APPLICATIONS OF
MODELLING AND SIMULATION
http://arqiipubl.com/ams eISSN 2600-8084 VOL 10, 2026, 213-225

ReLeaf: A Mobile Application for Solanaceous Crop Disease Detection Using Lightweight YOLOv8n

Adha Rizwan[1], A. R. Syafeeza[1,*], Asar Khan[1] and Shahid Rahman[2]

[1]Faculty of Electronics and Computer Technology and Engineering, Universiti Teknikal Malaysia Melaka (UTeM), Hang Tuah Jaya, 76100 Durian Tunggal, Melaka, Malaysia
[2]Department of Computer Science, University of Buner, Pakistan

[*]Corresponding author: syafeeza@utem.edu.my

Submitted 21 August 2025; Revised 09 March 2026; Accepted 01 August 2026; Available online 25 August 2026.

Abstract: Solanaceous crops such as tomatoes, eggplants, potatoes, and peppers are highly susceptible to visually similar diseases, making accurate manual diagnosis difficult and error-prone. This study presents *ReLeaf*, a mobile application that leverages a lightweight deep learning model (YOLOv8n) for on-device plant disease detection in solanaceous crops. The trained YOLOv8n model was converted into multiple deployment formats (PyTorch, ONNX, TFLite), benchmarked, and tested for inference accuracy, latency, and performance across platforms. Experimental evaluation on test data achieved a mean Average Precision (mAP) of 98.7%, with PyTorch providing the best balance of accuracy and efficiency. However, when applied to real-world samples, the deployed model achieved an overall detection accuracy of 70.2%, highlighting the gap between controlled validation and field performance. The final prototype, built with Flutter SDK, integrated the PyTorch model with a user-friendly interface to deliver real-time detection and treatment recommendations. Field validation confirmed the feasibility of mobile deployment with acceptable latency, while also identifying limitations in distinguishing visually similar leaf diseases. This study demonstrates a practical pathway for deploying AI-based crop disease diagnostics on smartphones, with potential for scaling to other crops and improved robustness through future dataset expansion and model optimization.

Keywords: Disease diagnostics; Deep learning; Mobile deployment; Plant disease; Solanaceous crops.

1. INTRODUCTION

Solanaceous crops, such as tomatoes, peppers, and eggplants, are essential to global agriculture due to their economic and nutritional value. However, they are vulnerable to pests, diseases, and abiotic stress, necessitating farming solutions. Conventionally this is done by farmers manually identifying crop ailments, then performing treatment to remedy the ill health of said crops [1]. Historically, manual diagnosis has long supported steady crop production but remains susceptible to human error. This is especially apparent with crop diseases displaying similar visible traits, where human error falls when dealing with the complex specificity of identifying crop diseases. Human assessments can lead to misdiagnosis due to subjective interpretation of symptoms [2]. Figure 1 shows the likeness of three crop diseases with identical characteristics that can be misidentified as each other to the human eye.

In line with the vision of reducing such human errors, Artificial Intelligence in general has shown potential and precedence as an apt method of significantly enhancing agricultural practices. These applications range from disease prediction to irrigation management. Neural networks have been employed as accurate disease detection modes, facilitating early interventions and reduced crop losses [2]. Deep learning (DL) technology is particularly vital for analyzing agricultural datasets. For example, the DL model architecture of Convolution Neural Networks (CNNs) has been used to classify leaf diseases with over 90% accuracy [3]. However, specifically within DL, the subset of Object Detection frameworks would be more practical in field deployment scenarios, with its capacity to not just identify but also localize the faults.

Despite the potential of DL, it remains a method hurdled with requiring high device performance, thus barring ubiquitous integration of DL in the many facets of life. Conversely, smartphones have global ubiquity and are among the fastest-adopted technologies globally, with significant penetration in both developed and developing regions [4]. Thus, smartphones are the most likely deployable direct-to-user edge device. Nowadays, with the rapid adoption of Artificial Intelligence, many model architectures have been developed to fit with the growing number of presumed use cases, including deployment on edge devices. Among such many object detection DL model architectures, YOLO has shown promise of accuracy with real-time capabilities, with YOLOv8n being one of the latest variants with proven stability and accommodation for low performance, much suited for mobile use.

Figure 1. (a) Bacterial leaf spot, (b) Late blight, (c) Early blight.

Research has been trending with more adoption of DL in farming; numerous authors show a jarring lack of documentation on how their respective trained models are integrated for real-world mobilization. The furthest extent deployment being referred is the passing mention of utilizing trained models over the network, leveraging cloud and remote computing. This study aims to bridge this information gap of how trained models for object detection of crops and plants are reliably deployed in mobile application, format, where the distinguishing features of this paper are as per the following:

(a) End-to-end deployment focus. This paper emphasizes the end-to-end deployment of the trained model as an Android mobile application, where the model is integrated as the core component of the system.
(b) Deployment-specific considerations across model formats. Different model formats require distinct deployment approaches, each introducing specific constraints and design considerations that must be addressed to ensure correct and efficient integration.

2. PRIOR STUDIES

Recent deep learning research has increasingly focused on lightweight, edge-deployable models for plant disease detection and agricultural automation, targeting real-time inference on smartphones, embedded systems, and drones. Many approaches either adopt compact CNN architectures or optimize YOLO-based detectors to balance accuracy, speed, and resource usage in field conditions [5].

For maize produce, Khan *et al.* developed a mobile-based system that compared YOLOv3-tiny, YOLOv4, YOLOv5s, YOLOv7s, and YOLOv8n for three leaf diseases, reporting a mAP of 99.04% for YOLOv8n and successfully deploying the best model as a TensorFlow Lite Android app for on-device real-time detection and segmentation [6]. A more recent work on maize pests and diseases proposed YOLOv8-DBW, made with model architectural changes, thus improving mAP and reducing model parameters and FLOPs compared to YOLOv8n, enhancing suitability for mobile and embedded deployment [7]. Similarly, a lightweight YOLOv8 variants has been introduced for grape leaf disease detection (YOLOv8-ACCW), achieving notable gains in mAP of 3.1–5.6% over YOLOv8n while reducing model parameter count and size by 6.6% and 8.5%, respectively through real-time use on resource-constrained devices [8].

Beyond YOLO, several studies emphasize compact architectures for plant disease classification. VGG-ICNN employs about 6 million parameters and attains 99.16% accuracy on PlantVillage, outperforming many heavier CNNs while remaining suitable for deployment on constrained hardware [9]. Recent multimodal models like AgriFusionNet also highlight the importance of edge-oriented design, integrating efficient backbones and architecture refinement to maintain high accuracy with low latency and memory footprints [10].

From a systems perspective, on-device and edge AI surveys and reviews assert the significance of model compression (quantization, pruning, knowledge distillation) and algorithm–hardware co-design in making deep networks executable under mobile constraints [11], [12], [13]. Wang *et al.* synthesized that challenges such as limited memory, energy budgets, and real-time requirements, and identify model compression and hardware acceleration as key enablers of on-device AI across domains including agriculture [11]. Shuvo *et al.* reviewed efficient edge inference techniques, lightweight model architectures, optimization of existing models, and specialized accelerators as four main research directions for edge deployment [12].

Deeper into the software perspective, mobile platform-level comparisons show that the choice of development framework and language affects performance and resource usage. Wasilewski and Zabierowski compared Java, Flutter, and Kotlin/Native for sensor-driven Android apps and found that while Flutter can be competitive in CPU usage for some tasks, it tends to incur higher memory usage and should be used cautiously for highly resource-sensitive applications [14]. These findings are consistent with broader Android optimization work showing that leveraging hardware accelerators (GPU, NPU) and appropriate quantization schemes is crucial for achieving favorable tradeoffs between accuracy and inference speed when deploying deep learning models on mobile devices [11].

Despite the breadth of evaluation across these papers, none of them empirically explore the full life cycle of AI deployment within these frameworks. Specifically, there is a lack of study comparing model inference time, plugin reliability, and hardware utilization when AI models are run in mobile environments. Moreover, key concerns like on-device model updates, secure model input/output handling, and continuous learning mechanisms are entirely absent from these analyses.

This oversight reveals a critical research gap: although Flutter and Android Studio have been extensively evaluated from a software engineering perspective, their capacities for deploying and sustaining AI-powered mobile applications remain underexplored. As mobile devices increasingly become the de-facto platform for edge AI, it is essential to understand not only how these frameworks support AI development but how they perform under the resource and latency constraints that AI tasks often demand.

Despite the promising performance of lightweight YOLO-based models and advances in edge AI optimization, existing

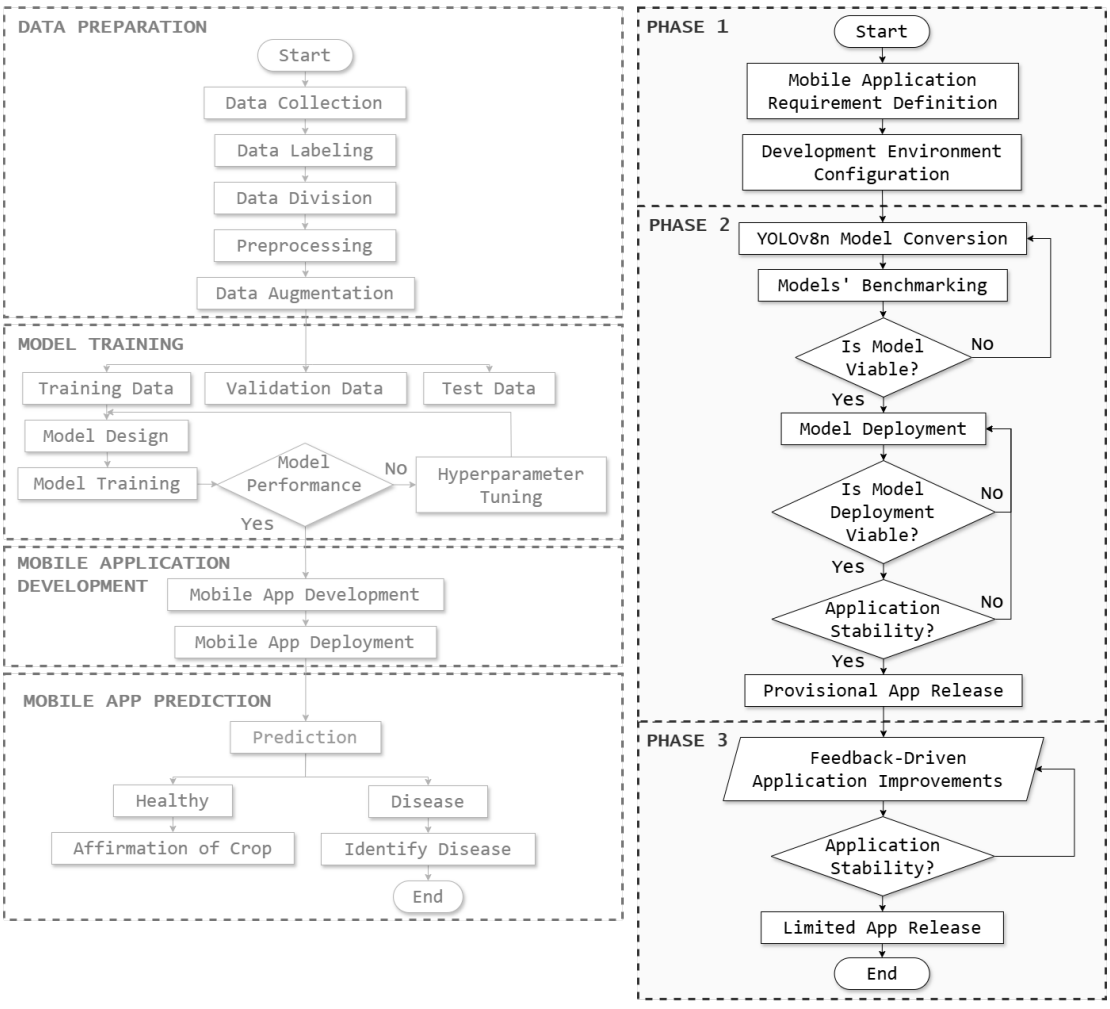
studies largely focus on model accuracy and architectural improvements. Comparatively little attention has been given to the practical challenges of end-to-end mobile deployment, including model conversion, runtime performance across deployment formats, framework integration, and real-world inference behavior. Consequently, the deployment characteristics of YOLO-based plant disease detection systems remain insufficiently explored, motivating the present study.

3. METHODOLOGY

The final model production parameters are detailed in Table 1. Figure 2(a) presents a flowchart showing a general overview of how the plant disease detection model is projected for production and deployment. Figure 2(b) illustrates a structured, phased development pipeline for building a mobile application that integrates a YOLOv8n object detection model. The workflow is divided into three major phases, each representing key stages of the application's evolution, from planning and setup, then AI model integration, and finally, testing and feedback-driven refinement for deployment.

Table 1. Model production parameters.

Aspect	Description
Dataset Specification	A total of 11,860 images after augmentation, of a standardized 640-by-640 pixels image size achieved through contrast stretching
Dataset Split	Train/validation/test split: 80:15:5, equally applied to all 23 classes.
Augmentation Techniques	a) Rotation: Between -3° and +3° b) Shear: ±1° Horizontal, ±1° Vertical c) Saturation: Between -10% and +10% d) Exposure: Between -6% and +6% e) Blur: Up to 1px f) Noise: Up to 0% of pixels g) Bounding Box: Horizontal Flip, Vertical Flip
Training Settings	a) Architecture: YOLOv8-nano b) Epochs: 150 c) Batch Size: 32 a) Optimizer: AdamW b) Learning Rate: 0.001 c) Weight Decay: 0.0001 a) Loss Function: Cross Entropy Loss
Hardware	Google Colab Pro: Python 3 Google Compute Engine Backend (GPU)



(a) (b)

Figure 2. The flowchart: (a) Model production and deployment; (b) Mobile app development.

The initial stage involves clearly defining the application's purpose, target users, functional requirements, and system specifications. This includes identifying and deliberating the need for real-time object detection, expected device compatibility, performance benchmarks, and user interface expectations. Proper outlining these elements as in Table 2 ensures the rest of the development process is goal-oriented and well-scoped.

Table 2. Mobile application requirement definition.

Aspect	Details
App Name	ReLeaf.
App Purpose	Detect disease in Solanaceous crops using YOLOv8n.
Target Users	Farmers, Agriculturists, Hobbyists.
Target Platform	Android as most mobiles are Android phones.
Functional Requirements	Object detection, detect diseased portion of Solanaceae crops. Then guides on method of preventive and curative treatments for the crops. Fast inference time, User-friendly interface.
AI Model	YOLOv8n in any compatible format, mAP 50 more than 80%.
Data	Images (.jpg, .jpeg, .png, etc.)
Security	Minimal concern as no sensitive information is being handled.
Connectivity	Offline if possible.

Next, the development environment is prepared by installing and configuring development tools such as Flutter or Android Studio, setting up dependencies, and ensuring compatibility with AI model formats. Configuration also includes integrating essential packages for camera access, file handling, and TFLite or ONNX runtime support to allow on-device model inference. The following step involves converting the YOLOv8n model, originally trained in PyTorch, into a mobile-friendly format such as TensorFlow Lite or ONNX, with quantization or pruning to reduce model size and inference latency. The goal is to prepare the model for optimal deployment on edge devices without significantly accuracy loss. The quantization method utilized was of the post-training TensorFlow Lite float16 weight quantization. This is where after model training, the model representation's weights were converted from 32-bit to 16-bit floating point precision. Strictly, this is only partial quantization as no integer arithmetic was used during conversion.

The conversion is done in a Python virtual environment primed to ensure dependencies do not conflict with one another. Python libraries' version compatibility must be strictly adhered, as some libraries lack back or forward compatibility with each other to otherwise work. The specific version of each library compatible with the development host machine and development environment is noted within a 'requirements.txt', which is referenced by the running code when installing the dependencies. Then the conversion code was run in a complete virtual environment as best practice to prevent library version conflicts.

Once converted, the produced model is benchmarked. Performance metrics such as inference time, memory usage, and accuracy parameters (e.g., mAP, precision, recall) are measured and collected for analysis. This step validates whether the converted model meets the predefined performance thresholds during inferencing. Each iteration of the conversion is benchmarked, as adjustments in the conversion code results in differently performing converted models.

At the model deployment decision point, the viability of the model is assessed based on the benchmarked results. If the model does not meet the performance criteria, the process is iterated back to the conversion or training phase to refine the model. Conversely, if the model is deemed viable, the process proceeds to integration of the validated model into the mobile application. This involves linking the model inference logic with application components and access permissions such as the camera feed and user interface output. The app is programmed to pass image frames to the model and then display the detection results inferenced by the model to the user.

The model deployment viability decision point checks whether the model operates as expected within the mobile app. If integration issues arise such as incorrect outputs, crashes, or bottlenecks, then further debugging is needed. If the deployment is successful, the development proceeds to full application testing. The app is tested for overall system stability under different usage conditions. This includes stress testing, error handling, frame rate maintenance, and responsiveness. If the app shows signs of instability or poor performance, further optimization is required before proceeding.

A provisional version of the app is then released to a small group of internal testers or beta users. The goal is to collect real-world feedback and monitor app performance in diverse hardware and usage conditions. This controlled release acts as a soft launch. Based on tester feedback, the application is iteratively improved. Adjustments may include refining model confidence thresholds, optimizing the user interface, fixing bugs, or improving battery efficiency. This ensures the app aligns with user expectations and operates smoothly in field conditions.

The revised application undergoes another round of stability verification. If issues persist, further refinements are made. Once confirmed stable, the app is deemed ready for limited-scale deployment. The app is released to a broader but still controlled user base via invitation-only users. This phase allows for monitoring app performance under higher load and real-world variability before a full public launch.

The dataset used in this study mainly came from the PlantVillage image corpus, a public repository hosted on Kaggle [15]. The PlantVillage project offers an open-access database with over 54,000 leaf images across 38 different plant-disease classes from 14 crop species, supporting research in plant health diagnostics through computer vision. For this study, we created a curated subset consisting of 23 classes, with about 300 images each. We gathered these images from various PlantVillage Kaggle uploads to maintain class balance and consistent annotation.

To increase class diversity and representation, especially for the Chilli (Capsicum annuum) and Eggplant (Solanum melongena) categories, we added more images from publicly available agricultural web sources. We obtained these extra samples from open-access research archives and agricultural extension sites, ensuring they were released under open licenses.

All extra images were processed to maintain uniform resolution, class consistency, and compatibility with the main dataset. Including these web-sourced images boosted intra-class variability and improved the model's ability to generalize across different lighting and environmental conditions.

License and Provenance: All datasets used in this study were sourced from open-access platforms intended for educational and research purposes. The PlantVillage corpus and its Kaggle derivatives follow the Creative Commons Attribution 4.0 International (CC BY 4.0) license, allowing redistribution and adaptation with appropriate credit to the original authors and hosting sites. The additional images from agricultural research portals and educational web archives were confirmed to be under similar CC BY 4.0 or public-domain licenses. We recorded metadata, dataset documentation, and source URLs to ensure clear data provenance and ethical reuse in line with open data sharing standards.

Appendix 1 presents the curated dataset used for training and validating the ReLeaf model, comprising of 23 image classes representing various solanaceous crop conditions across four primary species of chilli, eggplant, potato, and tomato drawn from curated subsets of the PlantVillage corpus distributed through Kaggle. Each class consists of 300 images, thus contributing to a balanced dataset. The table lists both healthy and diseased plant parts, including leaves and fruits, such as *Chilli Anthracnose Fruit*, *Eggplant Cercospora Leaf Spot*, *Potato Phytophthora Infestans Leaf*, etc. This uniform structure ensures consistent class representation and mitigates data imbalance issues that could affect model performance. By incorporating both foliar and fruit-level symptoms across multiple crops, the dataset enables the YOLOv8n model to generalize disease recognition under varied visual and physical traits, supporting reliable detection in real-world agricultural applications.

4. RESULTS AND DISCUSSION

The preliminary stage of mobile development begins with minimizing points of failure to ensure a streamlined development process, reducing need to backtrack progress from foundational failures. Due to such considerations, the pretrained YOLOv8n model must be rigorously tested to ensure faults do not stem from it during development.

The conversion of AI models between file formats is essential to facilitate efficient deployment across a range of distinct computing platforms. Runtime environments, hardware capabilities, and optimization toolchains affect which models work best. These conversions enable the integration of platform-specific enhancements, including quantization, pruning, and hardware acceleration.

Table 3 summarizes recommended model file formats for different deployment targets, aligning each platform with its native or most compatible inference framework. For Android, TensorFlow Lite (TFLite) has become a de facto standard for on-device deployment because it bundles a compact flatbuffer format with a specialized interpreter and operator kernels optimized for ARM processors and mobile DSPs [16]. Through Android's Neural Networks API (NNAPI), TFLite can transparently offload supported operators to hardware accelerators such as GPUs, NPUs, and DSPs, substantially improving latency and energy efficiency on devices that expose these backends [16]. Quantized models further reduce memory footprint and power consumption, showing that 8-bit quantization typically preserves near-floating-point accuracy [17].

Table 3. Model file format compatibility recommendations.

Platform	Standard	File Format	Justification
Android Native	TensorFlow Lite	.tflite	Utilizes NNAPI.
iOS Native	CoreML	.mlmodel	Apple Neural Engine optimized.
Cross-Platform	ONNX Runtime	.onnx	Widely compatible.
Desktop/Server (Windows/Linux)	ONNX Runtime, PyTorch, TensorFlow Lite	.onnx, .pt, .tf, .tflite	Compatible full-device utilization. Optimized for throughput and scaling.

On iOS, Apple's Core ML model file format is tightly integrated with the Apple Neural Engine and Metal Performance Shaders translation layer. Empirical evaluations on iPhone and iPad devices show that Core ML effectively exploits the Neural Engine to reduce inference time and battery usage for vision models such as MobileNetV2 and ResNet50, often outperforming TensorFlow and PyTorch in energy efficiency on newer SoCs. However, conversion pipelines from TensorFlow or TFLite into Core ML can be brittle: in some cases, converted models exhibit severe accuracy degradation or even zero accuracy, underlining the need for careful validation after export. [18]

Given that PyTorch dominates contemporary research workflows, particularly in computer vision, most new architectures are first implemented and trained in PyTorch before being exported to deployment formats. Surveys on lightweight and edge-oriented deep learning explicitly note PyTorch's flexibility and popularity in prototyping, while also highlighting that its deployment APIs are less mature than those of TFLite or Core ML, motivating strong ecosystem support for conversion to ONNX and TFLite. [16], [18], [19]

For deployment environments, the Open Neural Network Exchange (ONNX) model file format offers a uniform representation for different frameworks that can be exported from PyTorch or TensorFlow, then executed efficiently via ONNX Runtime. ONNX Runtime supports multiple execution providers (e.g., CUDA, TensorRT, DirectML, CPU-optimized backends), enabling the same model to target diverse hardware with minimal code changes. Empirical work shows that converting models such as ResNet, MobileNet, and BERT to ONNX often yields lower inference time and, in many cases, improved energy efficiency relative to native framework runtimes, especially when coupled with TensorRT execution providers on GPU. [16], [17], [19]

Model conversion is done for mobile applications development compatibility. The resultant model may have differed performance before and after conversion, with varied compatibility with different platforms. The target device platform -

architecture may differ, which will exhibit varied performance efficiency from the device platform's compatibility with the file format of the AI model deployed. For Android application development, the model file formats used are PyTorch, ONNX, and TensorFlow Lite, and their average performance as in Table 4. The specific losses in performance were calculated with Equation (1) and Equation (2), whereas The F1-Score was calculated using Equation (3).

Table 4. Validation performance of each model file format.

Average Performance	PyTorch	ONNX	TFLite (FP32)	TFLite (FP16)
mAP @ 0.50 (%)	98.7	93.5	39.9	39.9
mAP @ 0.50:0.95 (%)	86.7	81.1	38.2	38.2
Precision (%)	98.7	93.6	64.2	64.2
Recall (%)	97.2	92.9	50.3	50.3
F1-Score (%)	97.9	93.3	49.3	49.3
Inference Speed (ms)	22.8	24.9	19.1	19.1
File Size (MB)	6.0	11.6	11.6	5.9
Typical mAP/F1-Score Loss (%)	–	~1-3	~2-5	~2-5%
Actual mAP@50 Loss (%)	–	5.3	59.6	49.6
Actual F1-Score Loss (%)	–	4.7	59.6	49.6

$$F1\_Score\ Conversion\ Loss = \frac{F1\_Score\ PyTorch\ Model - F1\_Score\ Converted\ Model}{F1 - Score\ PyTorch\ Model} \quad (1)$$

$$m@AP50\ Conversion\ Loss = \frac{m@AP50\ PyTorch\ Model - m@AP50\ Converted\ Model}{m@AP50\ PyTorch\ Model} \quad (2)$$

$$F1\_Score\ = \frac{Precision \times Recall}{Precision + Recall} \quad (3)$$

Based on Table 4, on average, PyTorch performs the best with ONNX following close behind. Only the TensorFlow Lite model files display a glaring difference in overall performance. Notably, both Float 32 and Float 16 TensorFlow Lite models display similar performance. While the Float 16 model is quantified to a degree, the model's computation is still done in Float32, i.e. they are dequantized to Float 32 runtime during inference. The losses after conversion from PyTorch format are clarified for ONNX, TFLite (FP32) and TFLite (FP16), where TFLite at both 32 and 16 floating point precision display no difference, in contract with how ONNX is performing with minimal loss of performance. To further clarify this aggregate data, Figures 3(a) and 3(b) tabulated based data with the class index of Table 5. These figures escalate the visualization of the performance gaps between the models, where the TensorFlow Lite models have a low F1-Score, and the model is unreliable from having imbalances between the classes. When the gap is so large, the model would be detrimental to the prospective application development. Overall, the ONNX model format shows a small decrease in performance compared to the PyTorch model, while the TFLite model displays a substantial decrease. The said value effects are affected across the board to all the classes within the model. This shows that the conversion is suboptimal.

Table 5. Class Index in the dataset.

[0] Chilli__Anthracnose_Fruit	[8] Eggplant__Healthy_Fruit	[16] Tomato__Bacterial_Spot_Leaf
[1] Chilli__Bacterial_Leaf_Spot	[9] Eggplant__Healthy_Leaf	[17] Tomato__Early_Blight_Leaf
[2] Chilli__Healthy_Leaf	[10] Potato__Alternaria_Solani_Leaf	[18] Tomato__Healthy_Fruit
[3] Chilli__Healthy_Fruit	[11] Potato__Common_Scab_Fruit	[19] Tomato__Healthy_Leaf
[4] Chilli__Mosaic_Virus_Leaf	[12] Potato__Healthy_Fruit	[20] Tomato__Late_Blight_Leaf
[5] Eggplant__Cercospora_Leaf_Spot	[13] Potato__Healthy_Leaf	[21] Tomato__Leaf_Mold
[6] Eggplant__Colorado_Potato_Beetle	[14] Potato__Phytopthora_Infestans_Leaf	[22] Tomato__Yellow_Leaf_Curl_Virus
[7] Eggplant__Fruit_Rot	[15] Tomato__Anthracnose_Fruit	

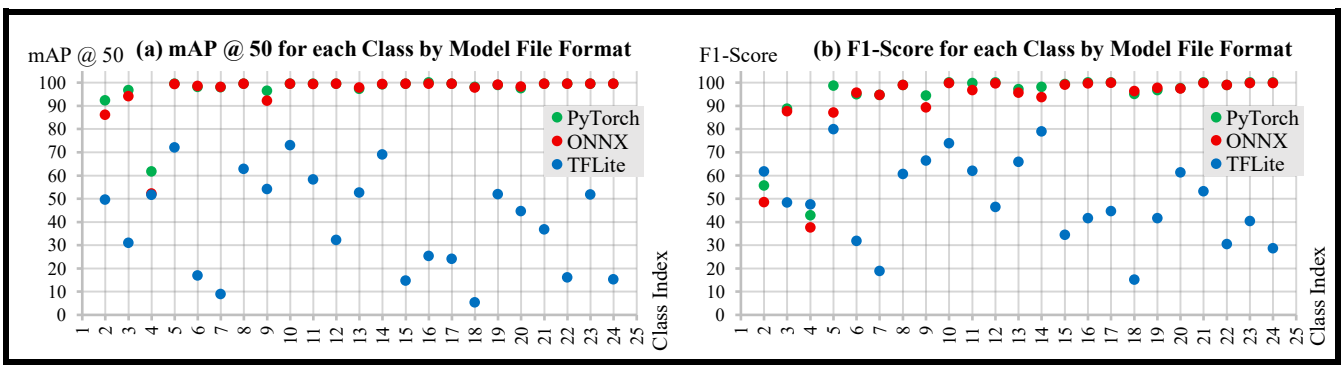


Figure 3. Plot for each class by model file format: (a) mAP@50, (b) F1-Score.

Table 6 shows the model performance by class, and it is noted that some classes had much lower recall values than others, especially in the TensorFlow Lite implementation. For example, Chilli Mosaic Virus Leaf (20.2%) and Potato Phytophthora Infestans (27.7%) had the lowest recall rates. This means the model missed a significant number of true positive samples in these categories. The drop in recall likely comes from several factors. These include the visual similarity between disease symptoms, the lack of distinct lesion patterns, and the effects of model quantization that usually happen when converting to TensorFlow Lite for edge use.

Table 6. Model performance by class.

Model Properties	mAP @ 0.50 (%)			Precision (%)			Recall (%)		
Format / Class Index	.pt	onnx	.tflite	.pt	onnx	.tflite	.pt	onnx	.tflite
[1]	92.3	86.1	31.0	99.4	100.0	73.8	80.3	78.1	35.9
[2]	96.7	94.1	49.6	61.4	64.4	65.8	50.9	38.9	58.0
[3]	61.7	52.2	51.6	48.9	37.6	40.1	38.2	37.6	58.4
[4]	99.5	99.4	72.1	100.0	87.1	84.0	97.4	87.1	76.2
[5]	98.1	98.5	17.0	93.5	94.6	74.2	96.6	96.6	20.2
[6]	98.0	98.1	9.0	92.8	94.4	75.0	96.7	95.1	10.8
[7]	99.5	99.5	62.8	98.0	98.0	48.6	100.0	100.0	80.6
[8]	96.4	92.2	54.1	97.0	89.3	71.9	92.0	89.3	61.7
[9]	99.5	99.5	73.1	100.0	99.7	68.9	100.0	100.0	79.6
[10]	99.5	99.5	32.2	100.0	99.4	46.0	100.0	100.0	47.1
[11]	97.3	97.9	52.6	99.9	99.6	64.9	94.7	92.0	67.0
[12]	99.2	99.3	69.1	100.0	93.7	87.4	96.2	93.7	71.9
[13]	99.5	99.5	14.8	98.8	99.1	30.8	100.0	99.1	39.1
[14]	100.0	99.5	25.4	100.0	99.4	83.7	100.0	100.0	27.7
[15]	99.5	99.5	24.1	100.0	99.9	44.9	100.0	100.0	44.3
[16]	98.1	97.8	5.3	93.5	96.8	88.9	96.8	95.9	8.3
[17]	98.9	99.1	52.0	96.9	97.2	28.8	96.7	98.3	74.6
[18]	97.5	98.2	44.6	100.0	99.6	84.0	95.1	95.2	48.2
[19]	99.5	99.5	36.8	100.0	99.6	73.0	100.0	100.0	41.8
[20]	99.5	99.5	16.1	97.9	98.1	81.0	100.0	100.0	18.7
[21]	99.5	99.5	51.8	100.0	99.7	26.5	100.0	100.0	84.0
[22]	99.5	99.5	15.3	100.0	100.0	85.7	100.0	99.6	17.2

.pt – PyTorth; .onnx – ONNX; .tflite – TensorFlow Lite

The lower recall indicates a higher rate of false negatives, meaning some diseased leaves might go undetected during inference. In real-world farming situations, like mobile plant disease detection, this could lead to underreporting of infections and delays in treatment. Although high precision values in most classes show the model is confident in correct detections, the lower recall points to a trade-off between optimizing model size and sensitivity in detection. Thus, more fine-tuning, data augmentation, or using mixed-precision quantization could help close this performance gap and improve detection reliability in lightweight model use.

Converting models between frameworks and formats introduces noticeable risks. Conversion pipelines must map operators, tensor layouts, and numerical precisions across toolchains that make different design choices. Mismatches in operations (e.g., padding semantics, activation implementations, batch normalization folding) and differences in floating-point handling can accumulate rounding errors and alter the internal feature representations, potentially degrading predictive performance. Quantization adds further perturbations: while INT8 post-training quantization is usually safe, more aggressive schemes or poorly calibrated pipelines can introduce bias and accuracy loss. [12], [16], [17]

Quantization can further contribute to a model's decline in performance due to reduced numerical precision. Model conversion from float32 to float16 reduces the number of mantissa bits, resulting in a coarser numerical representation of weights and activations, thus logically the smaller model will be less effective. This reduced precision introduces rounding errors and error accumulation during inference, affecting prediction stability. Explicitly, FP16 does not induce quantization bias that is associated with integer quantization, this loss of numerical precision follows through to measurable accuracy degradation, especially in lightweight architectures that are more sensitive to small numerical perturbations such as YOLOv8-nano used in this study. These effects, when unaccounted for, can cumulatively cause substantial loss in accuracy, particularly in lightweight model architecture.

PyTorch model files are often saved as a compact and binary format, with efficient internal serialization as opposed to ONNX model files which are uncompressed, hence why a converted ONNX model will undoubtedly be larger. The compression ratio, percentage of reduction and percentage of latency speedup are calculated using Equations (4), (5) and (6) respectively. Since the ONNX Runtime acts as the bridge between model types due to its inherent interoperability and uncompressed state, it will serve as the baseline for the calculations with Equations (4) and (5) for the various models. However, the only compression that occurs within the confines of this study is from ONNX to TFLite float16 (FP16). The conversion results are displayed in Table 7.

$$Compression\ Ratio\ = \frac{Baseline\ Model\ Footprint, MB}{Converted\ Model\ Footprint, MB} \tag{4}$$

$$Reduction\ (\%) = 100 \times \left(1 - \frac{1}{Compression\ Ratio}\right) \tag{5}$$

$$Latency\ Speedup\ (\%) =\ 100 \times \left(1 - \frac{Baseline\ Latency}{Converted\ Latency}\right) \tag{6}$$

Table 7. Model properties after conversion.

File Format	File Size (MB)	Compression Ratio	Size Reduction (%)	mAP @ 50 (%)	Latency Speedup (%)
PyTorch	6.0	N/A	N/A	98.7	9.2
ONNX	11.6	-	-	93.5	-
TFLite (FP32)	11.6	0.0	0.0	39.9	30.4
TFLite (FP16)	5.9	1.97	49.1	39.9	30.4

While the TFLite models have a substantial decrease in latency as shown in Table 7, the models' performance degrades beyond acceptable use. On the other hand, PyTorch retains a confident lead in both model performance and latency compared to the ONNX model format. Format conversion clearly degenerates a model's prowess.

Openja *et al.* [16] performed a large empirical study of Keras and PyTorch models converted to ONNX and Core ML. Overall, they found that prediction accuracy typically remained comparable to the originals and that converted models often enjoyed smaller sizes and faster inference, particularly when run with ONNX Runtime. However, the study also reported cases where converted models behaved differently in terms of accuracy, robustness to adversarial attacks, and runtime performance, especially for Core ML targets, stressing that developers must not assume meaning after conversion. Follow-up work on runtime infrastructures similarly shows that ONNX Runtime frequently improves inference time relative to native PyTorch and TensorFlow for vision and language models, though the exact gains depend on model type, batch size, and backend configuration [17], [18].

Beyond pure accuracy, runtime environments themselves introduce variability. ONNX Runtime, TFLite, and Core ML use different kernel libraries, graph optimizations, and execution strategies that prioritize throughput or latency to varying degrees. Overall, model conversion and deployment form a cascade of transformations spanning operator mapping, quantization, graph rewrites, and backend-specific optimizations. Recent surveys and empirical studies converge on the recommendation that accuracy-critical applications should employ robust toolchain validation, quantization-aware training where possible, and systematic cross-runtime benchmarking of accuracy, latency, memory, and energy usage to preserve performance during deployment.

Finally, at the deployment stage, the PyTorch model was chosen. This model's deployment accuracy was tested with images sourced online. Figure 4 shows the field-tested accuracy result. Overall, the classes where leaves were the prominent feature struggled at being identified accurately. Often time the leaves were returned as other classes, thus being unreliable in detecting disease with plant leaves. This is apparent with how the class index 14 (potato phytopthora_infestans_leaf) class had no accurate detections, where the overall model accuracy dropped to 70.9%. This drop is further corroborated with the confusion matrix Figure 5, showing image samples misidentified as multiple other classes that had leaves/leaf as their main feature. For clarity, the classes with their accuracy are plotted in Figure 4 collated from Figure 5, those where a leaf or leaves are a main feature has their accuracy value highlighted in green.

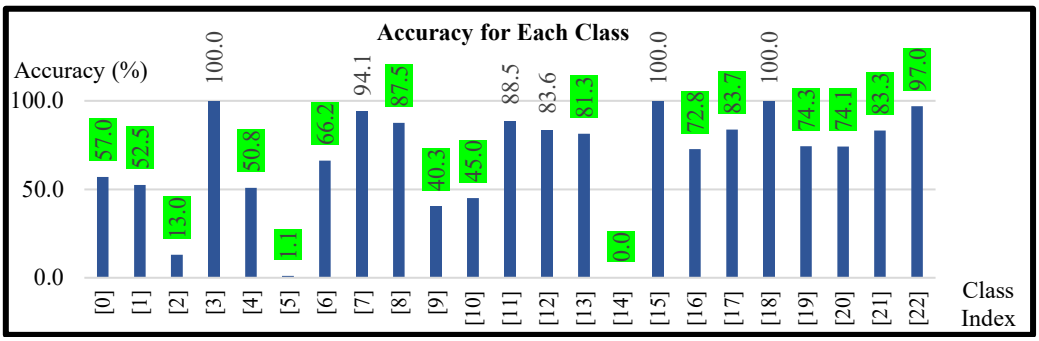


Figure 4. Field-tested accuracy through the deployment mode.

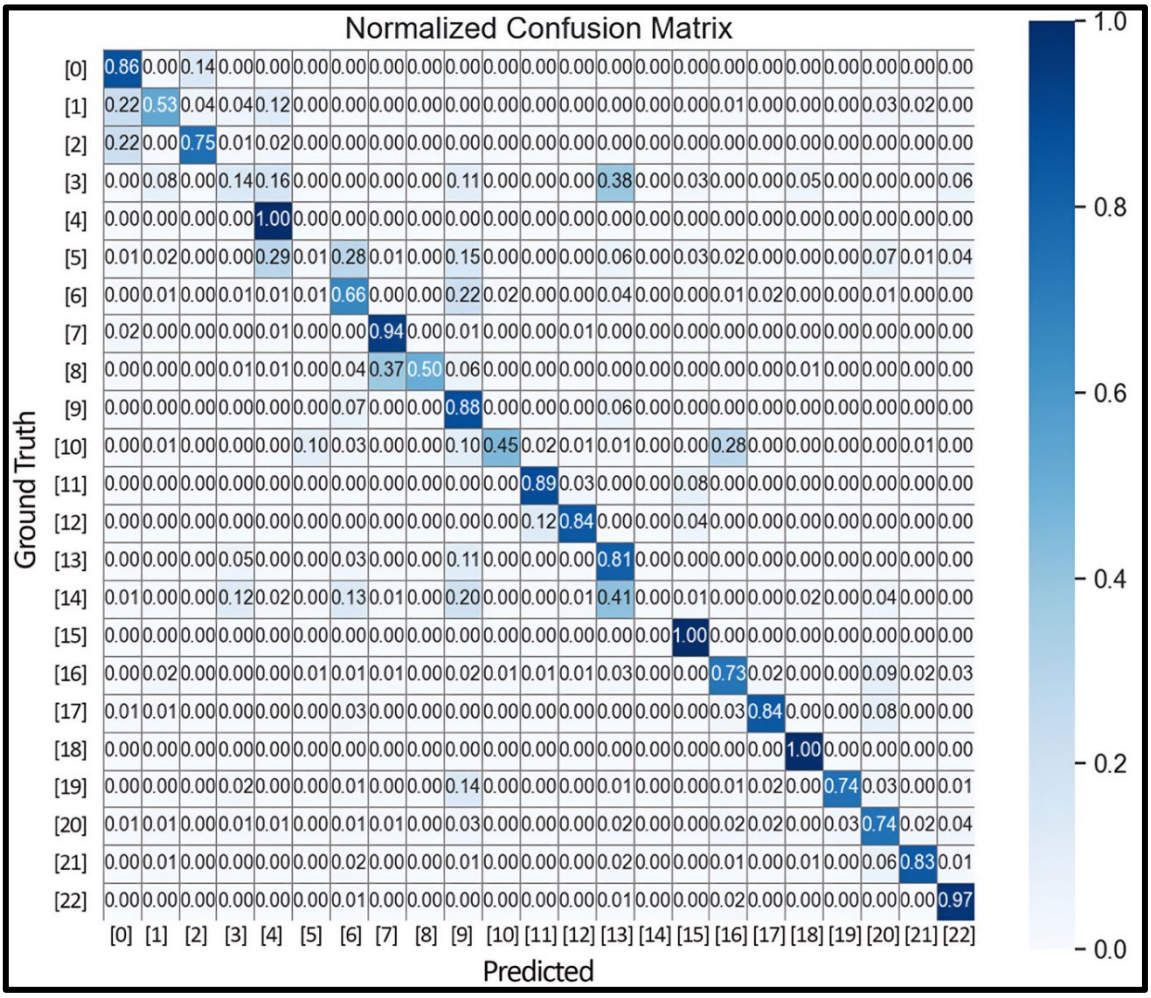


Figure 5. Normalized confusion matrix of field-tested accuracy.

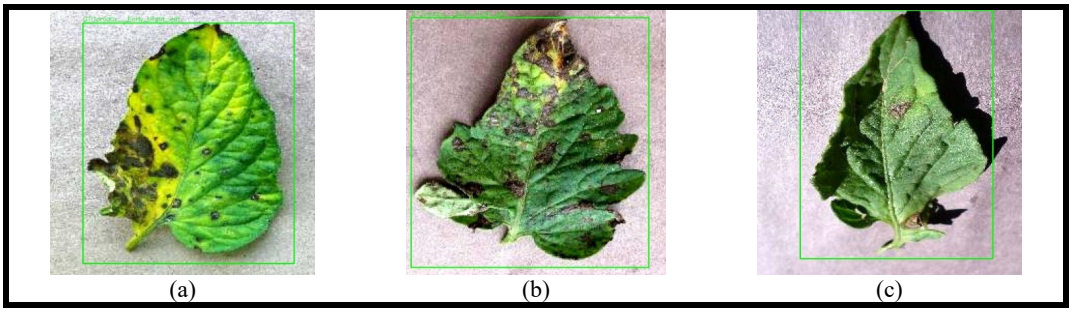


Figure 6. (a) Tomato early blight; (b) Tomato bacterial leaf spot; (c) Tomato late blight.

The accuracy of leaf classes is lower that the fruit classes as illustrated in Figure 6. The images within the figure exhibit visually distinct traits such as differences in leaf shape, coloration, venation and texture aside from the symptomatic lesions. Despite the distinction in said traits the model frequently labels them as "Tomato_Early_blight_leaf". This behavior stems from several well-documented issues in plant disease vision systems.Plant disease recognition research frequently relies on datasets curated under controlled laboratory conditions, with homogeneous backgrounds, stable illumination, and carefully framed leaves. Systematic reviews note that many benchmark datasets used for training deep models in agriculture lack the diversity of real farm environments, where background clutter, variable lighting, occlusion, and sensor noise are prevalent. Consequently, models that achieve near-perfect accuracy on lab collections such as PlantVillage often exhibit substantial drops when evaluated on mixed lab and in-field datasets like PlantDoc, FieldPlant, or wild imagery captured in operational settings [20], [21].

Recent work explicitly quantifies this domain shift by training models on lab datasets and testing them on field datasets across different crops. Ensemble architectures that perform extremely well on PlantVillage (≈99–100% accuracy) typically fall to ≈60% accuracy on more realistic datasets such as PlantDoc, despite using aggressive data augmentation. Domain

adaptation studies formalize this problem as a shift between source (lab) and target (field) domains. For instance, Wu *et al.* propose MSUN, an unsupervised domain adaptation framework that leverages multi-representation learning, subdomain adaptation, and uncertainty regularization to bridge this gap, achieving significant improvements on PlantDoc, Plant-Pathology, and other in-field benchmarks compared with conventional transfer learning baselines [22].

A key driver of this generalization gap is the visual ambiguity among leaf diseases. Many pathologies share overlapping symptoms such as necrotic or chlorotic spots, halos, edge discoloration, and vein pattern changes, resulting in high inter-class similarity and large intra-class variation [20], [21]. Feature spaces learned from standardized lab images can become entangled, causing the model to confuse classes with similar symptom patterns when exposed to more diverse field imagery. This effect is magnified when leaf-based classes constitute a large proportion of the training set and dominate optimization, leading to shared internal representations that are highly sensitive to subtle context cues and background artifacts.

Another limiting factor is insufficient within-class diversity in existing training dataset. Many datasets lack variation in viewing angles, developmental stages of symptoms, lighting conditions, and environmental context. The consequence is overfitting to dataset-specific biases such as uniform backgrounds, canonical leaf poses, or sensor-dependent color calibration. Recent studies advocate combining lab and field imagery and using advanced augmentation or domain-adaptation strategies to improve robustness. Style-based augmentation, for example, performs style-consistent image translation to transfer background, lighting, and textural properties from one class or domain to another while preserving object structure, thereby synthesizing more diverse training examples and boosting generalization for classification, detection, and segmentation tasks. [20], [21]; Unsupervised domain adaptation approaches, including those that align subdomains and incorporate uncertainty regularization, have also shown consistent gains on cross-dataset plant disease classification benchmarks [22].

Collectively, current literature emphasizes that high performance on controlled benchmarks does not guarantee reliability in real agricultural settings. Achieving robust field deployment requires deliberate efforts to mitigate domain shift through diversified data collection, style and domain-aware augmentation, and explicit domain adaptation, alongside careful consideration of model compression and deployment formats so that edge-optimized models preserve accuracy when confronted with real-world variability.

Collectively, the deployment factors model conversion, quantization, and inference on mobile devices may introduce numerical or precision changes. These can disproportionately affect predictions for boundary cases where various classes are visually similar. Addressing these issues requires: collecting diverse field-image data, applying augmentation or domain-adaptation, using confusion-aware loss or label-smoothing, and verifying conversion/inference fidelity in deployment environments.

The field-acting inference was performed by using TensorFlow Lite on a CPU-based Flask server, where the model was hosted through a Flask server running on a generic instance of Windows 11 and Ubuntu. The default XNNPACK delegate was utilized for optimizing floating-point executions. Inferences performed well through these default settings, hence not needing GPU delegate, CUDA or any other appendance. No appreciable differences were observed through the multiple devices the inferencing was performed on.

The application then evaluated its scalability. Figure 7 illustrates the simulated average inference time per image input as the number of simultaneous user requests increases. The inference time curve exponentially, especially after 100 requests. At this stage of development, the infrastructure in place is suitable to accommodate approximately 100 users before the user experience degrades beyond feasible use.

The developed mobile application is organized around a structured and intuitive architecture that integrates multiple interactive pages, a selection of which is illustrated in Figure 8. This architecture ensures coherent navigation, visual uniformity, and ease of use across all functional components. Broadly, the system is divided into two core sections Detection and Library: each designed to fulfill a specific role within the plant disease detection framework.

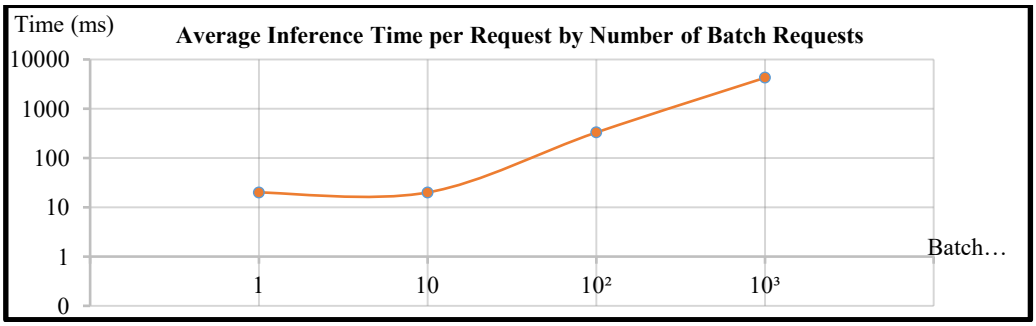


Figure 7. Average inference time for each request when number of simultaneous requests increases.

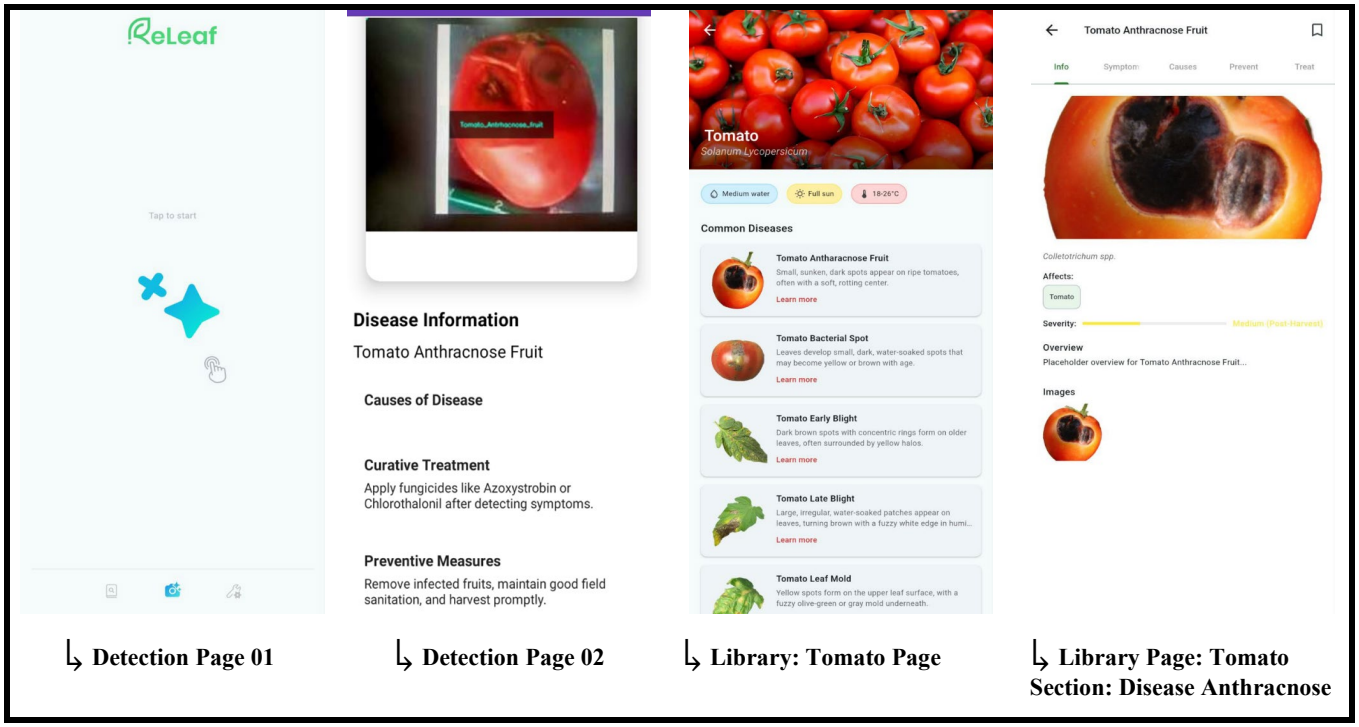


Figure 8. Sample pages from mobile application.

Upon initialization, the user is automatically directed to Detection Page 01, which functions as the primary operational interface of the system. This page provides a straightforward yet effective means of initiating the detection process. With a single tap, the user may either select an existing image from the device's gallery or capture a new photo using the built-in camera. Once the image is obtained, it is automatically preprocessed and transmitted to the embedded deep learning model for inference. The model subsequently analyzes the image in real time to identify any visual indicators of plant disease. The results are then presented on Detection Page 02, where the detected regions, class labels, and confidence scores are clearly displayed. This streamlined workflow enables users, including those with minimal technical background, to conduct rapid and reliable disease assessments with minimal effort or intervention.

The Library Section, on the other hand, serves as a comprehensive repository of information for users who wish to explore plant diseases manually. Beyond automated detection, this section provides detailed insights into each disease, including its characteristic symptoms, causal factors, preventive measures, and treatment methods. The library is arranged hierarchically-first categorized by plant species and subsequently by specific diseases within each plant type. This layered organization facilitates efficient information retrieval and supports deeper learning.

By integrating automated detection with an informative reference system, the application effectively bridges the gap between technology and agricultural knowledge. It not only enables timely and accurate diagnosis but also fosters user understanding of plant health management. This dual-purpose design as both a diagnostic and educational tool, aligns with broader goals of promoting sustainable agricultural practices through accessible technological innovation.

CONCLUSION

The development of *ReLeaf*, a mobile application for solanaceous crop disease detection using YOLOv8n, demonstrates the potential of integrating lightweight AI models into smartphones for agricultural diagnostics. Simulation results showed strong performance, achieving a 98.7% mAP on the test dataset. However, real-world validation highlighted a significant drop in accuracy to 70.2%, particularly in differentiating between leaf disease classes with similar visual features. This performance gap was mainly due to two factors: (i) loss of fidelity during model conversion and quantization, and (ii) the inherent challenge of distinguishing visually similar leaf diseases under variable field conditions such as lighting and texture. These findings highlight the complexity of translating controlled model performance into reliable field applications.

Despite these challenges, the deployment of the PyTorch model provided a workable solution with stable inference speeds and user-friendly functionality, offering a meaningful step toward real-time AI adoption in farming communities. Future improvements should focus on expanding the dataset with more diverse real-world images to improve robustness, and applying optimization methods such as quantization-aware training or pruning to achieve better efficiency on mobile devices. Broader field trials with farmers across different regions are also essential for validating usability and scalability. Additionally, integrating user feedback features and simple IoT sensor inputs (e.g., soil or weather conditions) could evolve *ReLeaf* into a more comprehensive smart farming tool. Ultimately, this study provides a foundational framework for scaling AI-powered plant disease detection across diverse crops and advancing digital agriculture for smallholder farmers.

ACKNOWLEDGEMENT AND FUNDING

This study is funded by Ministry of Higher Education (MOHE) through the Fundamental Research Grant Scheme (FRGS), No. FRGS/1/2022/ICT02/UTEM/02/6.

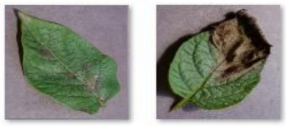
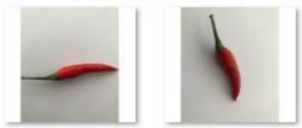
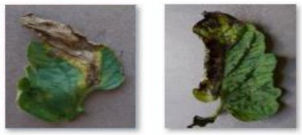
DECLARATION OF CONFLICTING INTERESTS

The authors declare no potential conflicts of interest with respect to the research and publication of this article.

REFERENCES

[1] D. Sharma, R. Negi, K. Sharma, A. Verma, S. Thukral, Dimple and S. Gautam, Diagnosis and management of important diseases of solanaceous vegetables, *The Pharma Innovation: International Journal*, 11(10S), 2022, 640–647.
[2] J. Liu and X. Wang, Plant diseases and pests detection based on deep learning: a review, *Plant Methods*, 17(1), 2021, 22.
[3] S. P. Mohanty, D. P. Hughes and M. Salathé, Using deep learning for image-based plant disease detection, *Frontiers in Plant Science*, 7, 2016, 1419.
[4] J. James, The smart feature phone revolution in developing countries: Bringing the internet to the bottom of the pyramid, *The Information Society*, 36(4), 2020, 226–235.
[5] A. Rathod, D. Panchal, N. Minocha, and D. D. Bhatt, Crop Disease detection using lightweight deep learning model for smartphone, *International Journal on Science and Technology*, 16(2), 2025, 1–10.
[6] F. Khan, N. Zafar, M. N. Tahir, M. Aqib, H. Waheed and Z. Haroon, A mobile-based system for maize plant leaf disease detection and classification using deep learning, *Frontiers in Plant Science*, 14, 2023, 1079366.
[7] X. Gan, S. Cao, J. Wang, Y. Wang and X. Hou, YOLOv8-DBW: An Improved YOLOv8-based algorithm for maize leaf diseases and pests detection, *Sensors*, 25(15), 2025, 4529.
[8] Z. Chen, J. Feng, K. Zhu, Z. Yang, Y. Wang and M. Ren, YOLOv8-ACCW: Lightweight grape leaf disease detection method based on improved YOLOv8, *IEEE Access*, 12, 2024, 123595–123608.
[9] P. S. Thakur, T. Sheorey and A. Ojha, VGG-ICNN: A lightweight CNN model for crop disease identification, *Multimedia Tools and Applications*, 82(1), 497–520.
[10] S. Albahli, AgriFusionNet: A lightweight deep learning model for multisource plant disease diagnosis, *Agriculture*, 15(14), 2025, 1523.
[11] X. Wang, Z. Tang, J. Guo, T. Meng, C. Wang, T. Wang and W. Jia, Empowering edge intelligence: A comprehensive survey on on-device AI models, *ACM Computing Surveys*, 57(9), 2025, 1–39.
[12] Md. M. H. Shuvo, S. K. Islam, J. Cheng and B. I. Morshed, Efficient acceleration of deep learning inference on resource-constrained edge devices: A review, *Proceedings of IEEE*, 111(1), 2023, 42–91.
[13] G. Kanwar, Edge AI revolution: Compressed neural networks enabling real-time mobile processing, *European Modern Studies Journal*, 9(5), 2025, 114–128.
[14] K. Wasilewski and W. Zabierowski, A comparison of Java, Flutter and Kotlin/Native technologies for sensor data-driven applications, *Sensors*, 21(10), 2021, 3324.
[15] PlantVillage Dataset. https://www.kaggle.com/datasets/abdallahalidev/plantvillage-dataset (accessed 30.06.2026).
[16] M. Openja, A. Nikanjam, A. H. Yahmed, F. Khomh and Z. M. J. Jiang, An empirical study of challenges in converting deep learning models, *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Limassol, Cyprus, 2022, 13–23.
[17] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen and T. Blankevoort, *A White Paper on Neural Network Quantization*, arXiv:2106.08295, 2021.
[18] V. M. F. Jacques, N. Alizadeh and F. Castor, A study on the battery usage of deep learning frameworks on iOS devices, *Proceedings of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems*, Lisbon Portugal, 2024, 1–11.
[19] N. Alizadeh and F. Castor, Green AI: A preliminary empirical study on energy consumption in dl models across different runtime infrastructures, *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, New York, USA, 2024, 134–139.
[20] W. Shafik, A. Tufail, A. Namoun, L. C. De Silva, and R. A. A. H. M. Apong, A systematic literature review on plant disease detection: Motivations, classification techniques, datasets, challenges, and future trends, *IEEE Access*, 11, 2023, 59174–59203.
[21] F. Zubair, M. Saleh, Y. Akbari and S. Al Maadeed, A robust ensemble model for plant disease detection using deep learning architectures, *AgriEngineering*, 7(5), 2025, 159.
[22] X. Wu, X. Fan, P. Luo, S. D. Choudhury, T. Tjahjadi and C. Hu, From laboratory to field: Unsupervised domain adaptation for plant disease recognition in the wild, *Plant Phenomics*, 5, 2023, 38.

APPENDIX 1: Dataset Sample Images

0.	Chilli Anthracnose Fruit (300 images)		12.	Potato Healthy Fruit (300 images)	
1.	Chilli Bacterial Spot Leaf (300 images)		13.	Potato Healthy Leaf (300 images)	
2.	Chilli Healthy Leaf (300 images)		14.	Potato Phytopthora Infestans Leaf (300 images)	
3.	Chilli Healthy Fruit (300 images)		15.	Tomato Anthracnose Fruit (300 images)	
4.	Chilli Mosaic Virus Leaf (300 images)		16.	Tomato Bacterial Spot Leaf (300 images)	
5.	Eggplant Cercospora Leaf Spot (300 images)		17.	Tomato Early Blight Leaf (300 images)	
6.	Eggplant Colorado Potato Beetle Leaf (300 images)		18.	Tomato Healthy Fruit (300 images)	
7.	Eggplant Fruit Rot (300 images)		19.	Tomato Healthy Leaf (300 images)	
8.	Eggplant Healthy Fruit (300 images)		20.	Tomato Late Blight Leaf (300 images)	
9.	Eggplant Healthy Leaf (300 images)		21.	Tomato Leaf Mold (300 images)	
10.	Potato Alternaria Solani Leaf (300 images)		22.	Tomato Yellow Leaf Curl Virus (300 images)	
11.	Potato Common Scab Fruit (300 images)