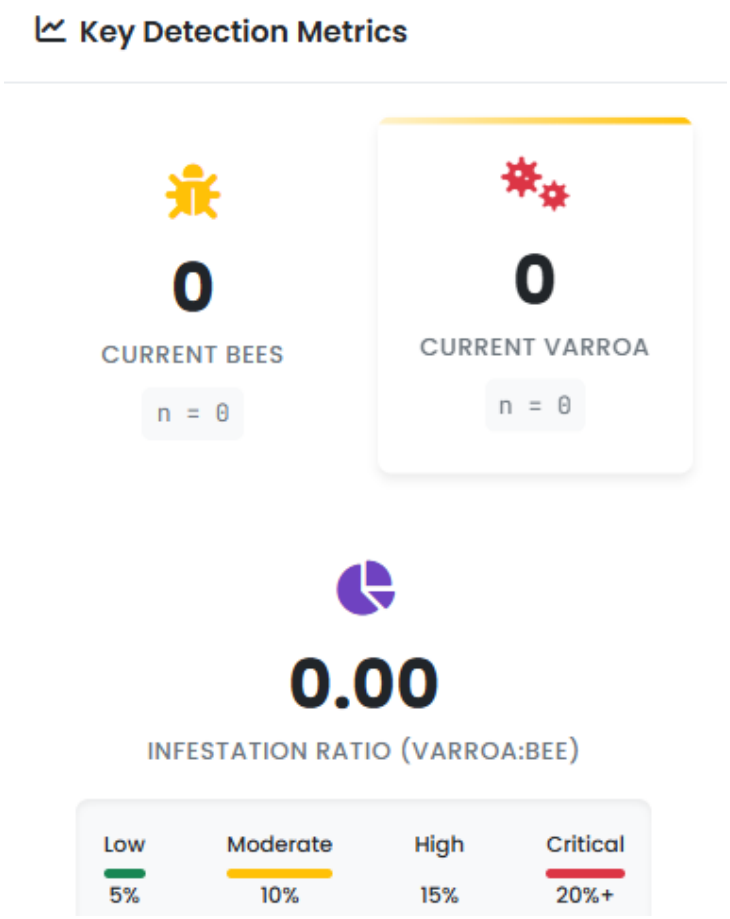

Figure D.3: Styling of detection metrics with color-coded risk levels.

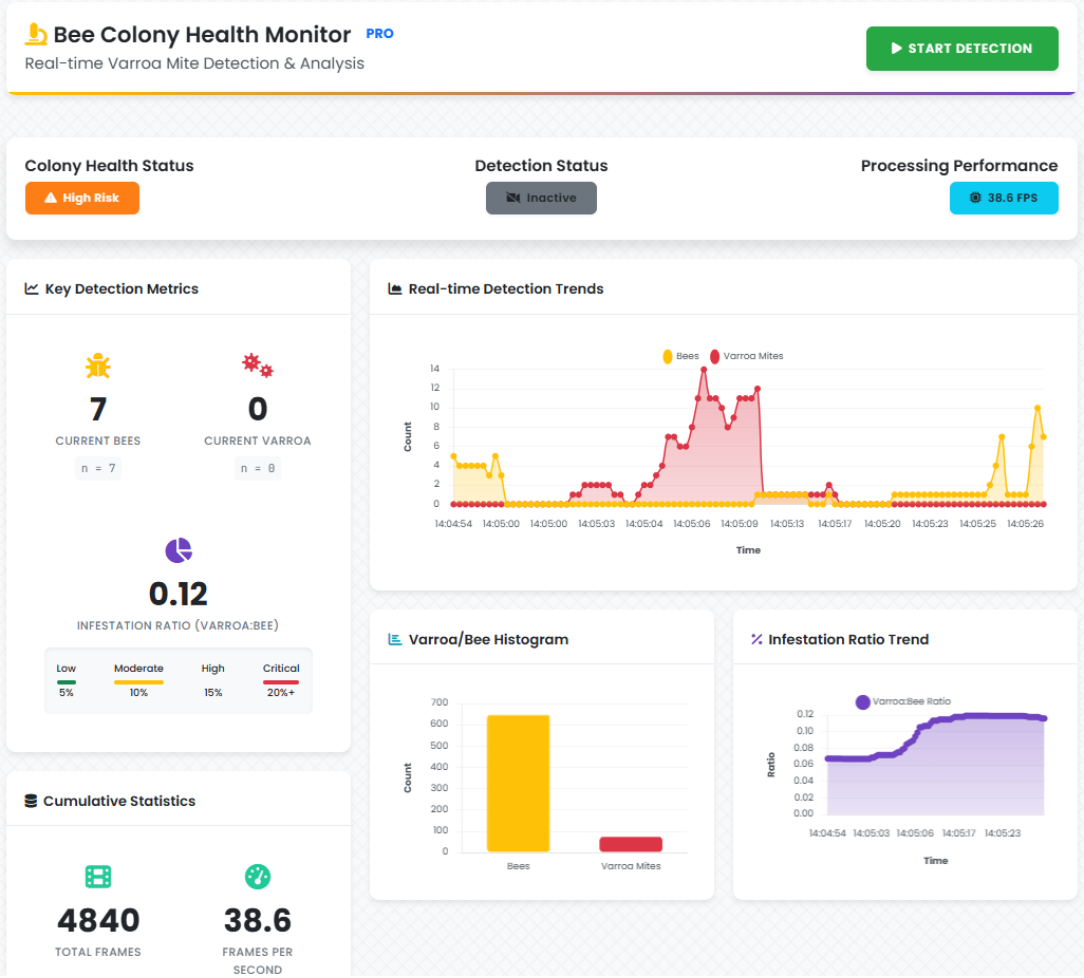

Figure D.4: Animated transition of metric values using JavaScript step functions.

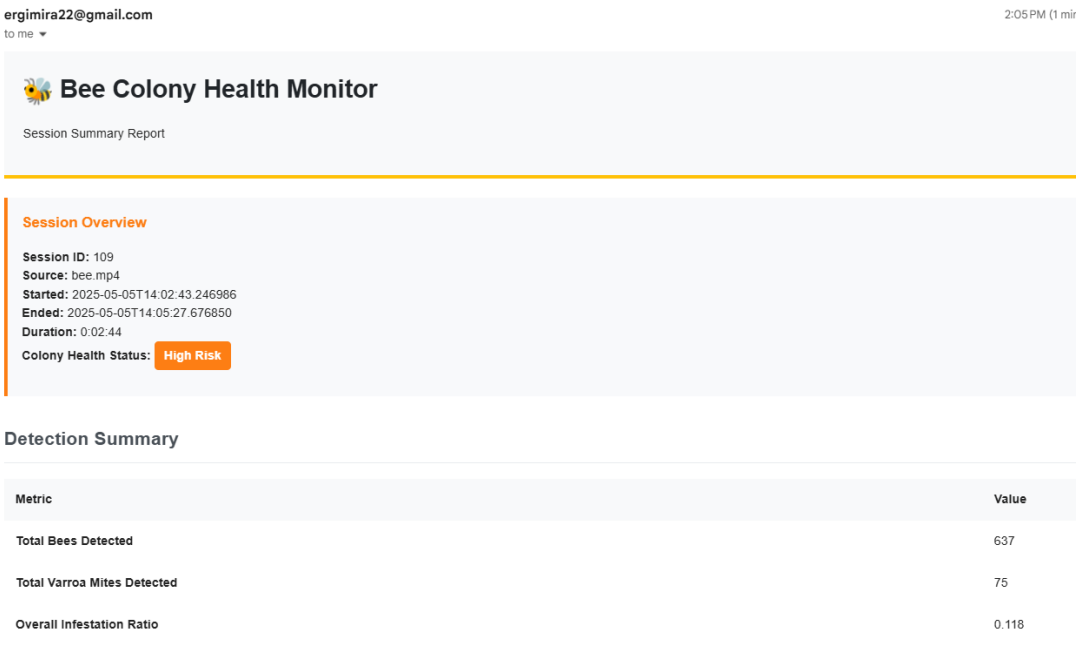

Figure D.5: The detection summary email notification.

List of References

Achmadiah, M. N., Ahamad, A., Sun, C.-C. and Kuo, W.-K. (2025), 'Energy-efficient fast object detection on edge devices for iot systems', *IEEE Internet of Things Journal* . Click here for the link.

Ajay, K., Sundaramoorthi, A. and Mary, S. P. (2025), Optimization of transportation traffic using agile approach, *in* '2025 International Conference on Electronics and Renewable Systems (ICEARS)', IEEE, pp. 1683–1692. Click here for the link.

Alice (2025*a*), 'Bee detection dataset [v14]'. Click here for the link.

Alice (2025*b*), 'Beehive dataset [v7]'. Click here for the link.

Alves, A. and Hora, A. (2024), Testmigrationsinpy: A dataset of test migrations from unittest to pytest, *in* 'Proceedings of the International Conference on Mining Software Repositories', IEEE. Click here for the link.

Bee (2025), 'Beehive dataset [v5]'. Click here for the link.

Berkaya, S. K., Gunal, E. S. and Gunal, S. (2021), 'Deep learning-based classification models for beehive monitoring', *Ecological Informatics* **64**, 101353. Click here for the link.

Bilik, S., Kratochvila, L., Ligocki, A., Bostik, O., Zemcik, T., Hybl, M., Horak, K. and Zalud, L. (2021), 'Visual diagnosis of the varroa destructor parasitic mite in honeybees using object detector techniques', *Sensors* **21**(8), 2764. Click here for the link.

Bjerge, K., Frigaard, C. E., Mikkelsen, P. H., Nielsen, T. H., Misbih, M. and Kryger, P. (2019), 'A computer vision system to monitor the infestation level of varroa destructor in a honeybee colony', *Computers and Electronics in Agriculture* **164**, 104898. Click here for the link.

Bolo (2025), 'Beehive detection dataset [v2]'. Click here for the link.

Bringas, P. G., García, H. P., de Pisón, F. J. M., Álvarez, F. M., Lora, A. T., Herrero, Á., Rolle, J. L. C., Quintián, H. and Corchado, E. (2023), *Hybrid Artificial Intelligent Systems: 18th International Conference, HAIS 2023, Salamanca, Spain, September 5–7, 2023, Proceedings*, Vol. 14001, Springer Nature. Click here for the link.

Calvo, H., Mart'ınez-Villase nor, L., Ponce, H., Cabada, R. Z., Rivera, M. M. and Mezura-Montes, E. (2024), *Advances in Computational Intelligence. MICAI 2023 International Workshops: WILE 2023, HIS 2023, and CIAPP 2023, Yucat'an, Mexico, November 13–18, 2023, Proceedings*, Vol. 14502, Springer Nature. Click here for the link.

Divas'on, J., Romero, A., Martinez-de Pison, F. J., Casalongue, M., Silvestre, M. A.,

Santolaria, P. and Y'aniz, J. L. (2024), 'Analysis of varroa mite colony infestation level using new open software based on deep learning techniques', *Sensors* **24**(12), 3828. Click here for the link.

Feng, F., Hu, Y., Li, W. and Yang, F. (2024), 'Improved yolov8 algorithms for small object detection in aerial imagery', *Journal of King Saud University-Computer and Information Sciences* **36**(6), 102113. Click here for the link.

Feng, H., Mu, G., Zhong, S., Zhang, P. and Yuan, T. (2022), 'Benchmark analysis of yolo performance on edge intelligence devices', *Cryptography* **6**(2), 16. Click here for the link.

Fezari, M. and Al-Dahoud, A. (2023), 'Raspberry pi 5: The new raspberry pi family with more computation power and ai integration'. Click here for the link.

Garc'ıa-Vicente, E. J., Benito-Murcia, M., Mart'ın Dom'ınguez, M., P'erez P'erez, A., Gonz'alez S'anchez, M., Rey-Casero, I., Alonso Rodr'ıguez, J. M., Barquero-P'erez, O. and Risco P'erez, D. (2024), 'Main causes of producing honey bee colony losses in southwestern spain: a novel machine learning-based approach', *Apidologie* **55**(5), 67. Click here for the link.

Goh, H.-A., Ho, C.-K. and Abas, F. S. (2023), 'Front-end deep learning web apps development and deployment: a review', *Applied intelligence* **53**(12), 15923–15945. Click here for the link.

Hall, H., Bencsik, M. and Newton, M. (2023), 'Automated, non-invasive varroa mite detection by vibrational measurements of gait combined with machine learning', *Scientific Reports* **13**(1), 10202. Click here for the link.

Harshavardhan, T., Vasisht, N., Praneeth, M., Deepika, N., Hariharan, S. and Kukreja, V. (2024), Web application for identifying emerging trends and technologies using scopus and openai apis, *in* '2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)', IEEE, pp. 1–4. Click here for the link.

Ilieva, S., Ivanov, P. and Stefanova, E. (2004), Analyses of an agile methodology implementation, *in* 'Proceedings. 30th Euromicro Conference, 2004.', IEEE, pp. 326–333. Click here for the link.

Iqbal, M. Z., Arcuri, A. and Briand, L. (2011), Automated system testing of real-time embedded systems based on environment models, Technical Report 2011-19, Simula Research Laboratory. Click here for the link.

Jin, Z. (2020), Design and implementation of full-stack testing for web spa in javascript, Master's thesis, Aalto University, School of Science, Otaniemi, Finland. Click here for the link.

Konig, A. (2021), 'First results of the bee-nose on mid-term duration hive air monitoring for varroa infestation level estimation', *Sensors Transducers* **250**(3), 39–43. Click here for the link.

Konig, A. (2022), 'An in-hive soft sensor based on phase space features for varroa infestation level estimation and treatment need detection', *Journal of Sensors and Sensor Systems* **11**(1), 29–40. Click here for the link.

Kriouile, Y. (2022), Detection of Nested Objects in Dense Scenes using Deep Learning-Application to Bee and Varroa Detection, PhD thesis, Universit'e Paris sciences et lettres. Click here for the link.

Kriouile, Y., Ancourt, C., Wegrzyn-Wolska, K. and Bougueroua, L. (2024), 'Nested object detection using mask r-cnn: application to bee and varroa detection', *Neural Computing and Applications* **36**(35), 22587–22609. Click here for the link.

Krispin-Avraham, I., Orfaig, R. and Bobrovsky, B.-Z. (2024), 'Real-time 3d object detection using innovizone lidar and low-power hailo-8 ai accelerator', *arXiv preprint arXiv:2412.05594* . Click here for the link.

Kumar, G. and Bhatia, P. K. (2012), 'Impact of agile methodology on software development process', *International Journal of Computer Technology and Electronics Engineering (IJCTEE)* **2**(4), 46–50. Click here for the link.

Lee, Y., Cho, H., Kim, B.-Y. and Moon, J. (2024), 'A yolov8-based two-stage framework for non-destructive detection of varroa destructor infestations in apis mellifera colonies', *Journal of the Korea Society of Computer and Information* **29**(10), 137–148. Click here for the link.

Liu, M., Cui, M., Xu, B., Liu, Z., Li, Z., Chu, Z., Zhang, X., Liu, G., Xu, X. and Yan, Y. (2023), 'Detection of varroa destructor infestation of honeybees based on segmentation and object detection convolutional neural networks', *AgriEngineering* **5**(4), 1644–1662. Click here for the link.

Longbottom, R. (2019), 'Raspberry pi 5 64 bit benchmarks and stress tests', *Longbottom Performance Test Reports* . Click here for the link.

Mika, K., Griessl, R., Kucza, N., Porrmann, F., Kaiser, M., Tigges, L., Hagemeyer, J., Trancoso, P., Azhar, M. W., Qararyah, F. et al. (2023), Vedliot: Next generation accelerated aiot systems and applications, *in* 'Proceedings of the 20th ACM International Conference on Computing Frontiers', pp. 291–296. Click here for the link.

Mrozek, D., Grny, R., Wachowicz, A. and Małysiak-Mrozek, B. (2021), 'Edge-based detection of varroosis in beehives with iot devices with embedded and tpu-accelerated machine learning', *Applied Sciences* **11**(22), 11078. Click here for the link.

Narcia-Macias, C. I., Guardado, J., Rodriguez, J., Park, J., Rampersad-Ammons, J., Enriquez, E. and Kim, D.-C. (2024), Intellibeehive: An automated honey bee, pollen, and varroa destructor monitoring system, *in* '2024 International Conference on Machine Learning and Applications (ICMLA)', IEEE, pp. 845–850. Click here for the link.

Nguyen, H. (2022), 'Single-page application and front-end testing methods – built with react and react router, tested with jest and cypress'. Click here for the link.

Picek, L., Novozamsky, A., Frydrychova, R. C., Zitova, B. and Mach, P. (2022), Monitoring of varroa infestation rate in beehives: A simple ai approach, *in* '2022 IEEE International Conference on Image Processing (ICIP)', IEEE, pp. 3341–3345. Click here for the link.

Sary, I. P., Andromeda, S. and Armin, E. U. (2023), 'Performance comparison of yolov5 and yolov8 architectures in human detection using aerial images', *Ultima Computing: Jurnal Sistem Komputer* **15**(1), 8–13. Click here for the link.

Schurischuster, S. (2018), Image analysis approaches for parasite detection on honeybees, PhD thesis, Wien. Click here for the link.

Sharma, A., Varastehpour, S., Ardekani, I. and Sharifzadeh, H. (2024), Bee disease varroa prediction: Utilizing convolutional neural networks with augmentation for robust detection and identification of honeybee infection, *in* '2024 1st International Conference on Innovative Engineering Sciences and Technological Research (ICIESTR)', IEEE, pp. 1–6. Click here for the link.

Solimani, F., Cardellicchio, A., Dimauro, G., Petrozza, A., Summerer, S., Cellini, F. and Renò, V. (2024), 'Optimizing tomato plant phenotyping detection: Boosting yolov8 architecture to tackle data complexity', *Computers and Electronics in Agriculture* **218**, 108728. Click here for the link.

Terven, J., C'ordova-Esparza, D.-M. and Romero-Gonz'alez, J.-A. (2023), 'A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas', *Machine learning and knowledge extraction* **5**(4), 1680–1716. Click here for the link.

Tidwell, T. T. (2001), 'Wilhelm schlenk: the man behind the flask', *Angewandte Chemie International Edition* **40**(2), 331–337. Click here for the link.

Turyagyenda, A., Katumba, A., Akol, R., Nsabagwa, M. and Mkiramweni, M. E. (2025), 'Iot and machine learning techniques for precision beekeeping: A review', *AI* **6**(2), 26. Click here for the link.

Varghese, R. and Sambath, M. (2024), Yolov8: A novel object detection algorithm with enhanced performance and robustness, *in* '2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)', IEEE, pp. 1–6. Click here for the link.

Wachowicz, A., Pytlik, J., Małysiak-Mrozek, B., Tokarz, K. and Mrozek, D. (2022), 'Edge computing in iot-enabled honeybee monitoring for the detection of varroa destructor', *International Journal of Applied Mathematics and Computer Science* **32**(3). Click here for the link.

Wang, A. and Zhang, E. (2024), Assessment of novel acaricides to counter european honeybee memory loss and varroa destructor resistance using image recognition and artificial intelligence approach, *in* '2024 IEEE 3rd International Conference on Computing and Machine Intelligence (ICMI)', IEEE, pp. 1–5. Click here for the link.

Yang, G., Wang, J., Nie, Z., Yang, H. and Yu, S. (2023), 'A lightweight yolov8 tomato detection algorithm combining feature enhancement and attention', *Agronomy* **13**(7), 1824. Click here for the link.

Yazdani-Asrami, M., Sadeghi, A., Song, W., Madureira, A., Murta-Pina, J., Morandi, A. and Parizh, M. (2022), 'Artificial intelligence methods for applied superconductivity: material, design, manufacturing, testing, operation, and condition monitoring', *Superconductor Science and Technology* **35**(12), 123001. Click here for the link.