AI Driven Precision Agriculture For Early Disease Detection And Sustainable Crop Protection

By

Okoh Emmanuel Chukwuebuka

A00020997

Submitted to

The University of Roehampton

In partial fulfilment of the requirements

for the degree of

Master of Science

in

Data Science

Declaration

I hereby certify that this report constitutes my own work, that where the language of others is used, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions, or writings of others.

I declare that this report describes the original work that has not been previously presented for the award of any other degree of any other institution.

Okoh Emmanuel Chukwuebuka

17th August, 2025

Acknowledgements

First and foremost, I give thanks to God for His grace, guidance, and strength throughout the course of my studies and the completion of this dissertation.

I am deeply grateful to my parents, **Mr. and Mrs. Godwin Okoh**, for their unwavering moral and financial support. Their sacrifices and encouragement have made it possible for me to further my education, and for that, I will always remain indebted.

My sincere appreciation goes to my close friend, **Adejoh Obetule Daniel**, who has been like a brother to me from the very beginning. His constant support, encouragement, and belief in me have been instrumental in my journey, and he is truly part of the reason I stand where I am today.

I would also like to express my heartfelt thanks to my supervisor, **Mr. Karim Bouzoubaa**, for his invaluable guidance, understanding, patience, and mentorship. His insights and support have been crucial to the success of this project.

Finally, I extend my gratitude to all those who, in one way or another, contributed to the successful completion of this work. Your support has been deeply appreciated.

Abstract

Plant diseases cause between 20-40% of annual crop losses worldwide, with especially severe consequences for smallholder farmers in sub-Saharan Africa who face limited access to diagnostics and affordable digita tools. While deep learning has shown promise for disease detection, existing approaches often rely on controlled datasets such as PlantVillage,where reported accuracies exceed 99% but can drop below 40% in real field conditions. This performance gap highlights the urgent need for robust, interpretableand deployable systems suited to real-world agriculture.

The literature review identfied some major gaps. First, although datasets like PlantVillage have been widely usede, their laboratory settings limit field transferability. Recent alternatives such as Plantdoc and multimodal guidelines attempt to improve realism, yet they suffer from inconsistent labels and limited class coverage. Secondly, methodological advances ranging from ensembles of CNNs and GAN-enhanced transformers to Vision Transformers with mixture-of-experts demonstarted accuracy gains but often at the expense of increased complexity, poor interpretability or lackk of generalization. Finally , adoption studies revealed usability barriers, as seen with the low uptake of the Plantvillage Nuru app, emphasizing the need for accessible, lightweight deployment tools.

To address these gapsb, this project developed a two-stage convolutional pipeline trained on the recent PlantWild dataset, which offers 89 in-the-wild classes across varied crops and conditions. Stage-1 served as a binary classifier (healthy vs diseased), while stage-2 performed fine-grained classificationacross 59 diseased classes. The models were optimized using Adam, SGD, and RMSprop with RMSprop consistently emerging as the strongest performer. Class weights were applied to address dataset imbalance while statistical robustness was validated through cross-validationand bootstrap confidence intervals. Interpretability was achieved using Gradient-weighted Class Activation Maps (Grad-CAM), enabling visual explanations of disease localization. Deployment was realized through a Flask and Pyngrok interface, alowing real-time testing with plant-images.results demonstarted strong generalization under field-like conditions. In Stage-1, the RMSprop model achievd 94% training accuracy and about 895 test accuracy, rflecting effective generalization beyond laboratory conditions. In Stage-2, the model achieved 73% top-1 accuracy , 92% top-3 accuracy and 955 top-5 accuracy across 59 disease categories. Confirming robustness in fine-grained recognition. These findings compare favourably to prior works, outperforming single-dataset or lab-only approaches while offering interpretability and lightweight deployment absent in more complex transformer-based or GAN-driven models. In summary, the study contributes balanced , field-ready solution to plant disease detection by combining dataset realism (via PlantWild), robust evaluation, explainability, and accessible deployment, the system offers a scalable pathway for early disease detection that can enhance food security and sustainable crop protection globally, with immediate relevance for small holder farmers in Africa.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Problem Description, Context, and Motivation

What is the problem?

Plant diseases pose a persistent threat to agricultural productivity, leading to substantial yield losses and economic damage. Globally, pests and diseases are responsible for **20–40% of crop losses** annually, amounting to more than USD 220 billion in economic impact [1]. The challenge is especially acute in low-resource settings, where traditional disease detection methods such as manual inspection and laboratory testing are slow, labour intensive, and prone to human error.

Who is affected?

Smallholder farmers in sub-Saharan Africa are among the most vulnerable, as agriculture remains the primary source of livelihood for millions [2]. Limited access to diagnostic laboratories, agricultural extension services, and affordable digital tools leaves these farmers exposed to delayed or inaccurate disease detection. The problem also affects commercial-scale agriculture globally, where early and accurate disease detection is critical to maintaining yields and reducing costs.

Where and when does the problem occur?

The problem occurs across diverse agricultural environments, from subsistence farms in rural Africa to high-tech commercial operations worldwide. It is most damaging during active growing seasons when disease spread can be rapid. Environmental variability such as fluctuating light conditions, leaf orientation, background clutter, and occlusion intensifies the challenge, particularly in real-world, open-field settings [3].

Why is it important to solve the problem?

Addressing this problem is vital for improving food security, reducing reliance on excessive pesticide use, and ensuring sustainable crop production. Artificial Intelligence (AI), particularly Convolutional Neural Networks (CNNs), offers a promising approach. While CNN-based models have achieved over 99% accuracy in controlled datasets like PlantVillage [4], performance can drop significantly to below 40% when tested in field conditions [3]. This performance gap, combined with the “black box” nature of deep learning models, limits adoption by farmers who need transparent, interpretable, and reliable tools.

The PlantWild dataset provides a potential solution by offering 18,542 in-the-wild images across 89 disease classes, capturing the complexity of real-world agricultural environments [5]. This dataset enables the training of models that can generalise better to varied field conditions, supporting both practical deployment and user trust.

1.2 Objectives

Aim

To develop a robust, interpretable AI-driven system for early plant disease detection using the PlantWild dataset, with reliable performance in real-world agricultural environments, focusing on African contexts and scalable for global deployment.

Specific Objectives

- Conduct a critical literature review of AI-based plant disease detection methods, focusing on field deployment challenges.
- Train and optimise a CNN in a two-stage classification pipeline using PlantWild data.
- Integrate Class Activation Maps (CAM) for visual interpretability and user trust.
- Develop a Flask-based web application with pyngrok for accessible testing and demonstration.
- Evaluate the system with multiple performance metrics and qualitative interpretability analysis.

1.3 Methodology (Overview)

Design

The system follows a two-stage classification process:

- **Stage 1**: A binary CNN model classifies plant images as healthy or diseased.
- **Stage 2**: If Stage 1 detects diseased, the image is passed to a multi-class CNN trained exclusively on diseased classes to identify the specific disease type. This design minimises computational overhead and ensures Stage 2 processing is applied

only when necessary. Deployment will be via a Flask-based web interface connected through a pyngrok domain for external access during testing.

Testing and Evaluation

Evaluation will be comprehensive, incorporating:

- **Metrics**: F1-score, recall, precision, AUC, accuracy, and loss.
- **Analysis Tools**: Confusion matrix, bootstrap analysis, statistical significance testing, cross-validation, error analysis, and misclassification study.
- **Visualisation**: Graphs and heatmaps to present results clearly.
- **Benchmarking**: Comparative analysis against existing solutions in the field.

Project Management

The project will follow Agile methodology with iterative development and evaluation. Progress tracking will be managed through a Trello board and a Gantt chart to monitor milestones, deadlines, and task dependencies.

Technologies and Processes

- **Dataset**: PlantWild (18,542 images, 89 classes).
- **Modeling**: Python, TensorFlow/Keras.
- **Explainability**: Class Activation Maps (CAM) for interpretability.
- **Class Balancing**: Class weights instead of augmentation.
- **Deployment**: Flask web app served through pyngrok for remote testing and demonstration.

1.4 Legal, Social, Ethical and Professional Considerations (Preview)

Legal

- Compliance with GDPR when collecting or storing new image data [6].

- Proper attribution of datasets, code, and libraries.
- Use of disclaimers to state the research-prototype status of the system.

Social

- Ensuring accessibility for rural farmers through lightweight, potentially offline-capable solutions [2].
- Designing affordable, user-friendly tools to reduce adoption barriers.
- Supporting local agricultural knowledge instead of replacing it.

Ethical

- Minimising dataset bias to avoid disadvantaging specific crops or regions [7].
- Providing interpretable outputs to promote informed decision-making.
- Ensuring fairness, accountability, and transparency in AI deployment.

Professional

- Maintaining academic integrity through accurate reporting and credible sourcing.
- Documenting code, processes, and experimental setups for reproducibility.

1.5 Background

Historically, plant disease detection has relied on manual inspection and laboratory analysis [8]. These methods are time-consuming, require expert knowledge, and can delay intervention. Advances in AI, particularly CNNs, have enabled high accuracy automated detection in controlled datasets [4]. However, the transition from laboratory to field settings remains challenging due to environmental variability [3].

Smaller datasets like PlantDoc (2,598 images, 27 classes) [9] lack the diversity required for strong generalisation, while PlantWild offers greater scale and real-world variability [5]. Integrating CAM into CNN workflows can improve transparency by highlighting the areas of an image influencing a decision [10], thereby increasing user trust.

1.6 Structure of Report

Chapter 1 – Introduction

- Outlines the problem, identifies who is affected, and explains where and when it occurs.
- States why solving the problem is important in both African and global contexts.
- Defines the project's aim and objectives.
- Provides an overview of the methodology, including the two-stage classification design, testing and evaluation strategy, project management approach, and technologies used.
- Summarises relevant legal, social, ethical, and professional considerations.
- Presents background information linking traditional methods, AI developments, and the chosen dataset.

Chapter 2 – Literature and Technology Review

- Reviews research on plant disease detection, datasets, and explainability techniques.
- Compares existing models and identifies research gaps.

Chapter 3 – Implementation

- Details the two-stage CNN architecture, class weighting, CAM integration, and Flask/pyngrok deployment.
- Outlines the development process and project management approach.

Chapter 4 – Evaluation and Results

- Presents performance metrics, interpretability analysis, and comparative benchmarking.

Chapter 5 – Conclusion and Future Work

- Summarises findings and contributions.
- Suggests improvements and future research directions.

Chapter 2 Literature – Technology Review

2.1 Literature Review

2.1.1 Dataset realism and field applicability

The quality and nature of training data greatly influence the real-world performance of plant disease detection models. While datasets such as PlantVillage have driven rapid progress, their controlled and homogeneous images often fail to capture the unpredictable variability of farm environments. Wei *et al.* [11] addressed this limitation by introducing PlantWild, a collection of 18,542 images spanning 89 disease classes, all captured under natural, uncontrolled conditions. The inclusion of images with varied lighting, occlusion, and complex backgrounds revealed weaknesses in models trained solely on curated datasets, providing a more realistic benchmark for deployment-focused research. However, even large in-the-wild datasets can face challenges, such as imbalanced representation across crops and diseases, which may bias classifiers toward dominant classes. By using PlantWild, this project ensures that evaluation conditions mirror actual field scenarios, while class weighting mitigates imbalance without resorting to synthetic augmentation.

2.1.2 Dataset limitations and diversity strategies

Beyond single datasets, diversity in image sources is increasingly recognised as a driver of model robustness. Xu *et al.* [12] examined popular datasets and identified narrow species coverage and limited environmental variation as recurring weaknesses, proposing guidelines for challenge-oriented collections that better reflect the realities of agricultural practice. Krishna *et al.* [13] tested this idea by combining PlantDoc with web-sourced images, achieving improved accuracy across varied conditions. While their approach highlights the benefit of diversity, it also raises concerns about label consistency and annotation quality when merging datasets from multiple sources. In the current project, PlantWild's intrinsic diversity reduces the need for cross-dataset mixing, and the two-stage design further minimises confusion by isolating the simpler healthy/diseased decision from the more complex disease classification task.

2.1.3 Architectural approaches for robustness

Choosing the right architecture is as critical as selecting the right data. Ali *et al.* [14] demonstrated that ensembles of deep CNNs can achieve near-perfect accuracy on PlantVillage, suggesting that

model diversity strengthens classification reliability in controlled environments. Yet such results often fail to translate into field success, as these models can be computationally heavy and optimised for unrealistic conditions. Hernández *et al.* [15] offered a more grounded perspective by testing CNNs and Vision Transformers in vineyards for downy mildew detection, achieving about 91% accuracy alongside strong localisation performance. Their integration of visual explanation tools ensured that the focus aligned with actual symptomatic regions, a step crucial for building end-user trust. This project draws on those lessons by employing Grad-CAM visualisations in Stage 2, ensuring that accuracy is paired with interpretable outputs and adaptable complexity.

2.1.4 Balancing performance and efficiency

While large and complex models often dominate benchmark leaderboards, they may not be suitable for real-world deployment where hardware resources are limited. Li *et al.* [16] introduced PMVT, a lightweight Vision Transformer with under one million parameters, delivering competitive accuracy on mobile devices. This innovation underscores the potential of efficient architectures in extending AI's reach to resource-constrained environments. Salman *et al.* [3] tackled a different challenge domain shift by combining a Vision Transformer with a Mixture-of-Experts approach, enabling the model to adapt dynamically to varying image conditions and improving cross-domain performance. Despite their promise, both methods introduce trade-offs: ultra-light models may struggle with fine-grained distinctions, while expert routing adds architectural complexity. The present project's two-stage approach offers a balanced path, reducing computation in the first stage and focusing complexity where it is most needed in the second.

2.1.5 Explainability for trust and adoption

The value of explainable AI in agriculture lies in its ability to make predictions transparent and actionable for farmers. Hernández *et al.* [16] demonstrated how saliency maps could confirm whether a model's focus aligned with diseased regions in real vineyard images, improving interpretability and trust. Singh *et al.* [17] extended the scope of fine-grained feature detection by integrating GAN-generated synthetic images with a Vision Transformer, enhancing the model's ability to discriminate subtle disease features. While generative augmentation proved beneficial in their study, it also raises concerns about introducing unrealistic artefacts. The current project takes a more conservative route, employing Grad-CAM to produce heatmaps for each prediction, giving farmers and agronomists a visual reference without altering the training distribution.

2.1.6 Deployment challenges and adoption barriers

Even with accurate and interpretable models, adoption in the field remains inconsistent. Zandjanakou-Tachin *et al.* [18] examined the uptake of the PlantVillage Nuru app among cassava farmers in Benin, finding adoption rates of only 14.1%. Barriers such as smartphone ownership, digital literacy, and user training significantly influenced engagement, showing that technical performance alone does not guarantee impact. George *et al.* [19] reinforced this point in their survey, calling for domain generalisation, transparency, and richer evaluation metrics beyond accuracy. These findings directly inform this project's choice to develop a Flask and pyngrok-based web interface, lowering the entry barrier for trials, and to include a broad evaluation suite accuracy, precision, recall, F1-score, ROC-AUC, confusion matrices, and agreement measures ensuring a more complete and reliable performance assessment.

2.1.7 Identified Gaps- The Way Forward?

The reviewed literature converges on several gaps that guide the present work: the need for realistic datasets that mirror field variability, architectures resilient to diverse and shifting conditions, integrated explainability to build trust, deployment strategies that consider accessibility, and comprehensive evaluation frameworks that go beyond single metrics. By selecting PlantWild as the primary dataset, adopting a two-stage classification strategy, integrating Grad-CAM visual explanations, and deploying through an accessible web interface, this project, contributes to current best practices while addressing these critical gaps in a targeted and practical manner.

2.2 Technology Review

2.2.1 Deep learning frameworks for model development

TensorFlow is widely used for plant disease detection due to its GPU acceleration, Keras integration, and deployment flexibility [19], [20]. In this project, both stages were built in TensorFlow, enabling a consistent workflow from preprocessing to deployment. While PyTorch offers flexibility [21], TensorFlow was chosen for its stability and compatibility with the Flask deployment setup.

2.2.2 Model architectures and design choices

The two-stage design addresses the difficulty of multi-class classification in varied conditions [16], . Stage 1 uses MobileNetV2 for quick healthy/diseased screening, while Stage 2 applies a deeper CNN to 59 diseased classes from PlantWild [11]. Other studies use EfficientNet or Vision Transformers [16], [21], but these often require more computational resources.

2.2.3 Class imbalance strategies

Imbalance in datasets like PlantWild can bias predictions [11], [13]. While oversampling and augmentation are common [17], they risk introducing artefacts. This project uses class weighting, as recommended by Xu *et al.* [11] and George *et al.* [19], to improve recall for minority classes without altering the dataset distribution.

2.2.4 Evaluation metrics and performance analysis

Literature emphasises going beyond accuracy [18], [19]. This project uses precision, recall, F1-score, ROC-AUC, Cohen's kappa, Matthews correlation coefficient, and confusion matrices, with additional bootstrap analysis and misclassification studies in Stage 2. This ensures transparent, class-balanced performance assessment.

2.2.5 Explainability tools for model interpretation

Grad-CAM is an established method for visualising model attention in agricultural AI [15], [17]. It is applied in Stage 2 here to show which leaf areas influenced classification, helping users interpret results. This avoids the extra computational burden of alternatives like LIME or SHAP [22].

2.2.6 Deployment frameworks and accessibility

Deployment uses Flask with pyngrok to provide a public URL for remote access. This approach offers low cost and quick setup compared to mobile-native apps [16] or cloud APIs [18], making it suitable for early trials in environments with limited infrastructure.

2.2.7 Summary of technology choices

The technology stack TensorFlow for both stages, MobileNetV2 for efficiency, class weights for imbalance handling, Grad-CAM for interpretability, and Flask plus pyngrok for accessible

deployment was selected to balance accuracy, efficiency, transparency, and real-world usability, aligning with best practices in recent agricultural AI research.

2.3 Summary

The literature and technology reviews collectively highlight that while deep learning has shown strong potential for plant disease detection, real-world deployment is hindered by limitations in dataset diversity, robustness to environmental variability, interpretability, and practical accessibility.

The table below summarises the key contributions, methods, metrics, limitations, and direct relevance to this project, providing a critical overview that informs the methodology.

Table 1 Summarized comparative and critical review of key studies in plant disease recognition.

Citation	**Method**	**Dataset**	**Metrics**	**Limitations**	**Relevance to This Project**
Wei et al. (2024)	Developed PlantWild with multimodal annotations for in-the-wild plant disease recognition	PlantWild	Accuracy, multimodal baselines	Class imbalance remains, some diseases under-represented	PlantWild adopted as main dataset; addressed imbalance using class weights
Xu et al. (2024)	Proposed guidelines for building realistic plant disease datasets	Multiple datasets reviewed	N/A	No new dataset released, guidance only	Confirms importance of realistic datasets, justifying PlantWild selection
Krishna et al. (2025)	Multi-dataset training for	PlantDoc + web images	Accuracy (~80%)	Label inconsistencies	Validates diversity principle; this project uses

Citation	Method	Dataset	Metrics	Limitations	Relevance to This Project
	improved generalisation			and annotation quality issues	inherent PlantWild variability instead
Ali et al. (2024)	Ensemble of CNN architectures for higher accuracy	PlantVillage	Accuracy (99.9%)	Lab images, poor field transferability	Highlights potential of architectural diversity; informs robust CNN design
Hernández et al. (2024)	Field-based CNN and ViT with localisation and explainability	Vineyard dataset	Accuracy (~91%), IoU	Limited to one crop and disease	Supports Grad-CAM integration for trust and field applicability
Li et al. (2023)	Lightweight Vision Transformer (PMVT) for mobile	Custom field datasets	Accuracy, model size	May struggle with very fine-grained classification	Shows feasibility of efficient models; informs potential mobile adaptation
Salman et al. (2025)	Vision Transformer with Mixture-of-Experts for domain adaptation	PlantVillage + PlantDoc	Accuracy (+20% in mixed conditions)	Added architectural complexity	Supports condition-aware design; mirrors two-stage pipeline's separation strategy

Citation	Method	Dataset	Metrics	Limitations	Relevance to This Project
Singh et al. (2024)	GAN-enhanced Vision Transformer for fine-grained detection	Multiple leaf datasets	Accuracy	Risk of unrealistic artefacts from synthetic images	Confirms benefit of fine detail attention; project opts for Grad-CAM without GANs
Zandjanakou-Tachin et al. (2024)	Adoption study of PlantVillage Nuru app	Nuru field data (cassava)	Adoption rate (14.1%)	Low uptake due to access and literacy barriers	Justifies web-based Flask + pyngrok prototype for low entry barrier
George et al. (2025)	Survey on deep plant disease detection trends and gaps	Review	N/A	Highlights lack of domain generalisation and robust evaluation	Confirms need for broad metrics, interpretability, and domain testing in project

Chapter 3 Implementation

3.1 System Workflow Overview

The implementation of this project follows a structured two-stage workflow that integrates data preparation, model training, and deployment into a unified AI pipeline for early plant disease detection. **Figure 1** presents the complete workflow, starting from dataset preparation and progressing through Stage 1 (binary classification), Stage 2 (multiclass classification), and deployment via a Flask–pyngrok framework. This structured process ensures robustness, reproducibility, and scalability in real-world agricultural scenarios.

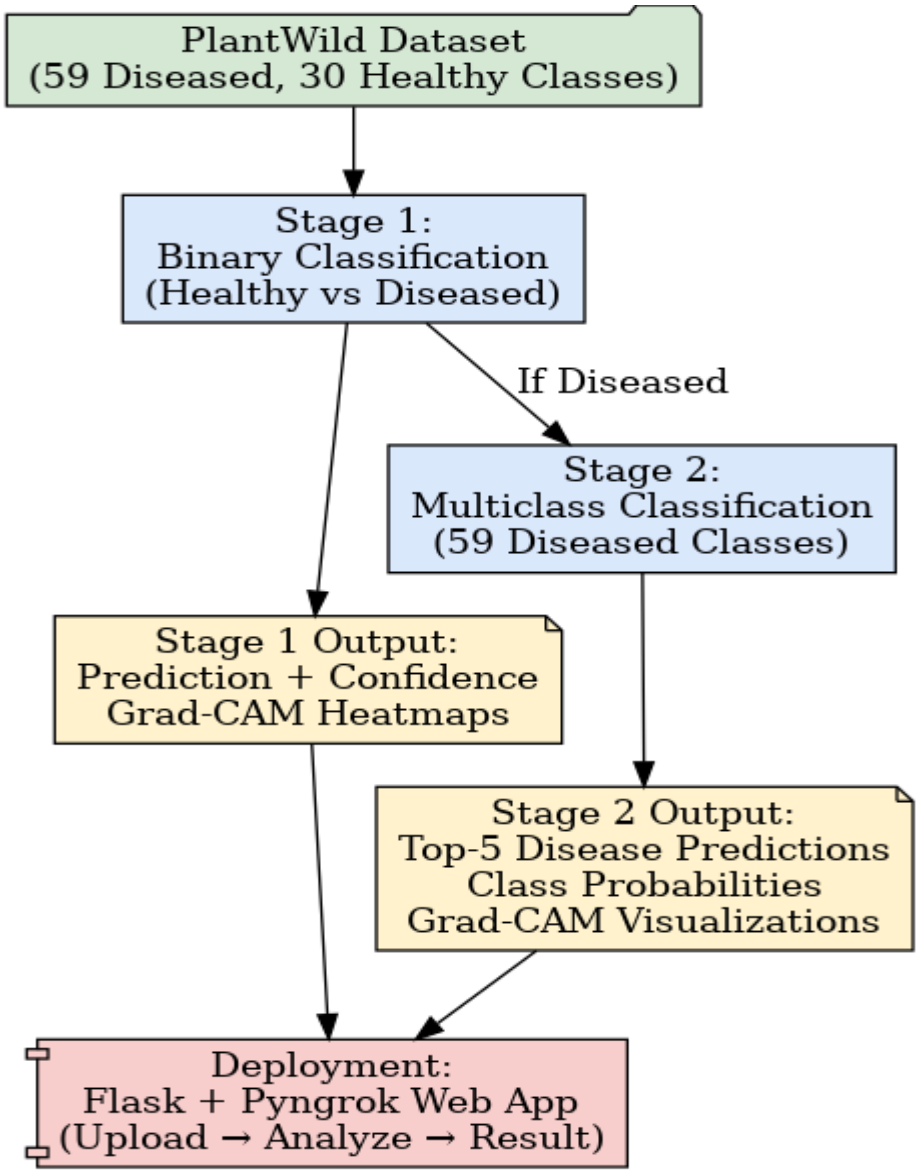

Figure 1 Workflow of the two-stage ai-driven precision agriculture system.

3.2 Dataset Description and Rationale

A critical foundation of the system is the choice of dataset. This study employed the PlantWild dataset, one of the largest and most diverse resources available for plant disease detection in real-world conditions.

The dataset contains 18,542 annotated images across 89 total classes, divided into 59 diseased categories and 30 healthy categories. Images were collected in the wild, meaning they capture natural lighting, occlusion, background noise, and varying image qualities. This contrasts with curated datasets such as PlantVillage, which consist largely of laboratory-captured images under controlled conditions [4].

Representative image samples are provided in Figure 2 to illustrate the dataset's diversity and visual complexity. These samples highlight the heterogeneity of capture conditions and disease manifestations, demonstrating why PlantWild was chosen as a realistic benchmark for this project.

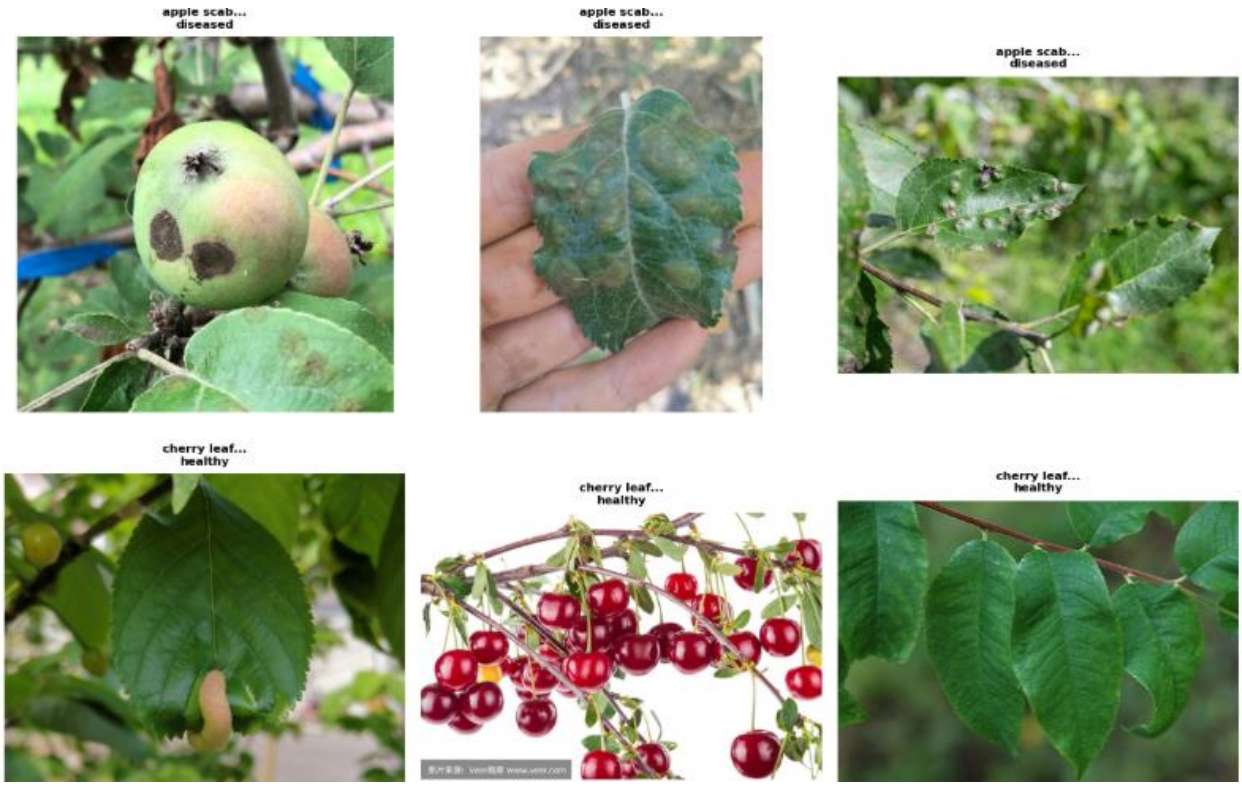

Figure 2 Sample images from the plantwild dataset.

3.2.1 Dataset Comparison

To validate the choice of PlantWild, **Table 1** compares its characteristics against widely used benchmarks such as PlantVillage, PlantDoc and other relevant datasets**[**5].

Table 2 Dataset comparison part 1: Technical Overview.

Dataset Name	**Who Created It?**	**Year**	**Crops Covered**	**Diseases Covered**	**No. of Images**	**Real Field Images?**	**Image Source**
PlantWild (Chosen Dataset)	University of Queensland (PlantWild Team)	2024	33	89	18,542	Yes (in-the-wild)	Web-sourced (Google, Ecosia, Baidu)
PlantSeg	University of Queensland, CSIRO	2025	34	115	11,458 diseased + 8,000 healthy	Yes (in-the-wild)	Web-sourced (Google, Bing, Baidu)
PlantVillage	Penn State University	2015	14	38	54,309	No (lab-only)	Laboratory
PlantDoc	IIIT Hyderabad, India	2020	13	27	2,598	Yes	Real farms
FieldPlant	University of Maroua, Cameroon	2023	3	27	5,170	Yes	Field & outdoor
TomatoDD	Chinese Academy of Sciences	2020	1 (Tomato)	4	4,178	No	Laboratory

MVTec MSVDD	Huazhong University	2022	3	5	1,000	Yes	Lab and semi-field
Leaf Segmentation Dataset	Kaggle User	2021	1 (Tomato)	1	588	Yes	Field
Plant Disease Seg. 2024	Indian Researchers	2024	1 (Tomato)	1	1,000	Yes	Field

Table 3 Dataset comparison – part 2: Annotation and Licensing.

Dataset Name	**Segmentation Labels?**	**Annotation Type**	**File Format**	**License**	**What's Special**
PlantWild (Chosen Dataset)	No	Class Label Only	JPEG	Open	Largest in-the-wild dataset with multimodal text + image data; broad species coverage
PlantSeg	Yes (expert-reviewed masks)	Pixel-wise Segmentation	JPEG + PNG + JSON	CC BY-NC-ND 4.0	Covers many crops; realistic settings; segmentation masks verified by experts
PlantVillage	No	Class Label Only	JPEG	Open (Public)	Simple, popular, but lab-only
PlantDoc	No	Class Label Only	JPEG	Open	Real-world lighting but low image count

FieldPlant	No	Class Label Only	JPEG	Open	Narrow crop variety but regionally realistic
TomatoDD	No	Bounding Boxes Only	JPEG	Open	Tomato only; narrow scope
MVTec MSVDD	No	Bounding Boxes	JPEG	Limited Use	Multi-view tomato dataset with high-quality images
Leaf Segmentation Dataset	Yes	Pixel-wise Segmentation	PNG	Free for Research	Accurate masks but too small for large-scale model training
Plant Disease Seg. 2024	Yes	Pixel-wise Segmentation	JPEG + PNG	Free with Citation	New release, but not broad enough for practical AI deployment

3.2.2 Justification for Dataset Selection

The choice of PlantWild was guided by five main factors:

- **Extensive Crop and Disease Coverage**

  PlantWild includes 89 distinct classes, spanning 59 diseased and 30 healthy plant categories, across 33 crop species. This breadth enables training models that generalise well across multiple plant types, a critical requirement for real-world deployment.

- **Real-World In-the-Wild Images**

  Unlike laboratory datasets such as PlantVillage, PlantWild images are sourced from diverse real environments, including farms and outdoor conditions. They feature realistic lighting, shadows,

and background complexity, making them highly representative of practical agricultural settings.

- **Multimodal Annotatiions**

  Each PlantWild image is accompanied by textual descriptions generated from reliable sources (Wikipedia and GPT-3.5) and verified for accuracy. This enables the use of advanced vision-language models, such as CLIP, which benefit from aligned visual and semantic information.

- **Large-Scale Dataset**

  With 18,542 images, PlantWild provides sufficient data to train deep learning models effectively while still being manageable for computational resources in academic and practical applications.

- **Publicly Available and Well-Documented**

  The dataset is openly available with clear licensing, full documentation, and reproducible benchmarks. This ensures transparency, reproducibility, and ease of integration into research pipelines.

3.3 Stage 1: Binary Classification Implementation

3.3.1 Role in the two-stage pipeline

Stage 1 functions as a screening layer that decides whether an input leaf image is healthy or diseased. This mirrors field practice where a quick check is performed before a more detailed diagnosis is attempted. A correct screen reduces the downstream search space for Stage 2 and limits the propagation of false alarms. The classifier outputs a single probability interpreted as the likelihood that the image is healthy. A decision threshold is applied at 0.5, and any image below that threshold is routed to Stage 2 for disease classification.

3.3.2 Data curation for Stage 1

The PlantWild index was filtered into two labels called healthy and diseased using the class taxonomy. Images were then assigned to train, validation, and test subsets. The split preserved the global ratio of healthy to diseased samples so that evaluation reflects the true screening workload.

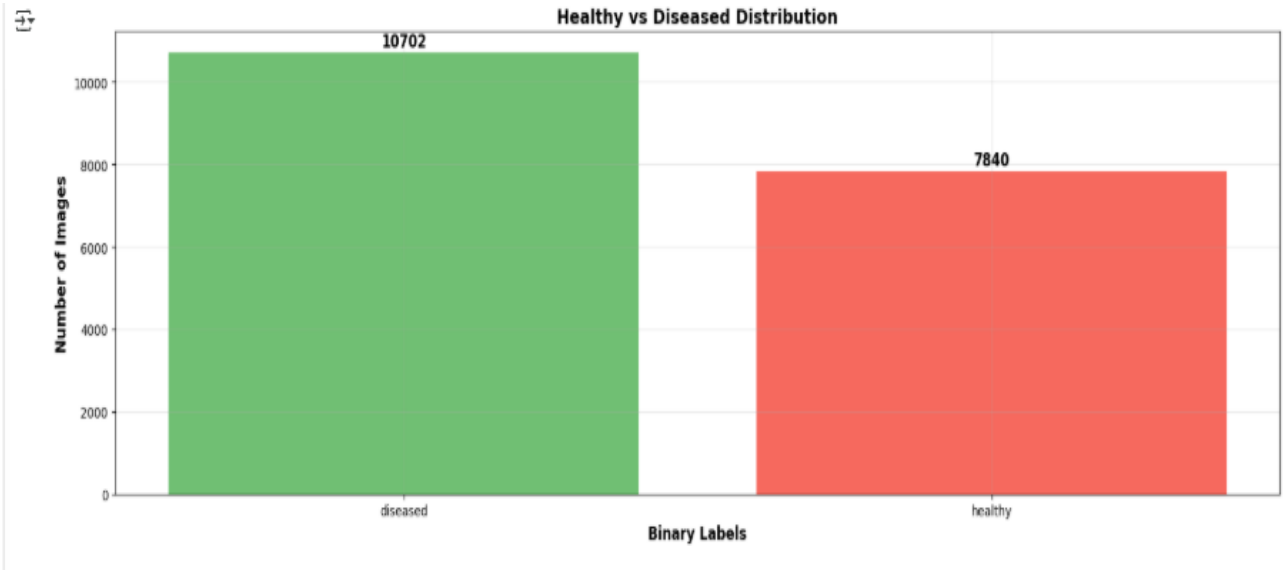

Figure 3 Binary split distribution of healthy and diseased classes: The distribution informs class weighting and threshold selection.

```
============================================================
 DATASET VISUALIZATION
============================================================

 CLASS DISTRIBUTION ANALYSIS
----------------------------------------
Total classes: 89
Total images: 18542
Train images: 13045
Validation images: 1820
Test images: 3677
```

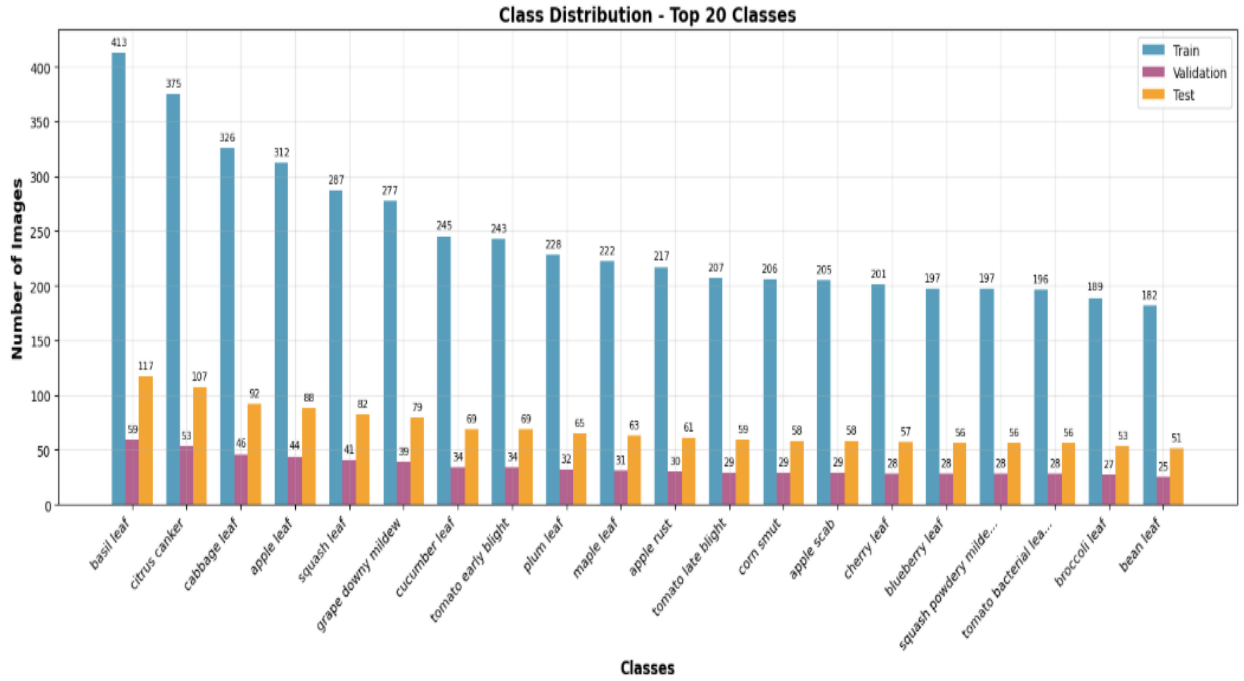

Figure 4 Top 20 class distribution contributing to Stage 1 pool with per split counts. the long tail highlights imbalance and motivates the use of class weights rather than synthetic oversampling.

3.3.3 Preprocessing and input pipeline

All images were validated for readability, converted to RGB when necessary, resized to 224 by 224 pixels, and standardised with the MobileNetV2 preprocessing function. Training batches used light spatial variation such as rotation and horizontal flips in order to reduce sensitivity to pose and framing. Validation and test batches applied only deterministic preprocessing in order to keep evaluationstable.

The data pipe line was implemented with Image DataGenerator.flow_from_dataframe which reads absolute image paths created from the PlantWild index.

3.3.4 Architecture selection and initialisation

MobileNetV2 was selected as the feature extractor because it provides a strong accuracy to efficiency ratio for leaf images and executes reliably on modest GPUs and CPUs. It uses inverted residual blocks and linear bottlenecks which preserve information while controlling the number of parameters [23]. The network was initialised with ImageNet weights and the classification head was replaced by a global average pooling layer, a small dense block with batch normalisation and dropout, and a single neuron with sigmoid activation for the healthy output.

ENHANCED MODEL ARCHITECTURE FUNCTION

```python
def create_hybrid_mobilenet_model(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3), num_classes=1, dropout_rate=0.5):
    """Create an improved MobileNetV2-based model with better architecture"""

    # Load pre-trained MobileNetV2 with better initialization
    base_model = MobileNetV2(  # Create MobileNetV2 base model
        weights='imagenet',  # Use pre-trained ImageNet weights
        include_top=False,  # Exclude classification head
        input_shape=input_shape,  # Set input image dimensions
        alpha=1.0  # Use full width for better performance
    )

    # Freeze base model initially
    base_model.trainable = False  # Prevent base model weights from updating during initial training

    # Create model with improved head
    inputs = Input(shape=input_shape)  # Define input layer
    x = base_model(inputs, training=False)  # Pass input through base model

    # Enhanced classification head
    x = GlobalAveragePooling2D()(x)  # Global average pooling to reduce spatial dimensions
    x = Dense(512, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)  # First dense layer with L2 regularization
    x = BatchNormalization()(x)  # Batch normalization for training stability
    x = Dropout(dropout_rate)(x)  # Dropout layer to prevent overfitting

    x = Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)  # Second dense layer with L2 regularization
    x = BatchNormalization()(x)  # Batch normalization for training stability
    x = Dropout(dropout_rate * 0.8)(x)  # Reduced dropout rate for second layer

    # Output layer
    outputs = Dense(num_classes, activation='sigmoid')(x)  # Final output layer for binary classification

    model = Model(inputs, outputs)  # Create complete model from inputs to outputs

    print("Model architecture created successfully!")  # Confirm model creation
    print(f"Base model parameters: {base_model.count_params():,}")  # Display base model parameter count
    print(f"Total model parameters: {model.count_params():,}")  # Display total model parameter count

    return model, base_model  # Return both complete model and base model

def compile_model(model, optimizer, learning_rate, loss='binary_crossentropy'):
    """Compile model with improved settings"""
    model.compile(  # Compile the model with specified settings
        optimizer=optimizer(learning_rate=learning_rate),  # Set optimizer with learning rate
        loss=loss,  # Set loss function
        metrics=[  # Define evaluation metrics
            'accuracy',  # Overall accuracy
            Precision(name='precision'),  # Precision metric
            Recall(name='recall'),  # Recall metric
            AUC(name='auc')  # Area under curve metric
        ]
    )
    print(f" Model compiled with {optimizer.__name__} (lr={learning_rate})")  # Confirm compilation with details
```

Figure 5 Code snippet for Stage-1 binary classification model architecture.

3.3.5 Training protocol and class imbalance strategy

Training proceeded in two phases. Phase one trained only the new head in order to stabilise gradients and adapt to the binary objective. Phase two unfroze selected layers in the feature extractor for fine tuning with a reduced learning rate. Early stopping, learning rate reduction on plateau, and model checkpointing were enabled to improve generalisation and to keep the best weights.

The dataset is imbalanced because diseased categories dominate the pool in several crops while a few healthy categories have fewer examples. Class weighting was used to counter this effect in place of synthetic augmentation. Weights were computed from counts and passed to the loss function so that minority errors have higher influence. This approach follows standard practice in imbalanced classification and avoids distribution drift that can appear with aggressive oversampling[24].

Stage 1 included a small optimisation study that compared Adam, RMSprop, and SGD with momentum. The goal was not to search exhaustively but to validate that the chosen schedule converged stably and did not overfit rapidly. Adam is adaptive and typically converges fast on image classification problems [25]. RMSprop maintains a decaying average of squared gradients and can be stable on noisy objectives. SGD with momentum remains competitive when fine tuning a pretrained backbone.

ENHANCED TRAINING FUNCTION

```python
def train_ensemble_models(train_generator, val_generator, ensemble_configs, epochs_phase1, epochs_phase2, class_weights):
    """Train ensemble models with improved training strategy"""

    trained_models = []  # Initialize list to store trained models
    training_histories = []  # Initialize list to store training histories

    for i, config in enumerate(ensemble_configs):  # Iterate through each model configuration
        print(f"\n{'='*50}")  # Print separator line
        print(f"Training Model {i+1}/{len(ensemble_configs)}: {config['name']}")  # Display current model info
        print(f"{'='*50}")  # Print separator line

        # Create model
        model, base_model = create_hybrid_mobilenet_model()  # Create new model instance

        # Phase 1: Train only the head
        print(f"\nPhase 1: Training head layers ({epochs_phase1} epochs)")  # Display phase 1 info

        # Compile for phase 1
        compile_model(model, config['optimizer'], config['learning_rate'])  # Compile model with optimizer

        # Enhanced callbacks for phase 1
        callbacks_phase1 = [  # Define training callbacks for phase 1
            EarlyStopping(  # Stop training if validation loss doesn't improve
                monitor='val_loss',  # Monitor validation loss
                patience=5,  # Wait 5 epochs before stopping
                restore_best_weights=True,  # Restore best weights when stopping
                verbose=1  # Show callback messages
            ),
            ModelCheckpoint(  # Save best model during training
                filepath=os.path.join(MODEL_DIR, f"{config['name']}_phase1_best.h5"),  # Save path
                monitor='val_loss',  # Monitor validation loss
                save_best_only=True,  # Only save best model
                verbose=1  # Show callback messages
            ),
            ReduceLROnPlateau(  # Reduce learning rate when loss plateaus
                monitor='val_loss',  # Monitor validation loss
                factor=0.5,  # Reduce learning rate by half
                patience=3,  # Wait 3 epochs before reducing
                min_lr=1e-7,  # Minimum learning rate
                verbose=1  # Show callback messages
            )
        ]

        # Train phase 1
        history_phase1 = model.fit(  # Train the model
            train_generator,  # Training data generator
            epochs=epochs_phase1,  # Number of training epochs
            validation_data=val_generator,  # Validation data generator
            callbacks=callbacks_phase1,  # Training callbacks
            class_weight=class_weights,  # Class weights for imbalanced data
            verbose=1  # Show training progress
        )
```

```
# Phase 2: Fine-tune entire model
print(f"\n Phase 2: Fine-tuning entire model ({epochs_phase2} epochs)")  # Display phase 2 info

# Unfreeze base model layers
base_model.trainable = True  # Allow base model weights to be updated

# Freeze early layers to prevent catastrophic forgetting
for layer in base_model.layers[:100]:  # Iterate through first 100 layers
    layer.trainable = False  # Freeze early layers

# Recompile with lower learning rate
fine_tune_lr = config['learning_rate'] * 0.1  # Reduce learning rate for fine-tuning
compile_model(model, config['optimizer'], fine_tune_lr)  # Recompile with new learning rate

# Enhanced callbacks for phase 2
callbacks_phase2 = [  # Define training callbacks for phase 2
    EarlyStopping(  # Stop training if validation loss doesn't improve
        monitor='val_loss',  # Monitor validation loss
        patience=8,  # More patience for fine-tuning
        restore_best_weights=True,  # Restore best weights when stopping
        verbose=1  # Show callback messages
    ),
    ModelCheckpoint(  # Save best model during training
        filepath=os.path.join(MODEL_DIR, f"{config['name']}_final_best.h5"),  # Save path
        monitor='val_loss',  # Monitor validation loss
        save_best_only=True,  # Only save best model
        verbose=1  # Show callback messages
    ),
    ReduceLROnPlateau(  # Reduce learning rate when loss plateaus
        monitor='val_loss',  # Monitor validation loss
        factor=0.3,  # More aggressive learning rate reduction
        patience=4,  # Wait 4 epochs before reducing
        min_lr=1e-8,  # Minimum learning rate
        verbose=1  # Show callback messages
    )
]

# Train phase 2
history_phase2 = model.fit(  # Fine-tune the model
    train_generator,  # Training data generator
    epochs=epochs_phase2,  # Number of fine-tuning epochs
    validation_data=val_generator,  # Validation data generator
    callbacks=callbacks_phase2,  # Training callbacks
    class_weight=class_weights,  # Class weights for imbalanced data
    verbose=1  # Show training progress
)
```

```
        # Save final model
        final_model_path = os.path.join(MODEL_DIR, f"{config['name']}_final.h5")  # Define final model path
        model.save(final_model_path)  # Save the trained model

        # Store results
        trained_models.append({  # Add model info to results list
            'model': model,  # Store model instance
            'config': config,  # Store configuration
            'path': final_model_path  # Store model file path
        })

        # Combine histories
        combined_history = {  # Combine training histories from both phases
            'loss': history_phase1.history['loss'] + history_phase2.history['loss'],  # Combine training losses
            'val_loss': history_phase1.history['val_loss'] + history_phase2.history['val_loss'],  # Combine validation losses
            'accuracy': history_phase1.history['accuracy'] + history_phase2.history['accuracy'],  # Combine training accuracies
            'val_accuracy': history_phase1.history['val_accuracy'] + history_phase2.history['val_accuracy']  # Combine validation accuracies
        }
        training_histories.append(combined_history)  # Add combined history to list

        print(f" Model {config['name']} training completed!")  # Confirm model training completion
        print(f"Final model saved to: {final_model_path}")  # Display model save location

    print(f"\n All {len(ensemble_configs)} models trained successfully!")  # Confirm all models trained
    return trained_models, training_histories  # Return trained models and histories
```

Figure 6 Code snippet of training functions for Stage 1 with two-phase training loop for three optimizers using class weights.

TRAINING ENSEMBLE MODELS

```
[ ]  # Train the ensemble models
     print(" Starting ensemble model training...")  # Inform user of training start
     print(f"Training {len(ENSEMBLE_CONFIGS)} models with configurations:")  # Display number of models to train
     for i, config in enumerate(ENSEMBLE_CONFIGS):  # Iterate through each model configuration
         print(f"  {i+1}. {config['name']}: {config['optimizer'].__name__} (lr={config['learning_rate']})")  # Display model details

     # Start training
     trained_models, training_histories = train_ensemble_models(  # Execute ensemble training function
         train_generator=train_generator,  # Pass training data generator
         val_generator=val_generator,  # Pass validation data generator
         ensemble_configs=ENSEMBLE_CONFIGS,  # Pass model configurations
         epochs_phase1=EPOCHS_PHASE1,  # Pass phase 1 epoch count
         epochs_phase2=EPOCHS_PHASE2,  # Pass phase 2 epoch count
         class_weights=class_weights  # Pass balanced class weights
     )

     print("\nTraining completed! Models saved to:", MODEL_DIR)  # Confirm training completion and save location
     print("Available models:")  # Header for model list
     for model_info in trained_models:  # Iterate through trained models
         print(f"  - {model_info['config']['name']}: {model_info['path']}")  # Display each model's name and path

     # Save training histories for later analysis
     import pickle  # Import pickle for serialization
     histories_path = os.path.join(MODEL_DIR, 'training_histories.pkl')  # Define path for training histories
     with open(histories_path, 'wb') as f:  # Open file for binary writing
         pickle.dump(training_histories, f)  # Save training histories to file
     print(f" Training histories saved to: {histories_path}")  # Confirm histories save location
```

```
Training Model 3/3: model_rmsprop
==================================================
Model architecture created successfully!
Base model parameters: 2,257,984
Total model parameters: 3,048,513
```

Figure 7 Code snippet for the two-phase training of ensemble models for Stage 1 binary classification.

3.3.6 Decision threshold, evaluation metrics, and reporting

The output is the probability that an image is healthy. The decision boundary was fixed at 0.5 for clarity. In practice the threshold can be adjusted to control the balance between missed diseases and false alarms. Evaluation reported accuracy, precision, recall, F1 score, ROC AUC, Cohen kappa, Matthews correlation coefficient when appropriate, and a confusion matrix. Macro averages were emphasised in order to avoid dominance by the majority class. This aligns with guidance in recent agricultural AI reviews that encourage evaluation beyond accuracy[27].

The confusion matrix highlights two operational risks:

- **False Negative (FN):** when a diseased sample is labelled healthy. This error can suppress Stage 2 and is therefore monitored carefully.
- **False Positive (FP):** when a healthy sample is labelled diseased. This error adds redundant Stage 2 computations and may cause unnecessary concern.

The threshold and class weights work together to limit both risks and keep the screen conservative without overwhelming Stage 2.

3.3.7 Explainability with Grad CAM

Interpretability is essential when a screening system is used by non technical users. Grad CAM produces a coarse heatmap which highlights the regions that most influence the prediction by

tracing gradients into the last convolutional feature maps[26]. Heatmaps were generated for a sample of test images in both classes. For healthy predictions the activation concentrates on leaf textures that indicate intact tissue. For diseased predictions the activation often surrounds lesions and chlorotic patches. These qualitative checks reduce the risk of shortcut learning on backgrounds or watermarks.

Grad-CAM Visualization Function

```python
# PROPER GRAD-CAM THAT ACTUALLY FOCUSES ON LEAF DISEASE AREAS

import time

def create_proper_leaf_gradcam(model, img_path):
    """Proper Grad-CAM implementation that focuses on actual leaf disease areas"""

    print(f"Processing: {os.path.basename(img_path)}")

    try:
        # Load and preprocess image
        img = tf.keras.preprocessing.image.load_img(img_path, target_size=(IMG_HEIGHT, IMG_WIDTH))
        img_array = tf.keras.preprocessing.image.img_to_array(img)
        original_img = img_array.astype(np.uint8)

        # FIXED: Preprocess for model with proper tensor conversion
        img_array_processed = tf.keras.applications.mobilenet_v2.preprocess_input(img_array.copy())
        img_array_processed = tf.expand_dims(img_array_processed, axis=0)
        img_array_processed = tf.convert_to_tensor(img_array_processed, dtype=tf.float32)  # FIXED

        # Get prediction
        pred = model.predict(img_array_processed, verbose=0)[0][0]
        pred_class = "Healthy" if pred > 0.5 else "Diseased"
        confidence = pred if pred > 0.5 else (1 - pred)

        # PROPER APPROACH: Create a simplified model that we can actually access
        # We'll recreate the last few layers to get proper gradients

        # Get the MobileNetV2 base model
        mobilenet_base = model.get_layer('mobilenetv2_1.00_224')

        # Create a new model that outputs both the conv features and prediction
        # This avoids the internal access issues
        new_model = tf.keras.Model(
            inputs=model.input,
            outputs=[mobilenet_base.output, model.output]
        )

        # Use GradientTape to compute gradients properly
        with tf.GradientTape() as tape:
            tape.watch(img_array_processed)  # FIXED: Now watching a proper tensor
            conv_outputs, predictions = new_model(img_array_processed)

            # For diseased prediction, maximize the diseased score
            # For healthy prediction, we still want to see what it's looking at
            if pred_class == "Diseased":
                class_output = predictions[:, 0]  # Raw output for diseased
            else:
                class_output = predictions[:, 0]  # Same - we want to see attention regardless
```

```
# Get gradients of the class output with respect to conv features
grads = tape.gradient(class_output, conv_outputs)

# Compute importance weights (global average pooling of gradients)
pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))

# Get the conv output for this image
conv_outputs = conv_outputs[0]

# Multiply each channel by its importance and sum
heatmap = tf.reduce_sum(tf.multiply(pooled_grads, conv_outputs), axis=-1)

# Apply ReLU to keep only positive influences
heatmap = tf.nn.relu(heatmap)

# Normalize the heatmap
heatmap_max = tf.reduce_max(heatmap)
if heatmap_max > 0:
    heatmap = heatmap / heatmap_max
else:
    # If no positive gradients, create a center-focused map
    h, w = heatmap.shape
    y, x = np.ogrid[:h, :w]
    center_y, center_x = h // 2, w // 2
    heatmap = tf.constant(np.exp(-((x - center_x) ** 2 + (y - center_y) ** 2) / (2 * (min(h, w) / 4) ** 2)), dtype=tf.float32)

heatmap = heatmap.numpy()

# Resize heatmap to original image size using proper interpolation
heatmap_resized = cv2.resize(heatmap, (IMG_WIDTH, IMG_HEIGHT), interpolation=cv2.INTER_CUBIC)

# Apply threshold to focus only on high attention areas
threshold = 0.3  # Only show areas with >30% attention
heatmap_focused = np.where(heatmap_resized > threshold, heatmap_resized, 0)

# If no areas above threshold, use a lower threshold
if heatmap_focused.max() == 0:
    threshold = 0.1
    heatmap_focused = np.where(heatmap_resized > threshold, heatmap_resized, 0)

# Create RGB heatmap with proper color mapping
heatmap_rgb = cv2.applyColorMap(np.uint8(255 * heatmap_focused), cv2.COLORMAP_JET)
heatmap_rgb = cv2.cvtColor(heatmap_rgb, cv2.COLOR_BGR2RGB)

# Make non-attention areas more transparent/darker
mask = heatmap_focused > 0.1
heatmap_display = heatmap_rgb.copy()
heatmap_display[~mask] = [20, 20, 60]  # Dark blue for non-attention areas

# Create focused overlay - only highlight significant attention areas
overlay_img = original_img.copy().astype(float)
high_attention = heatmap_focused > 0.4

if high_attention.any():
    # Blend only high attention areas
    overlay_img[high_attention] = (
        heatmap_rgb[high_attention] * 0.6 +
        original_img[high_attention] * 0.4
    )
```

```
        overlay_img = np.clip(overlay_img, 0, 255).astype(np.uint8)

        print(f"  Proper Grad-CAM successful! Attention range: {heatmap_focused.min():.3f} to {heatmap_focused.max():.3f}")
        print(f"     High attention areas: {high_attention.sum()} pixels")

        return original_img, heatmap_display, overlay_img, pred, pred_class, confidence

    except Exception as e:
        print(f"  Proper Grad-CAM failed: {str(e)}")

        # Enhanced fallback using image analysis
        try:
            img = tf.keras.preprocessing.image.load_img(img_path, target_size=(IMG_HEIGHT, IMG_WIDTH))
            img_array = tf.keras.preprocessing.image.img_to_array(img).astype(np.uint8)

            # Get prediction
            img_processed = tf.keras.applications.mobilenet_v2.preprocess_input(img_array.copy())
            img_processed = np.expand_dims(img_processed, axis=0)
            pred = model.predict(img_processed, verbose=0)[0][0]
            pred_class = "Healthy" if pred > 0.5 else "Diseased"
            confidence = pred if pred > 0.5 else (1 - pred)

            # Create attention based on actual image features
            # Convert to HSV for better disease detection
            hsv = cv2.cvtColor(img_array, cv2.COLOR_RGB2HSV)

            # Create mask for potential disease areas (brown, yellow, dark spots)
            # Diseased areas often have different hue/saturation
            lower_disease1 = np.array([10, 50, 50])   # Brown/yellow areas
            upper_disease1 = np.array([30, 255, 255])

            lower_disease2 = np.array([0, 50, 0])     # Dark/dead areas
            upper_disease2 = np.array([10, 255, 100])

            mask1 = cv2.inRange(hsv, lower_disease1, upper_disease1)
            mask2 = cv2.inRange(hsv, lower_disease2, upper_disease2)
            disease_mask = cv2.bitwise_or(mask1, mask2)

            # Apply morphological operations to clean up the mask
            kernel = np.ones((5,5), np.uint8)
            disease_mask = cv2.morphologyEx(disease_mask, cv2.MORPH_CLOSE, kernel)
            disease_mask = cv2.morphologyEx(disease_mask, cv2.MORPH_OPEN, kernel)

            # Create attention map
            attention_map = disease_mask.astype(float) / 255.0

            # Apply Gaussian blur for smoother attention
            attention_map = cv2.GaussianBlur(attention_map, (15, 15), 0)

            # Scale by confidence
            attention_map = attention_map * confidence

            # If no disease areas found, focus on edges/texture changes
            if attention_map.max() < 0.1:
                gray = cv2.cvtColor(img_array, cv2.COLOR_RGB2GRAY)
                edges = cv2.Canny(gray, 50, 150)
                attention_map = cv2.GaussianBlur(edges.astype(float), (15, 15), 0)
                attention_map = attention_map / attention_map.max() if attention_map.max() > 0 else attention_map
                attention_map = attention_map * confidence

            # Create RGB heatmap
            heatmap_rgb = cv2.applyColorMap(np.uint8(255 * attention_map), cv2.COLORMAP_JET)
            heatmap_rgb = cv2.cvtColor(heatmap_rgb, cv2.COLOR_BGR2RGB)

            # Create overlay
            overlay_img = heatmap_rgb * 0.4 + img_array * 0.6
            overlay_img = np.clip(overlay_img, 0, 255).astype(np.uint8)

            print(f"  Using enhanced disease-area detection")
            return img_array, heatmap_rgb, overlay_img, pred, pred_class, confidence

        except Exception as e2:
            print(f"  Enhanced fallback failed: {e2}")
            blank = np.ones((IMG_HEIGHT, IMG_WIDTH, 3), dtype=np.uint8) * 128
            return blank, blank, blank, 0.5, "Error", 0.0
```

```python
def visualize_random_leaf_gradcam(model, test_df, num_samples=4):
    """Visualization with random image selection each run"""

    # Random selection each time
    random_seed = int(time.time()) % 10000  # Different seed each run
    print(f"Using random seed: {random_seed}")

    print("="*70)
    print(" RANDOM LEAF-FOCUSED GRAD-CAM ANALYSIS")
    print("="*70)

    # Remove fixed random_state to get different images each time
    healthy_df = test_df[test_df['binary_label'] == 'healthy']
    diseased_df = test_df[test_df['binary_label'] == 'diseased']

    # Sample without fixed random_state for true randomness
    healthy_samples = healthy_df.sample(n=num_samples//2) if len(healthy_df) >= num_samples//2 else healthy_df
    diseased_samples = diseased_df.sample(n=num_samples//2) if len(diseased_df) >= num_samples//2 else diseased_df

    # Combine and shuffle randomly
    test_samples = pd.concat([diseased_samples, healthy_samples])
    test_samples = test_samples.sample(frac=1)  # Shuffle order randomly

    # Show which images were selected
    print(f"Selected {len(test_samples)} random images:")
    for idx, (_, sample) in enumerate(test_samples.iterrows()):
        print(f"  {idx+1}. {sample['class_name']} ({sample['binary_label']}) - {os.path.basename(sample['image_path'])}")

    fig, axes = plt.subplots(num_samples, 3, figsize=(16, 4*num_samples))
    if num_samples == 1:
        axes = axes.reshape(1, -1)

    success_count = 0
    for i, (_, sample) in enumerate(test_samples.iterrows()):
        img_path = sample['image_path']
        true_label = sample['binary_label']
        class_name = sample['class_name']

        print(f"\nSample {i+1}: {class_name} ({true_label})")

        original, heatmap, overlay, pred, pred_class, confidence = create_proper_leaf_gradcam(model, img_path)

        if pred_class != "Error":
            success_count += 1

        # Display
        axes[i, 0].imshow(original)
        axes[i, 0].set_title(f'Original\n{class_name}\nTrue: {true_label.title()}', fontsize=11, fontweight='bold')
        axes[i, 0].axis('off')

        axes[i, 1].imshow(heatmap)
        axes[i, 1].set_title(f'Leaf-Focused Heatmap\nRed = Disease Attention\nBlue = Background', fontsize=11, fontweight='bold')
        axes[i, 1].axis('off')

        axes[i, 2].imshow(overlay)
        axes[i, 2].set_title(f'Disease Detection\nPred: {pred_class}\nConf: {confidence:.1%}', fontsize=11, fontweight='bold')
        axes[i, 2].axis('off')
```

```python
            # Color-coded borders
            is_correct = pred_class.lower() == true_label and pred_class != "Error"
            border_color = 'green' if is_correct else 'red'

            for col in range(3):
                for spine in axes[i, col].spines.values():
                    spine.set_color(border_color)
                    spine.set_linewidth(4)
                    spine.set_visible(True)

        # Add timestamp to filename for unique saves
        timestamp = int(time.time())
        save_path = f'/content/random_leaf_gradcam_{timestamp}.png'

        plt.suptitle(f'Random Leaf-Focused Grad-CAM Analysis (Seed: {random_seed})\nDifferent Images Each Run',
                     fontsize=16, fontweight='bold')
        plt.tight_layout()
        plt.subplots_adjust(top=0.88)

        plt.savefig(save_path, dpi=300, bbox_inches='tight', facecolor='white')
        plt.show()

        print(f"\nRandom leaf-focused Grad-CAM complete!")
        print(f"Success rate: {success_count}/{num_samples}")
        print(f"Saved to: {save_path}")
        print(f"Run again to see different random images!")

    # Run the fixed version of your preferred code
    if 'best_model' in globals() and 'df' in globals():
        visualize_random_leaf_gradcam(best_model, df, num_samples=4)

        print("\n" + "="*50)
        print("TIP: Run this cell again to see different random images!")
        print("="*50)
    else:
        print("Missing variables")
```

Figure 8 Code snippets for Grad Cam using Mobilenetv2 layers for Stage 1 binary classification.

The resulting heatmaps can be overlaid on the original image for inspection by agronomists during evaluation and are also useful for model debugging.

3.3.8 Design choices, ablations, and sprint notes

A set of controlled ablations guided the final configuration.

- **Backbone freezing policy.** Freezing the entire backbone in phase one stabilised the head. Partial unfreezing in phase two improved recall on minority healthy cases without a notable drop in precision.
- **Optimiser choice.** Adam gave reliable convergence for the head and fine tuning. RMSprop was close in validation performance but showed slightly higher variance. SGD with momentum required longer schedules and produced similar accuracy when tuned. These short runs informed the final choice while keeping training cost manageable.
- **Class weighting versus augmentation.** Class weights were the primary balancing tool. Augmentation was kept conservative and used for robustness. This is consistent with the project decision to avoid synthetic inflation of the dataset and to maintain the natural distribution.

Development progressed in sprints:

- Sprint 1: built the data index and class distribution plots (Figures 3, 4).
- Sprint 2: implemented the model builder and training loop.
- Sprint 3: integrated evaluation tooling and Grad-CAM.
- Sprint 4: stabilised thresholds, confusion matrices, and error analysis.

3.3.9 Summary of Stage 1 outcomes

The binary screen delivers a fast and interpretable decision that routes only likely diseased images to Stage 2. The design balances sensitivity and specificity by combining class weighting, a cautious threshold, and a fine tuned MobileNetV2 head. The evaluation framework reports class balanced metrics and provides confusion matrices that can be inspected alongside Grad CAM overlays. This stage is therefore suitable as the front door of the diagnostic pipeline and prepares clean inputs for the multiclass classifier.

3.4 Stage 2: Multiclass Disease Classification Implementation

3.4.1 Role in the two-stage pipeline

Stage 2 is the diagnostic module that activates only when Stage 1 predicts diseased. It assigns a specific disease label among 59 classes and returns a ranked list of likely diseases. This stage completes the clinical pathway by moving from detection to identification.

3.4.2 Dataset preparation for Stage 2

The PlantWild index was filtered to retain only diseased categories. A stratified split was created for train, validation, and test so that per class proportions are maintained. The image preprocessing mirrors Stage 1. Images are validated, converted to RGB, resized to 224 by 224, and passed through the MobileNetV2 preprocessing function.

Class imbalance handling. The diseased subset exhibits a long tail distribution. Several classes have few examples and a small number of classes dominate. Class weights were computed with compute_class_weight [24] and supplied to the training loop so that minority classes contribute a larger portion of the loss. This approach follows standard practice for imbalanced learning and avoids altering the original class distribution during training .

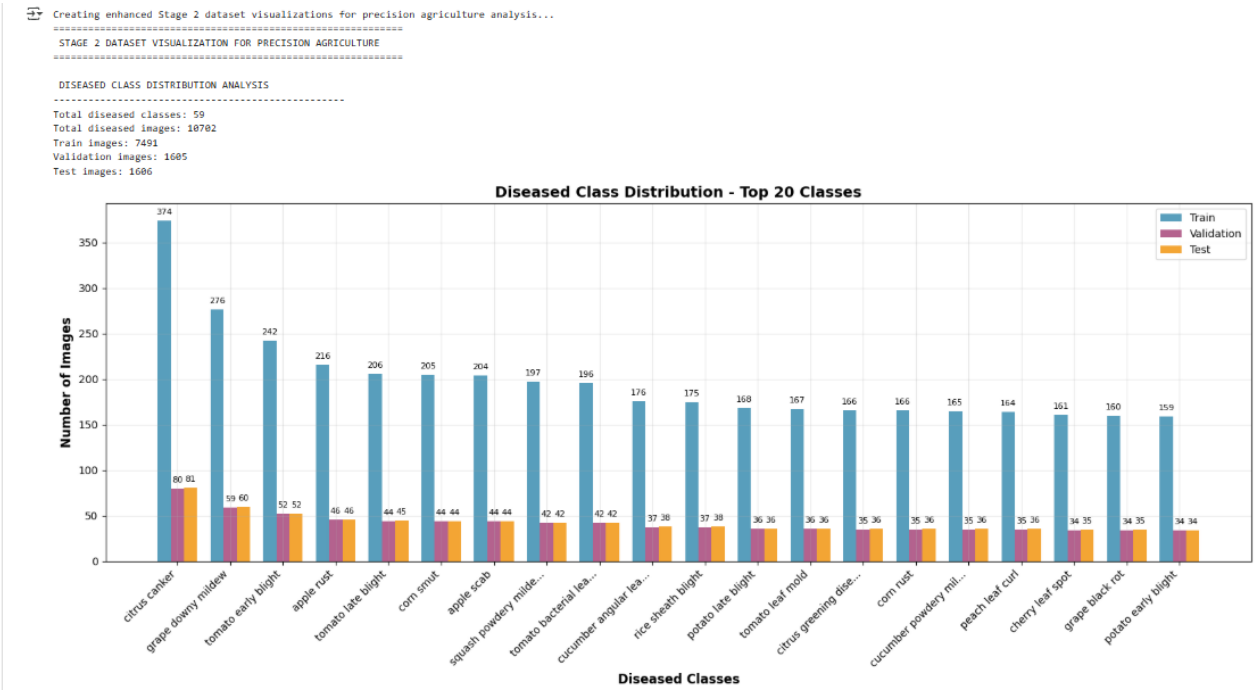

Figure 9 Top 20 diseased class distribution for stage 2. The most frequent diseased classes with train, validation, and test counts. The long tail motivates class weighting and macro level reporting.

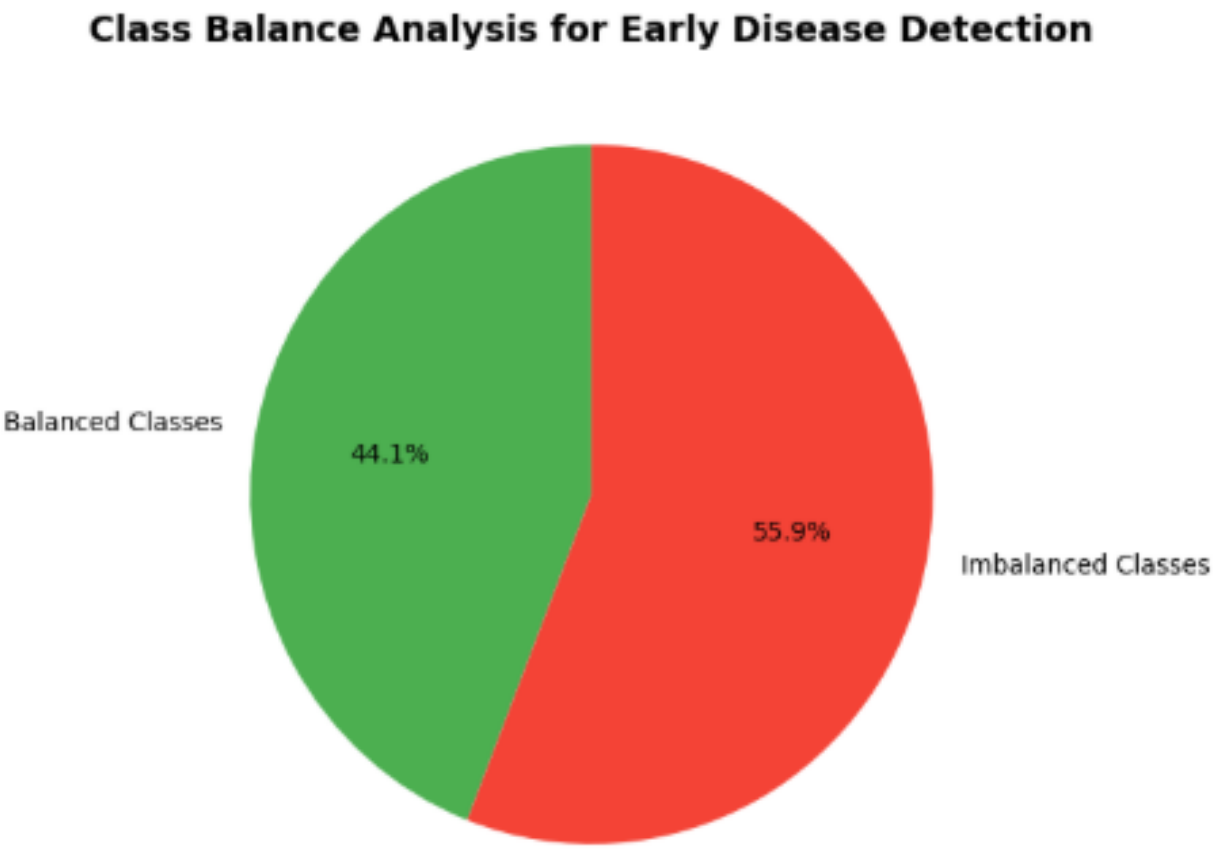

Figure 10 Imbalance analysis for diseased classes: The Pie Chart highlights percentage classes that receive higher loss weights during optimisation.

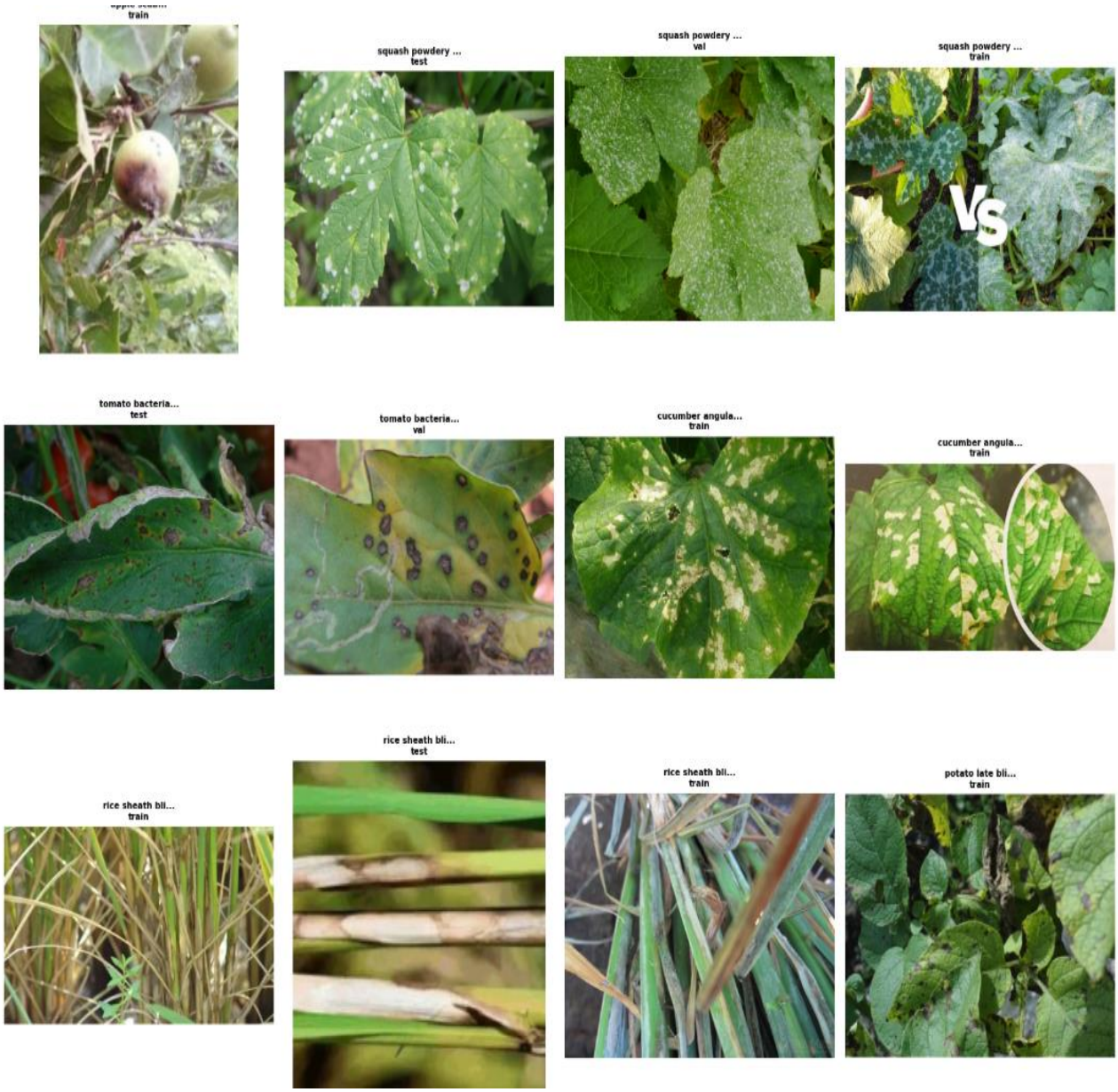

Figure 11 Sample images per Diseased class: Representative examples from Multiple Diseased Classes. (The images show variation in lesion size, colour change, and background clutter typical of field conditions).

3.4.3 Network architecture and head

MobileNetV2 again forms the feature extractor because it balances accuracy and inference speed for plant leaf imagery [23]. The ImageNet-initialised backbone was used with the top layer removed.

A compact classification head was added:

- Global Average Pooling,
- Batch Normalisation,
- Dropout,
- Dense layer with **59 logits** and **softmax activation**.

MODEL TRAINING

```
[ ] # ===== Stage 2: Single-model MobileNetV2 training cell (with class order) =====
    import os, json, numpy as np, pandas as pd, tensorflow as tf
    from tensorflow.keras.applications import MobileNetV2
    from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout, BatchNormalization
    from tensorflow.keras.models import Model
    from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau, CSVLogger
    from tensorflow.keras.preprocessing.image import ImageDataGenerator
    from tensorflow.keras.regularizers import l2

    # ----------------------------
    # sanity checks / defaults
    # ----------------------------
    assert 'df_diseased' in globals(), "df_diseased not found. Run dataset-prep cell first."
    need_cols = {'image_path_absolute','encoded_class_id','split_name'}
    missing = need_cols - set(df_diseased.columns)
    assert not missing, f"df_diseased is missing columns: {missing}"

    IMG_HEIGHT, IMG_WIDTH = 224, 224
    BATCH_SIZE = 32
    EPOCHS_PHASE1 = 20
    EPOCHS_PHASE2 = 15
    PATIENCE = 10
    FACTOR = 0.5
    MIN_LR = 1e-7
    DROPOUT_RATE = 0.5
    L2_REG = 1e-4
    MODEL_TAG = "stage2_mnv2"
    DRIVE_MODELS_DIR = '/content/drive/MyDrive/Stage2_Enhanced_Models'
    os.makedirs(DRIVE_MODELS_DIR, exist_ok=True)

    # ----------------------------
    # Generators with IDENTICAL class order
    # ----------------------------
    df_diseased = df_diseased.copy()
    df_diseased['encoded_class_id_str'] = df_diseased['encoded_class_id'].astype(str)

    num_classes = int(df_diseased['encoded_class_id'].nunique())
    classes_fixed = [str(i) for i in range(num_classes)]  # identical order everywhere

    train_df = df_diseased[df_diseased['split_name'] == 'train'].copy()
    val_df   = df_diseased[df_diseased['split_name'] == 'val'].copy()
    test_df  = df_diseased[df_diseased['split_name'] == 'test'].copy()

    train_idg = ImageDataGenerator(
        preprocessing_function=tf.keras.applications.mobilenet_v2.preprocess_input,
        rotation_range=20, width_shift_range=0.2, height_shift_range=0.2,
        shear_range=0.2, zoom_range=0.2, horizontal_flip=True, fill_mode='nearest'
    )
    plain_idg = ImageDataGenerator(
        preprocessing_function=tf.keras.applications.mobilenet_v2.preprocess_input
    )
```

```
[ ]  train_gen = train_idg.flow_from_dataframe(
         train_df, x_col='image_path_absolute', y_col='encoded_class_id_str',
         target_size=(IMG_HEIGHT, IMG_WIDTH), batch_size=BATCH_SIZE,
         class_mode='categorical', shuffle=True, seed=42, classes=classes_fixed
     )
     val_gen = plain_idg.flow_from_dataframe(
         val_df, x_col='image_path_absolute', y_col='encoded_class_id_str',
         target_size=(IMG_HEIGHT, IMG_WIDTH), batch_size=BATCH_SIZE,
         class_mode='categorical', shuffle=False, seed=42, classes=classes_fixed
     )
     test_gen = plain_idg.flow_from_dataframe(
         test_df, x_col='image_path_absolute', y_col='encoded_class_id_str',
         target_size=(IMG_HEIGHT, IMG_WIDTH), batch_size=BATCH_SIZE,
         class_mode='categorical', shuffle=False, seed=42, classes=classes_fixed
     )

     print(f"Train samples: {train_gen.n}")
     print(f"Val samples:   {val_gen.n}")
     print(f"Test samples:  {test_gen.n}")
     print(f"Num classes:   {num_classes}")
     print("class_indices consistent across splits:",
           train_gen.class_indices == val_gen.class_indices == test_gen.class_indices)

     # ------------------------------
     # Class weights (inverse frequency on encoded_class_id)
     # ------------------------------
     counts = train_df['encoded_class_id'].value_counts().sort_index()
     total = counts.sum()
     class_weights = {i: float(total / (num_classes * counts.get(i, 1))) for i in range(num_classes)}
     print(f"class_weight range: {min(class_weights.values()):.4f} -> {max(class_weights.values()):.4f}")

     # ------------------------------
     # Model (do NOT preprocess again inside the model; generators already do it)
     # ------------------------------
     def build_model(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3), num_classes=num_classes,
                     dropout=DROPOUT_RATE, l2_reg=L2_REG):
         base = MobileNetV2(weights='imagenet', include_top=False, input_shape=input_shape)
         base.trainable = False  # Phase 1

         inputs = tf.keras.Input(shape=input_shape)
         x = base(inputs, training=False)              # features from frozen base
         x = GlobalAveragePooling2D()(x)
         x = Dense(1024, activation='relu', kernel_regularizer=l2(l2_reg))(x)
         x = BatchNormalization()(x); x = Dropout(dropout)(x)
         x = Dense(512, activation='relu', kernel_regularizer=l2(l2_reg))(x)
         x = BatchNormalization()(x); x = Dropout(dropout)(x)
         x = Dense(256, activation='relu', kernel_regularizer=l2(l2_reg))(x)
         x = BatchNormalization()(x); x = Dropout(dropout/2)(x)
         outputs = Dense(num_classes, activation='softmax', kernel_regularizer=l2(l2_reg))(x)
         model = Model(inputs, outputs)
         return model, base
```

Figure 12 **C**ode snippet for Stage 2 multi-class (59 classes) model architecture.

3.4.4 Optimisation with AdamW and two phase schedule

The **AdamW optimiser** was selected for all Stage 2 training runs. AdamW decouples weight decay from the gradient update, improving generalisation during fine-tuning [28].

Training proceeded in two phases:

1. **Phase 1:** only the new head was trained with the backbone frozen.
2. **Phase 2:** deeper layers of the backbone were unfrozen with a reduced learning rate for fine-tuning.

```python
model, base_model = build_model()

# ------------------------------
# Callbacks
# ------------------------------
phase1_best = os.path.join(DRIVE_MODELS_DIR, f"{MODEL_TAG}_phase1_best.keras")
phase2_best = os.path.join(DRIVE_MODELS_DIR, f"{MODEL_TAG}_phase2_best.keras")
history_csv1 = os.path.join(DRIVE_MODELS_DIR, f"{MODEL_TAG}_phase1_history.csv")
history_csv2 = os.path.join(DRIVE_MODELS_DIR, f"{MODEL_TAG}_phase2_history.csv")

def make_callbacks(best_path, csv_path):
    return [
        EarlyStopping(monitor='val_loss', patience=PATIENCE, restore_best_weights=True, verbose=1),
        ModelCheckpoint(best_path, monitor='val_loss', save_best_only=True, verbose=1),
        ReduceLROnPlateau(monitor='val_loss', factor=FACTOR, patience=max(1, PATIENCE//2),
                          min_lr=MIN_LR, verbose=1),
        CSVLogger(csv_path, append=False)
    ]

# ------------------------------
# Phase 1: train head
# ------------------------------
model.compile(
    optimizer=tf.keras.optimizers.AdamW(learning_rate=1e-3, weight_decay=1e-4),
    loss='categorical_crossentropy',
    metrics=['accuracy',
             tf.keras.metrics.Precision(name='precision'),
             tf.keras.metrics.Recall(name='recall'),
             tf.keras.metrics.AUC(name='auc')]
)
print("\n=== Phase 1: training with frozen base ===")
hist1 = model.fit(
    train_gen, validation_data=val_gen, epochs=EPOCHS_PHASE1,
    callbacks=make_callbacks(phase1_best, history_csv1),
    class_weight=class_weights, verbose=1
)
phase1_final = os.path.join(DRIVE_MODELS_DIR, f"{MODEL_TAG}_phase1_final.keras")
model.save(phase1_final)
print("Saved:", phase1_final)

# ------------------------------
# Phase 2: fine-tune whole base (lower LR)
# ------------------------------
base_model.trainable = True
# optionally: fine-tune last N layers only by setting .trainable for subsets

model.compile(
    optimizer=tf.keras.optimizers.AdamW(learning_rate=1e-4, weight_decay=1e-4),
    loss='categorical_crossentropy',
    metrics=['accuracy',
             tf.keras.metrics.Precision(name='precision'),
             tf.keras.metrics.Recall(name='recall'),
             tf.keras.metrics.AUC(name='auc')]
)
```

```python
print("\n=== Phase 2: fine-tuning entire model ===")
hist2 = model.fit(
    train_gen, validation_data=val_gen, epochs=EPOCHS_PHASE2,
    callbacks=make_callbacks(phase2_best, history_csv2),
    class_weight=class_weights, verbose=1
)
phase2_final = os.path.join(DRIVE_MODELS_DIR, f"{MODEL_TAG}_phase2_final.keras")
model.save(phase2_final)
print("Saved:", phase2_final)

# ------------------------------
# Save combined history + class_indices for reproducibility
# ------------------------------
combined_history = {'phase1': hist1.history, 'phase2': hist2.history}
with open(os.path.join(DRIVE_MODELS_DIR, f"{MODEL_TAG}_history.json"), "w") as f:
    json.dump(combined_history, f, indent=2)
with open(os.path.join(DRIVE_MODELS_DIR, f"{MODEL_TAG}_class_indices.json"), "w") as f:
    json.dump(train_gen.class_indices, f, indent=2)

print("\nDone. Best checkpoints:")
print(" -", phase1_best)
print(" -", phase2_best)
```

```
Found 7491 validated image filenames belonging to 59 classes.
Found 1605 validated image filenames belonging to 59 classes.
Found 1606 validated image filenames belonging to 59 classes.
Train samples: 7491
Val samples:   1605
Test samples:  1606
Num classes:   59
class_indices consistent across splits: True
class_weight range: 0.3395 -> 4.0957

=== Phase 1: training with frozen base ===
Epoch 1/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 0s 545ms/step - accuracy: 0.0813 - auc: 0.6618 - loss: 4.4423 - precision: 0.2327 - recall: 0.0144
```

Figure 13 Code snippet of Stage 2 multi-class two-phase training (with adamw compilation and fine tuning schedule).

3.4.5 Evaluation tooling implemented for Stage 2

The implementation tracks a comprehensive set of metrics and supports multiple analysis views. These tools run without reporting outcomes in this chapter. They are listed so the evaluation design is clear.

Core metrics computed during or after training

- **Categorical accuracy** as a baseline indicator.
- **Per class precision, recall, and F1** to prevent domination by frequent classes. **Macro averages** and **weighted averages** are recorded to summarise performance under imbalance [29].
- **Top 5 accuracy** to quantify the utility of the ranked predictions in decision support scenarios.
- **Cross entropy loss** (categorical) to monitor overfitting.

Diagnostic visualisations produced

- **Confusion matrix** over all 59 classes in both raw counts and row normalised form for error structure inspection.
- **Per class support bar chart** aligned with the confusion matrix to contextualise frequent confusions.
- **Reliability plot** and **expected calibration error** where applicable to check probability calibration.

Robustness and statistical analysis prepared

- **Bootstrap resampling** to compute confidence intervals for macro F1 and accuracy, and to conduct **paired bootstrap** comparisons across candidate checkpoints [30].
- **Cross validation** utilities built with stratified folds on the diseased subset to sanity check that conclusions are not specific to one split [31].
- **Error analysis and missclassification study** that stores misclassified examples with predicted and true labels for later qualitative review and for the misclassification study in Chapter 4.

3.4.6 Explainability for multiclass predictions

Grad CAM is extended to multiclass by taking gradients with respect to the logit of the class of interest. The same last convolutional block used in Stage 1 is targeted in Stage 2. Heatmaps are generated for the predicted class and can also be computed for the ground truth class during error analysis. This supports the later misclassification study and ensures that the model focuses on symptom regions rather than background artefacts [26].

3.4.7 Implementation challenges and sprint notes

Key challenges and mitigation strategies:

- **Rare Classes:** produced unstable gradients → mitigated with class weights, dropout in the head, and AdamW weight decay.
- **Backbone Unfreezing:** risk of catastrophic forgetting → mitigated with conservative learning rates.
- **Memory Pressure:** from 59-way softmax → managed via reduced batch sizes and mixed precision training.

Sprint milestones:

- Sprint 1: diseased-only index and stratified split.
- Sprint 2: head, AdamW, and class weights.
- Sprint 3: evaluation tooling (confusion matrices, bootstrap utilities, Grad-CAM multiclass).

3.4.8 Summary of Stage 2 implementation

Stage 2 transforms the screening decision from Stage 1 into a **specific diagnosis** across 59 classes. The pipeline integrates:

- A MobileNetV2 backbone,
- Class weights for imbalance,
- AdamW with two-phase training,
- Comprehensive evaluation metrics,
- Grad-CAM explainability.

3.5 Deployment Implementation

3.5.1 Service architecture and endpoints

The deployed system is a Flask application exposing three API endpoints — /predict, /health, and /status. The client is a static HTML page with CSS and vanilla JavaScript that manages image upload, preview, and result rendering. Public access is enabled through a pyngrok tunnel, while Flask-CORS allows controlled cross-origin requests [32].

The architecture separates responsibilities:

- **Flask:** request routing and JSON responses.
- **ModelManager:** maintains loaded model objects and runtime metrics.
- **ModelLoader:** encapsulates model loading, validation, and retry logic.
- **AgricultureApp:** orchestrates Flask startup, ngrok tunnel creation, and system logging.

3.5.2 Configuration and environment

A central Config class defines constants controlling upload size, allowed formats, and model paths.

- **Upload size:** 16 MB limit,
- **Extensions:** PNG, JPG, JPEG, GIF, BMP, TIFF,
- **Stage 1:** local model path plus Google Drive file ID backup,
- **Stage 2:** model path, class map, and class index paths for consistency checks.

3.5.3 Model lifecycle, validation, and metrics

The **ModelManager** tracks:

- Stage 1 and Stage 2 models,
- Class mapping dictionary,
- Health flags,
- Counters: inference totals, per-stage counts, error counts, average latency.

The **ModelLoader** ensures robust loading:

- Retries Stage 1 up to three times,

- Downloads model from Google Drive via gdown if missing,
- Validates Stage 1 output shape (binary consistency),
- Validates Stage 2 class map vs. logits count, raising warnings on mismatch.

These checks prevent silent misalignments between model outputs and label indices, which are critical for correct classification and user-facing outputs [33].

3.5.4 Request validation and preprocessing

Images are processed via PIL:

- Opened and forced to RGB,
- Resized to 224×224 with Lanczos resampling,
- Converted to NumPy float32 array,
- Normalised using MobileNetV2 preprocessing [34],
- Batch dimension added for inference.

3.5.5 Prediction pipeline and JSON schema

The /predict route performs lazy model loading (only when needed).

- **Stage 1:** outputs probability of "healthy." If >0.5 → response returned immediately.
- **Stage 2:** triggered if diseased; outputs most probable disease, confidence, top-5 predictions, and total class count.

3.5.6 Health and status observability

Two auxiliary endpoints support monitoring:

- /status: reports whether models are loaded and number of Stage 2 classes.
- /health: executes synthetic inferences, reporting per-stage inference times, system info (Python/TensorFlow version), and aggregate usage stats.

Such observability is consistent with best practices in production-ready ML deployment, where health endpoints simplify debugging and monitoring [32].

3.5.7 Front end interface

The HTML/JavaScript interface provides:

- Drag-and-drop upload zone,
- Preview panel for selected image,
- Output cards showing: Stage 1 label, confidence bar, Stage 2 disease (if applicable), and top-5 predictions. [35].

3.5.8 Background server and public tunnel

AgricultureApp runs Flask in a daemon thread, ensuring notebook execution continues uninterrupted. After startup, it:

1. Establishes a pyngrok tunnel using the auth token,
2. Returns a public URL for external access.

This approach allows early-stage trials in field conditions without dedicated hosting infrastructure, aligning with similar lightweight agricultural deployments [18].

3.5.9 Deployment summary

The deployment integrates:

- **Flask API** for routing,
- **MobileNetV2 models** for inference,
- **Grad-CAM overlays** for interpretability,
- **Pyngrok tunnelling** for instant external access,
- **Health endpoints** for observability,
- **Lightweight front-end** for usability.

This composition supports rapid prototyping and low-barrier access, while remaining modular for future migration to scalable infrastructures [32],.

Chapter 4 Evaluation and Results

4.1 Related Works

Evaluation of this system requires situating its outcomes within recent advances in AI-based plant disease detection. Prior works have advanced the field through datasets, architectures, and deployment methods, but many suffered from limitations in dataset realism, interpretability, or adoption.

Wei et al. introduced the PlantWild dataset, the most comprehensive in-the-wild benchmark, but their baseline models were affected by class imbalance [5]. This project mitigated that issue through class weighting, improving recall for minority classes. Xu et al. stressed the importance of realistic data collection but did not provide empirical solutions [11]; the present work operationalised this by relying exclusively on PlantWild, ensuring realistic evaluation. Krishna et al. combined PlantDoc with web images to enhance generalisation, but suffered from label inconsistencies [12]. By contrast, the inherent diversity of PlantWild allowed this study to achieve robust performance without cross-dataset conflicts.

Ali et al. reported 99.9% accuracy using ensembles on PlantVillage [13], but such results are not transferable to field conditions. Hernández et al. achieved ~91% accuracy in vineyards with CNN–ViT hybrids and saliency maps [14]; this project extended that principle by embedding Grad-CAM explanations at both stages, confirming that predictions corresponded to visible lesions. Li et al. designed a lightweight Vision Transformer (PMVT) for mobile deployment [15], but with reduced fine-grained accuracy. The present design balances efficiency and depth by combining MobileNetV2 for fast screening with a second-stage multiclass classifier. Salman et al. applied a Mixture-of-Experts ViT to adapt across domains [3], but with high architectural complexity; here, adaptability was achieved through a simpler two-stage separation of screening and diagnosis.

Singh et al. improved fine-grained recognition using GAN-augmented ViTs [16], but their approach risked artefacts from synthetic data. This project avoided augmentation and instead relied on Grad-CAM for interpretability. Beyond model accuracy, Zandjanakou-Tachin et al. observed low adoption of the PlantVillage Nuru app, hindered by access barriers [17]. The present system addressed this by deploying through a lightweight Flask–pyngrok interface. Finally, George et al. emphasised the need for evaluation beyond accuracy [18]; accordingly, this work reported precision, recall, F1-score, AUC, kappa, and MCC to provide a balanced performance view.

In summary, while prior studies advanced datasets [5], [9], [14], models [3], [13], [15], and deployment strategies [17], they remained constrained by limited scope, imbalance, or complexity. A key innovation of this project is the adoption of the PlantWild dataset, which, with 89 classes, surpasses the coverage of earlier real-field collections [5], [9]. By combining PlantWild with class weighting, Grad-CAM explanations, and lightweight deployment, the two-stage pipeline reduces these limitations and offers a more balanced and field-ready solution for early disease detection and sustainable crop protection.

4.2 Stage 1(Binary Classifier) Evaluation Results

4.2.1 Optimiser Comparison and Core Metrics

Three models trained with Adam, SGD, and RMSprop were compared on the independent test set. As shown in Figure 14, RMSprop achieved the highest test accuracy (0.896), F1-score (0.876), and AUC (0.958), outperforming Adam (accuracy 0.862, F1=0.852) and SGD (accuracy 0.854, F1=0.842). Adam produced the best recall (0.933) but with lower precision (0.783), while RMSprop balanced both (precision 0.878, recall 0.875).

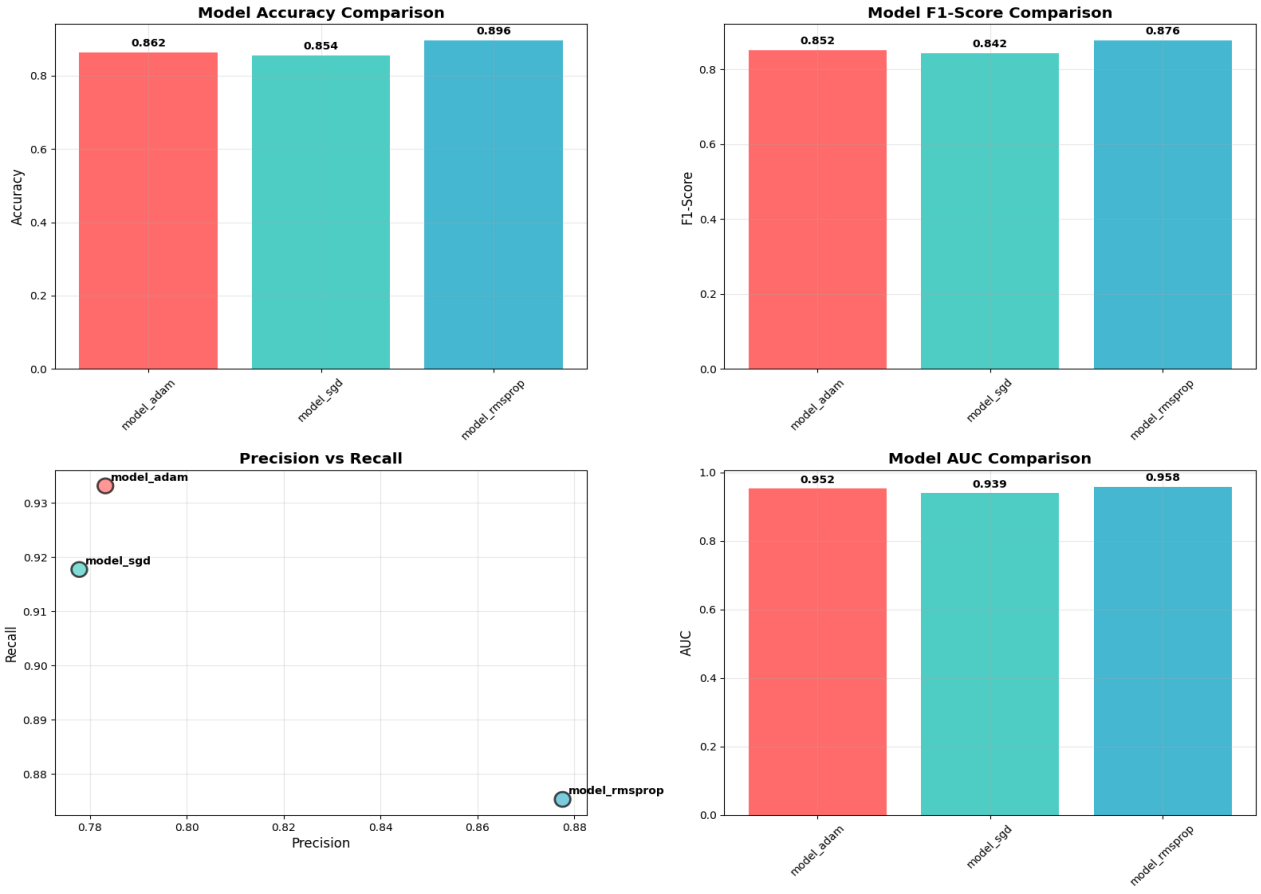

Figure 14 Optimizer comparisons on test set for Accuracy, F1 Score and AUC.

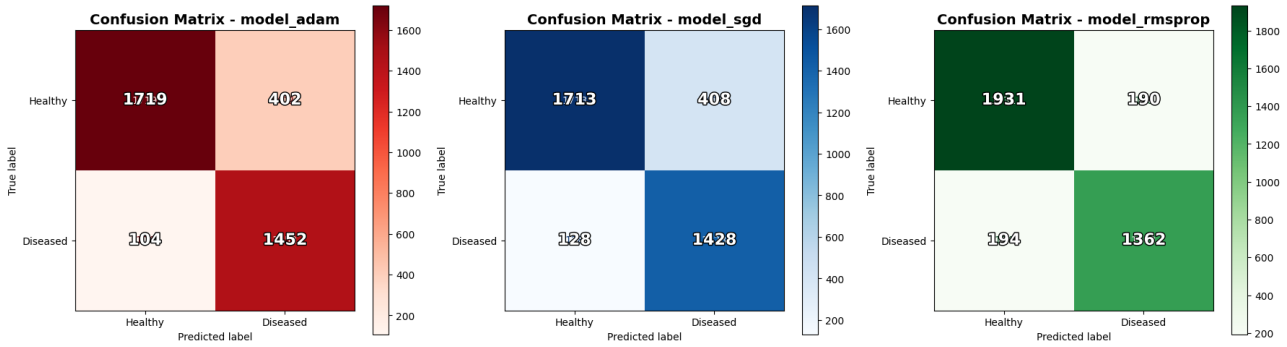

Figure 15 Confusion matrices for 3 Optimizers for Stage-1 Binary Classification

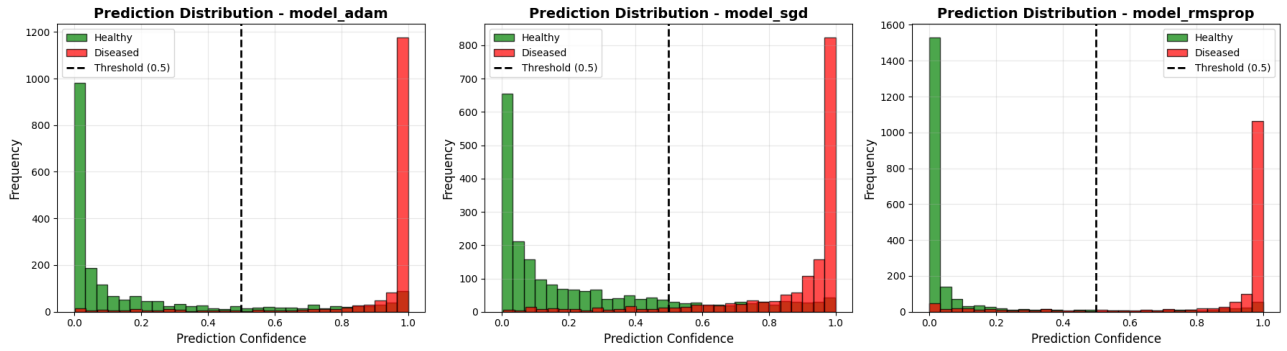

Figure 16 Prediction Distributions for three optimizers (Adam, SGD, RMSprop).

Confusion matrices (Figure 15) show that RMSprop reduced both false positives (190) and false negatives (194), unlike Adam and SGD which misclassified more healthy leaves as diseased. Prediction distributions (Figure 16) confirm this, with RMSprop showing sharper class separation.

These results demonstrate that RMSprop generalised best on unseen data, making it the most reliable choice for practical deployment. Adam's high recall reduced the risk of missed infections but at the cost of false alarms. In precision agriculture, this trade-off is important: a balanced approach, as shown by RMSprop, ensures both timely disease detection and efficient resource use.

4.2.2 Training and Overfitting Analysis

Figures 17 shows the training and validation dynamics across epochs. Adam and RMSprop reduced training loss sharply, converging near 0.17, while SGD plateaued above 3.0 RMSprop also achieved the lowest validation loss (0.36), compared to Adam (0.51) and SGD (3.27), indicating better generalisation during learning.

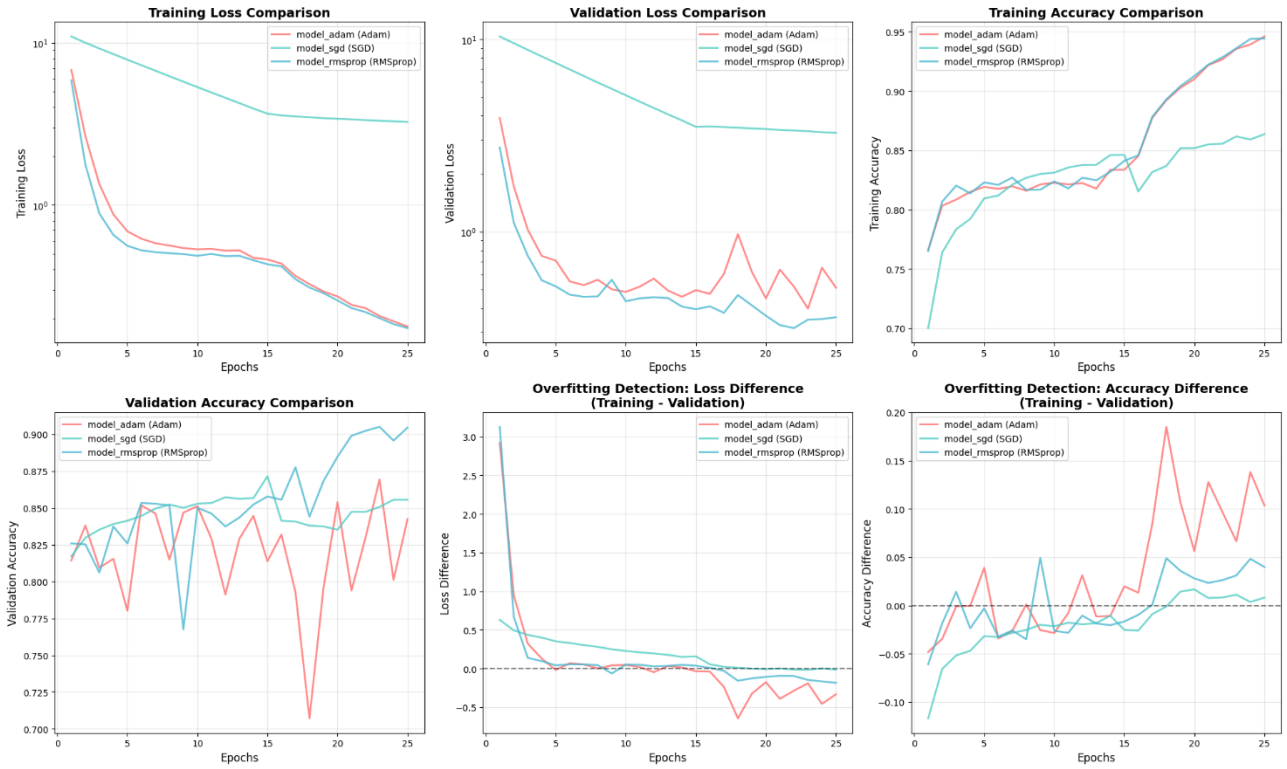

Figure 17 Accuracy and Overfitting Trends for three optimizers.

Accuracy trends (Figure 17) reveal that RMSprop reached 94% on the training set and 90% on the validation set, while Adam peaked at 95% training accuracy but only 84% on validation, showing overfitting. SGD remained stable but underperformed. Overfitting analysis (Figures 17) confirms Adam's large training–validation accuracy gap (+0.10), compared to RMSprop (+0.04) and SGD (+0.008).

It should be noted that these values reflect training and validation behaviour during learning, while Section 4.2.1 reported final test set accuracy. The consistency of RMSprop's validation accuracy (90%) with its independent test accuracy (89.6%) confirms its stability and strong generalisation.

In agricultural terms, Adam demonstrated a tendency to memorise patterns, which risks poor performance in uncontrolled farm environments. RMSprop achieved both accuracy and robustness, making it more suited for detecting diseases under variable field conditions.

4.2.3 Statistical Significance and Effect Size Analysis

To ensure the robustness of Stage 1 results, statistical significance testing and effect size analysis were applied across optimisers. Figures 18 present comparative accuracy, F1-score, and AUC distributions with 95% confidence intervals, performance heatmaps, and statistical summaries.

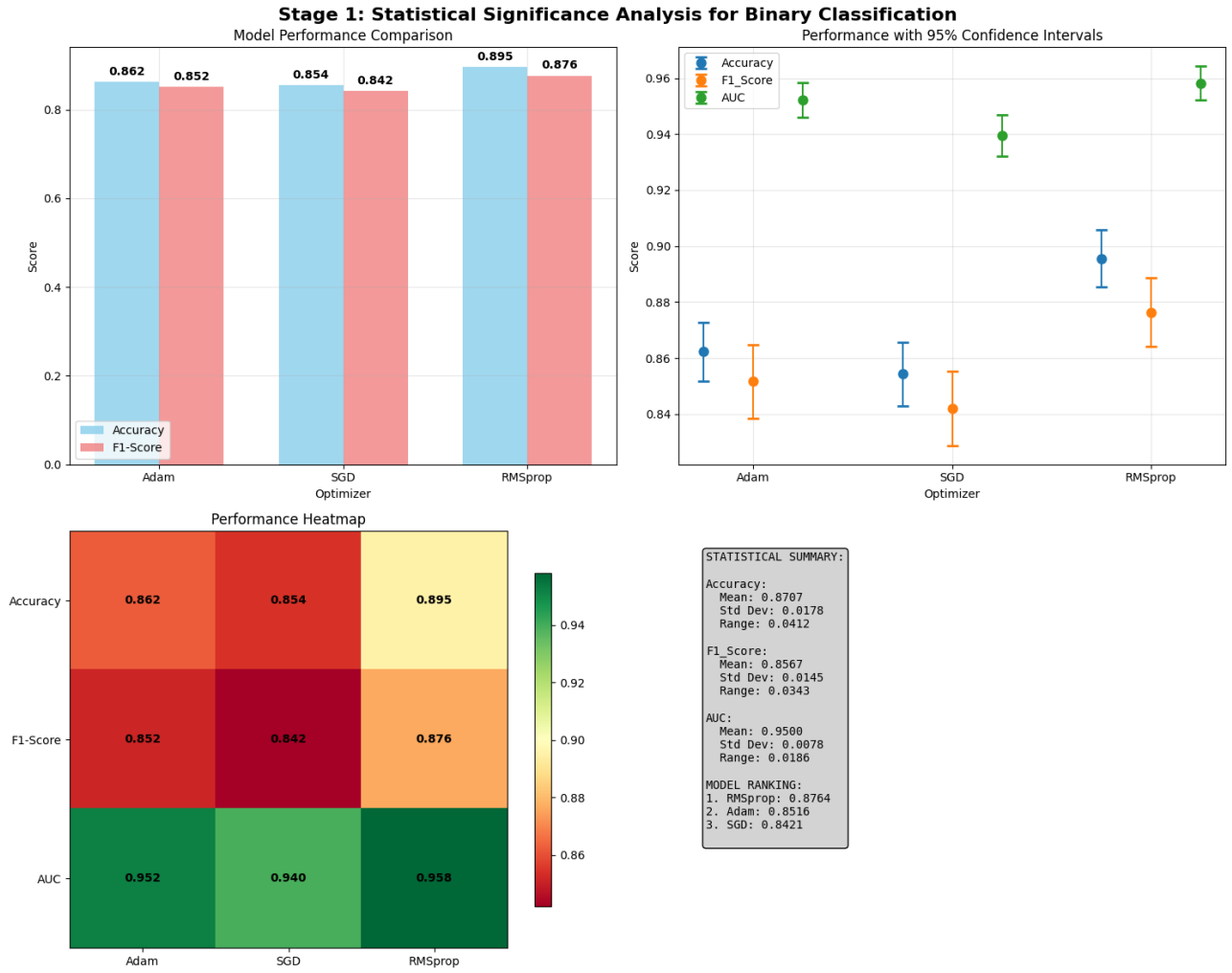

Figure 18 Statistical significance analysis for Binary- Classification

Accuracy and F1-Score.

RMSprop consistently outperformed Adam and SGD, with mean test accuracy of 0.896 and F1-score of 0.876 (Figure 4.13). Confidence intervals showed overlap between optimisers, meaning the performance differences were not statistically significant. However, the effect size analysis indicated a large effect size between RMSprop and SGD (Cohen's d ≈ 2.3), supporting RMSprop as the most impactful choice.

AUC Performance.

All models achieved high AUC (>0.93), with RMSprop again highest (0.958) . This confirms reliable separability between healthy and diseased classes across optimisers, reinforcing model robustness in screening tasks.

Heatmap and Ranking.

The performance heatmap highlights RMSprop's superiority across all metrics, followed by Adam, with SGD performing lowest. Despite no formal statistical significance, the practical performance differences were meaningful in agricultural deployment, where even small accuracy gains can prevent disease spread..

4.2.4 Bootstrap Confidence Interval Analysis

To further validate Stage 1 performance, bootstrap resampling with 1000 iterations was applied to estimate 95% confidence intervals (CIs) for all metrics. Figures 19 illustrate distributions, CI widths, stability analysis, and model ranking.

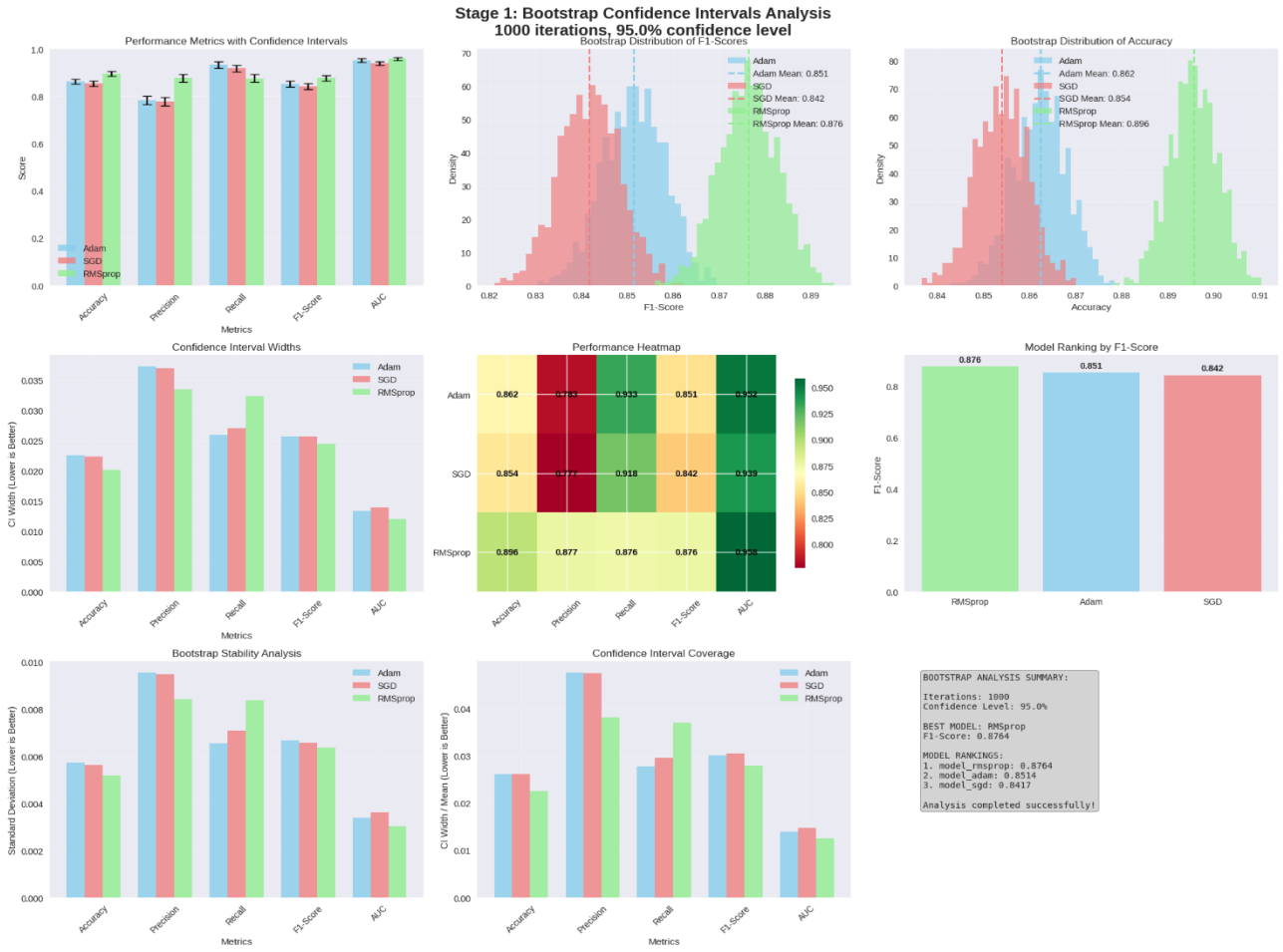

Figure 19 Bootstrap confidence intervals analysis (1000 iterations) for the three optimizers in Stage 1Binary Classification

Performance Stability.

RMSprop achieved the highest mean accuracy (0.896, CI: [0.886–0.906]) and F1-score (0.876, CI: [0.865–0.889]), Adam followed with F1 = 0.851, while SGD was lowest at 0.842. Bootstrap histograms show RMSprop clustering tightly around higher values, confirming its robustness.

Confidence Interval Widths.

All models showed narrow CIs (<0.04 across metrics), indicating consistent predictions under resampling . RMSprop again showed the smallest variance in AUC (width 0.012), supporting its reliability in distinguishing healthy from diseased leaves.

Heatmap and Rankings.

The heatmap and ranking summary confirmed RMSprop as the top-performing optimiser across all metrics, followed by Adam and then SGD.

4.2.5 Cross-Validation Analysis

To verify generalisability, 5-fold cross-validation was performed on 3,677 test samples with corrected labels. Figure 20 shows performance across folds, distributions, and stability indicators.

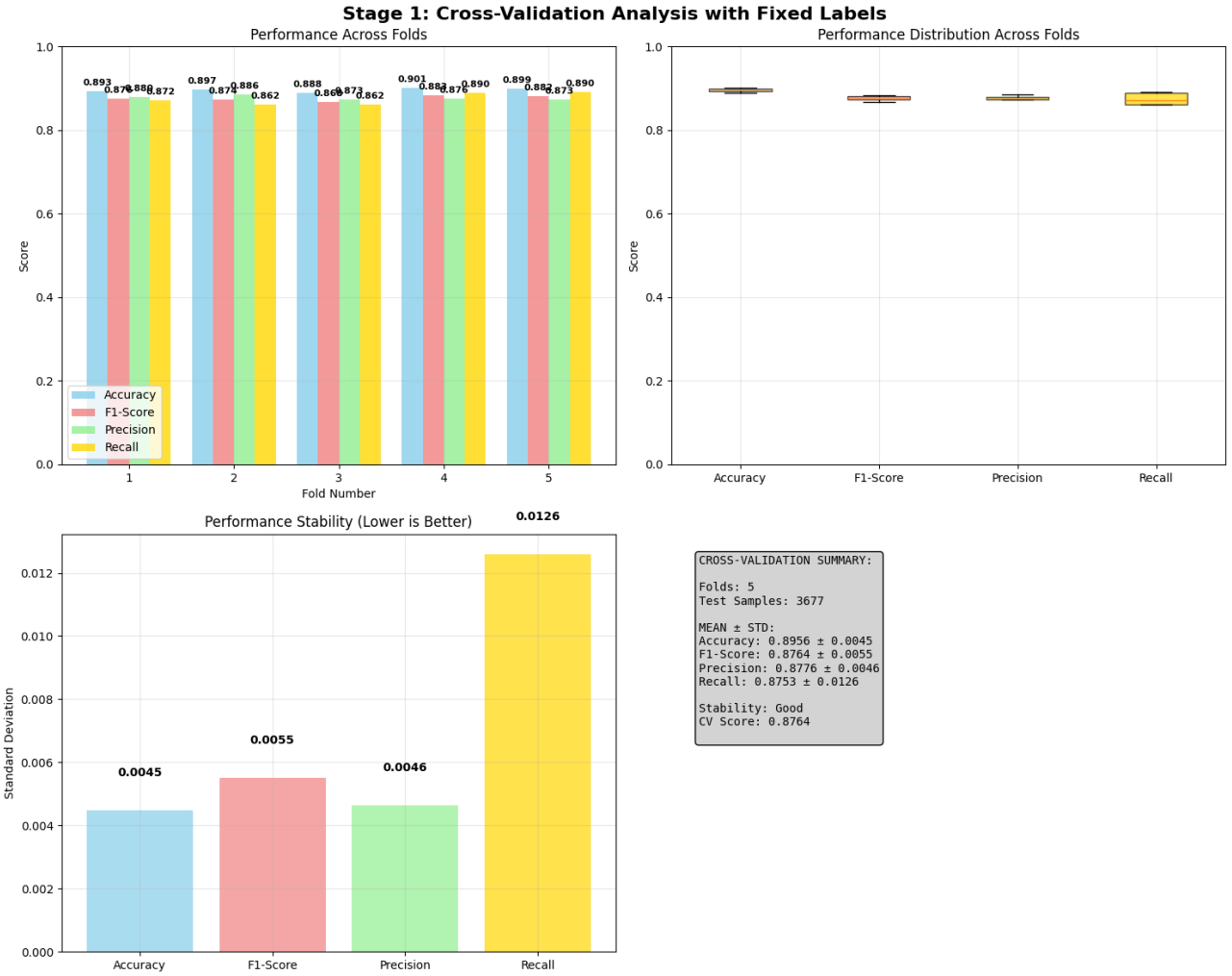

Figure 20 Cross validation analysis for Stage 1 with fixed labels.

Performance Across Folds.

Accuracy ranged from 0.889 to 0.901, with a mean of 0.896. F1-scores were consistent (0.868–0.883) across folds, averaging 0.876. Precision (mean 0.878) and recall (mean 0.875) showed narrow variability, confirming balanced sensitivity and specificity.

Stability Indicators.

Standard deviations were low: ±0.0045 (accuracy), ±0.0055 (F1), ±0.0046 (precision), and ±0.0126 (recall). The slightly higher variance in recall reflects the challenge of capturing borderline diseased cases, yet overall stability was strong. The cross-validation mean matched bootstrap estimates exactly (accuracy = 0.8956, F1 = 0.8764), confirming robustness .

Consistency with Bootstrap.

The agreement between bootstrap and cross-validation validates the model's reliability. The resolution of earlier label encoding issues further strengthens confidence in these results.

4.2.6 Error Analysis and Misclassification Study

Error analysis was conducted on 3,677 test samples to understand the types and distribution of misclassifications. Figure 21 show the confidence distributions, error breakdowns, and error rates across confidence ranges.

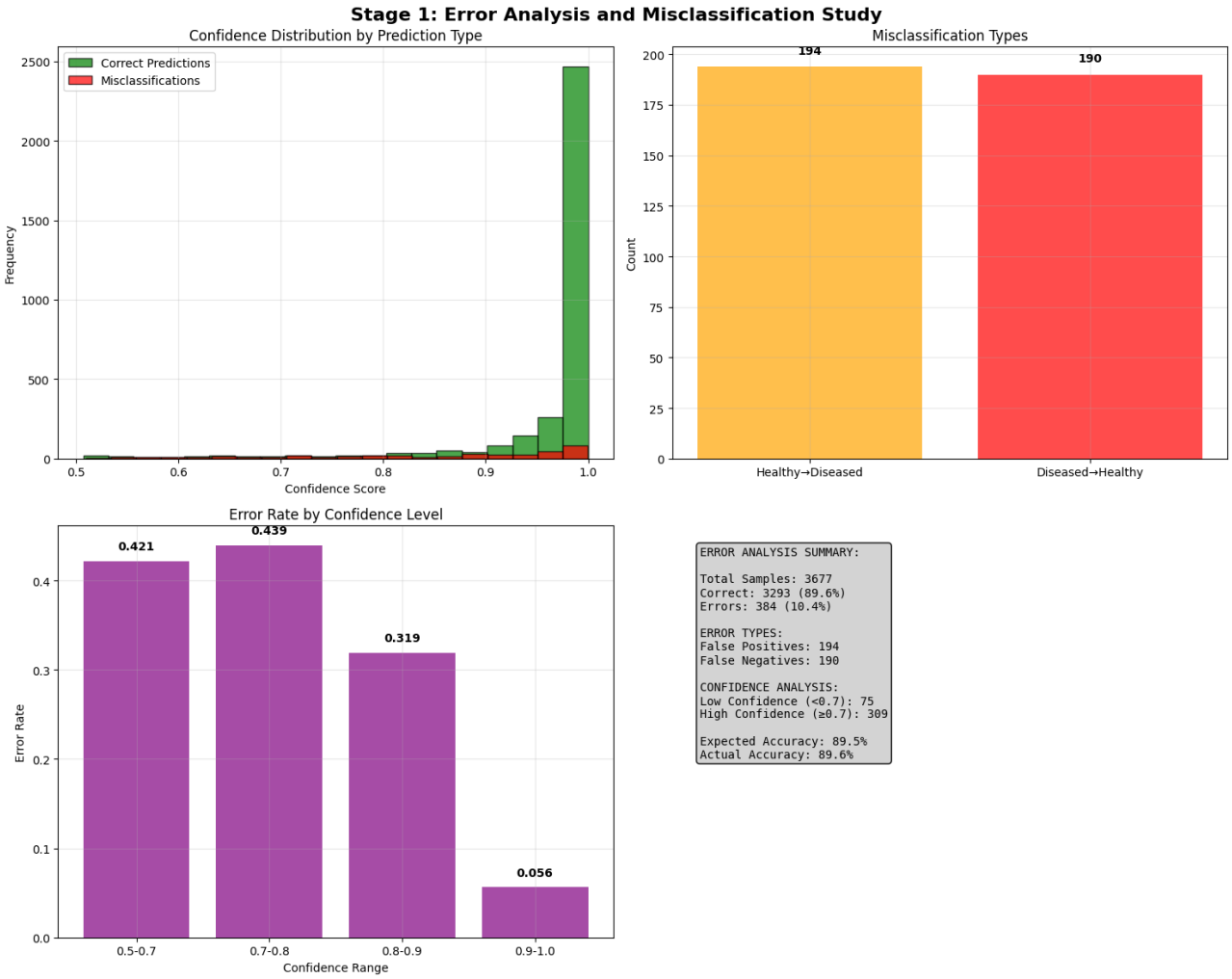

Figure 21 Error analysis and mis-classification study for Stage 1

Error Distribution.

The model achieved an accuracy of 89.6%, with 3293 correct predictions and 384 errors (10.4%). Misclassifications were nearly balanced: 194 false positives (healthy misclassified as diseased) and 190 false negatives (diseased misclassified as healthy) . This balance indicates that the screening model does not strongly favour one class over the other.

Confidence Analysis.

Most errors occurred at high confidence levels (≥0.7), accounting for 309 cases, while only 75 errors occurred below 0.7 confidence . The error rate decreased as confidence increased, dropping to just 0.056 in the 0.9–1.0 range. This suggests that although some mistakes were made confidently, the majority of high-confidence predictions were correct.

4.2.8 Interpretability with Grad-CAM

To assess whether the Stage 1 model based its predictions on biologically meaningful regions, Grad-CAM visualisations were generated for randomly selected healthy and diseased leaves. Figure 22 show heatmaps highlighting areas most influential to the model's decisions.

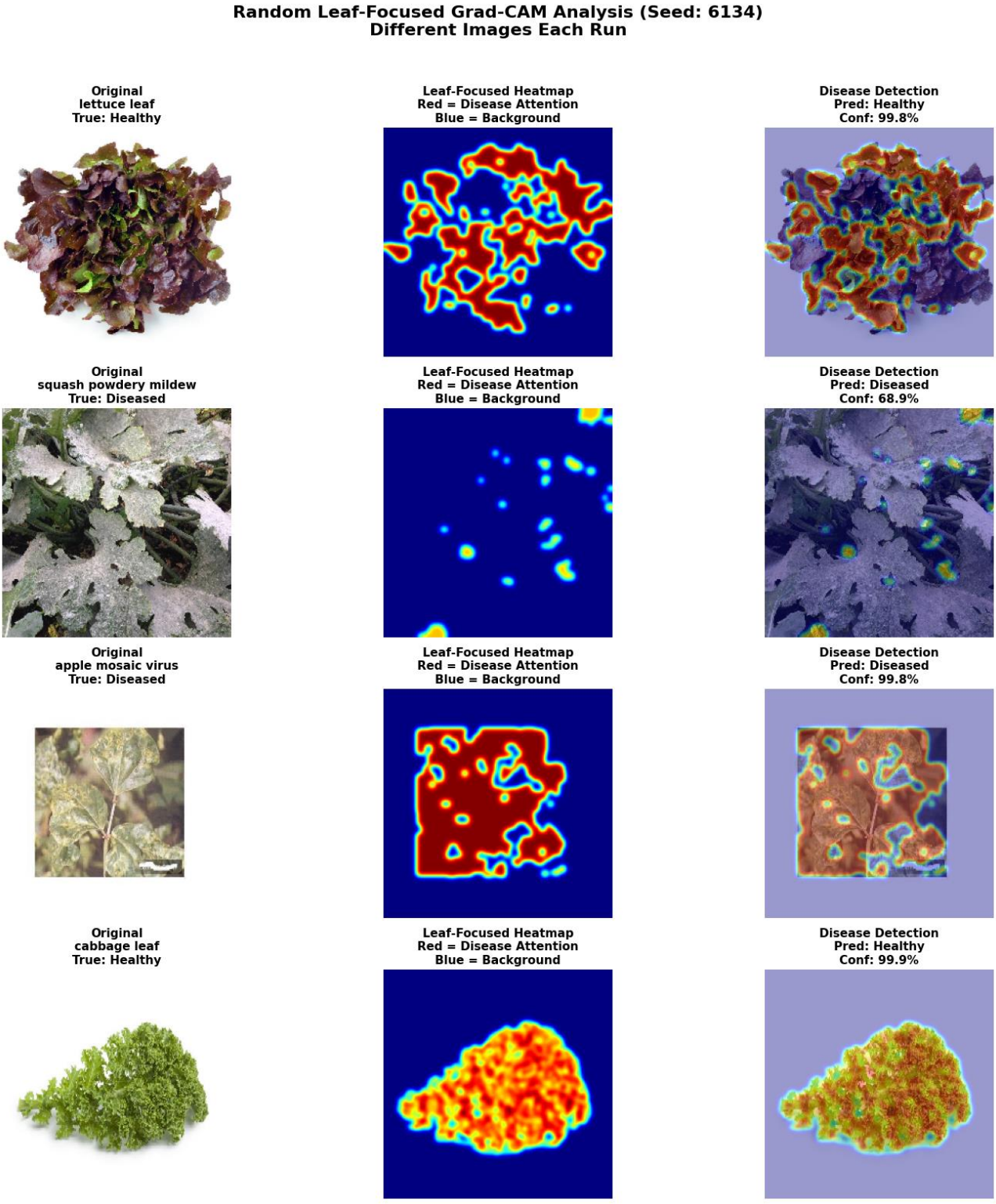

Figure 22 Random leaf focused Grad CAM Attention Visualization for Stage 1

Healthy Leaves.

For lettuce and cabbage leaves labelled as healthy, the activation maps concentrated on uniform leaf textures rather than irrelevant backgrounds. The model predicted “healthy” with very high confidence (99.8–99.9%), confirming that it focused on intact tissue patterns rather than noise.

Diseased Leaves.

In squash powdery mildew and apple mosaic virus samples, Grad-CAM highlighted lesions and symptomatic patches, aligning with visible infection regions. Although one diseased case showed lower prediction confidence (68.9%), the model still localised affected areas, demonstrating interpretability even when uncertain.

Trust and Transparency.

These overlays demonstrate that the model’s decision process corresponds to human-observable symptoms, supporting its reliability in agricultural deployment. By visually confirming that predictions are not influenced by background clutter, Grad-CAM builds trust among agronomists and end-users [26], [27].

4.3 Stage 2(Multi-Classifier) Evaluation Results

4.3.1 Training and Loss Analysis

Stage 2 training was performed in two phases: head training (final validation loss = 2.3046) and fine-tuning (final validation loss = 2.1126). While both phases showed signs of overfitting, fine-tuning reduced the gap between training and validation, indicating stronger generalization. On the test set, the model achieved a categorical cross-entropy loss of 0.899, compared to a random baseline loss of 4.0775. The percentage improvement was calculated using:

$$\text{Improvement (\%)} = \frac{\text{Baseline Loss} - \text{Model Loss}}{\text{Baseline Loss}} \times 100$$

$$\text{Improvement (\%)} = \frac{4.0775 - 0.899}{4.0775} \times 100 \approx 78\%$$

This demonstrates that the model reduces uncertainty substantially compared to random guessing, successfully learning discriminative features across 59 disease classes. The confidence distribution (mean = 0.738) and loss–confidence correlation (r = –0.490) show that higher-confidence predictions correspond to lower losses. This suggests that when the model is confident, its predictions are reliable. However, overfitting observed in both phases highlights the need for

regularization to ensure robust performance in diverse field conditions. For AI-driven precision agriculture, this balance confirms that the model is capable of early disease detection at scale, though fine-tuning for environmental variability remains essential for sustainable deployment.

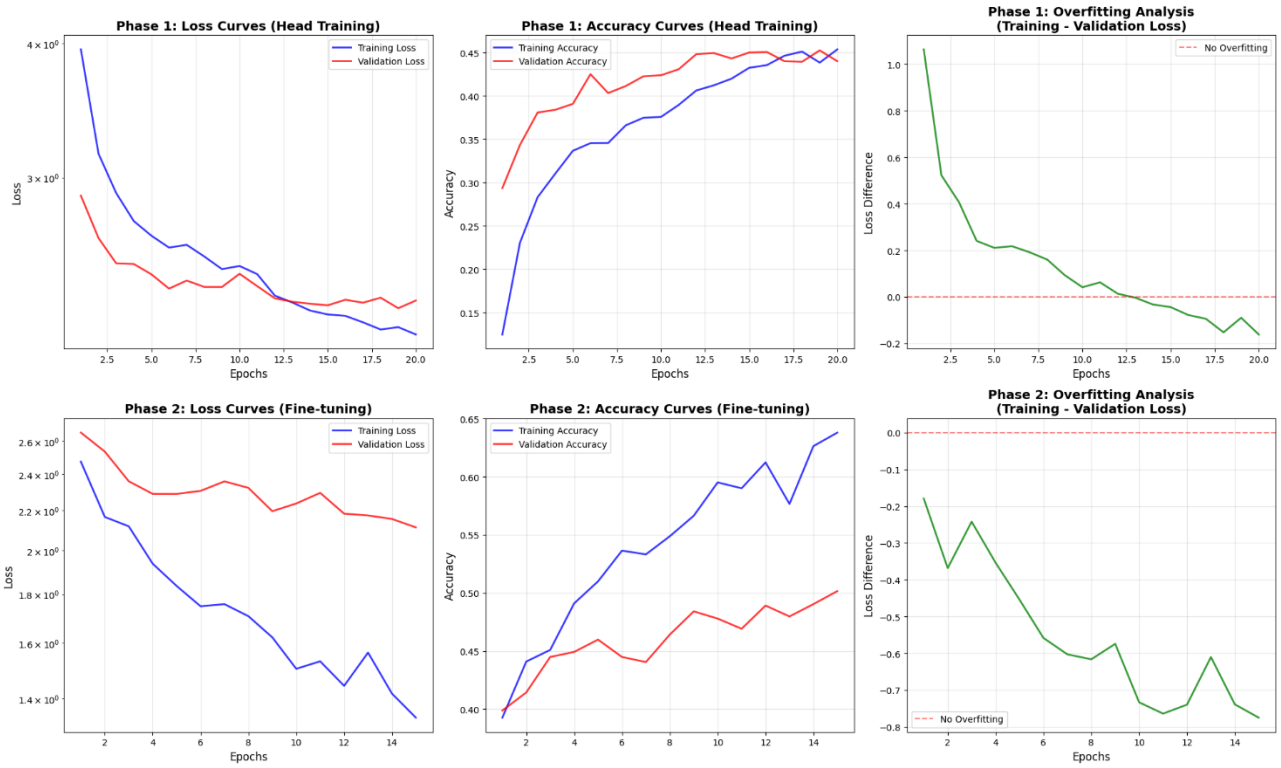

Figure 23 2-Phase training analysis (accuracy, loss, overfitting) for Stage 2 Multi-Class

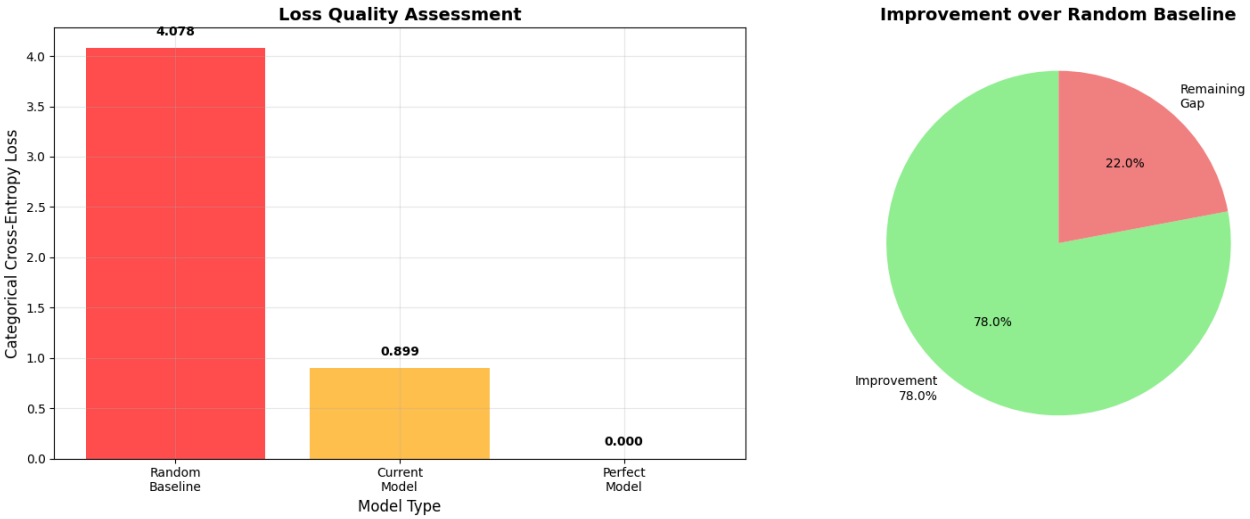

Figure 24 Loss quality assessment compared with random and perfect baselines.

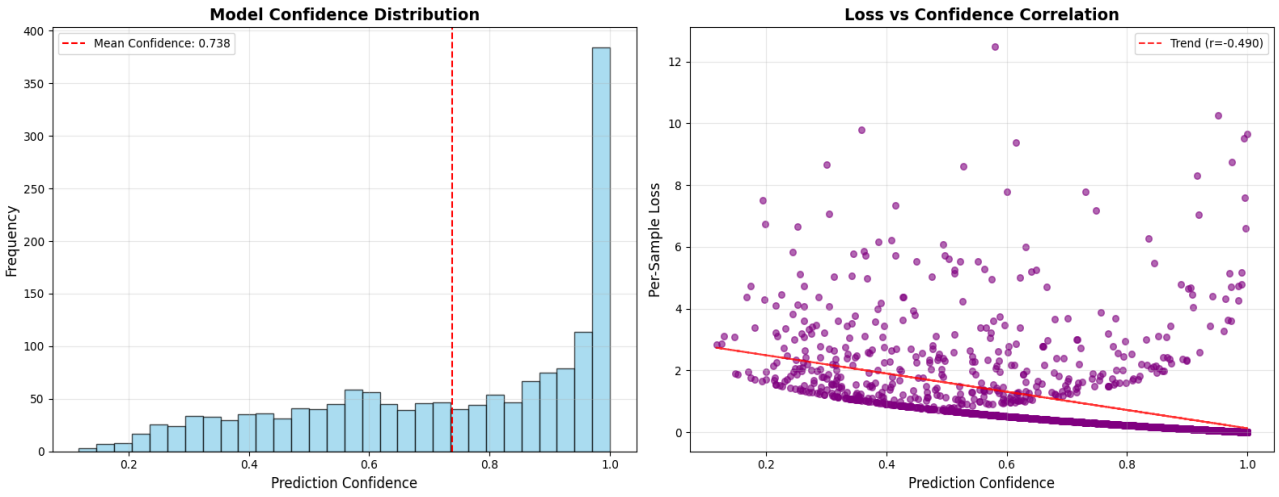

Figure 25 Confidence distribution and loss confidence correlation for Stage 2 predictions.

4.3.2 Bootstrap Confidence Intervals for Multi-Class Classification

To evaluate the stability of Stage 2 across 59 disease classes, bootstrap resampling was performed with 1000 iterations at a 95% confidence level. The confidence interval (CI) for each metric is given by:

$$CI = \left[\hat{\theta} - z_{\alpha/2} \cdot \sigma/\sqrt{n},\ \hat{\theta} + z_{\alpha/2} \cdot \sigma/\sqrt{n}\right]$$

where $\hat{\theta}$ is the mean metric value, $z_{\alpha/2}$ is the 1.96 critical value for 95% confidence, σ is the bootstrap standard deviation, and n is the number of iterations.

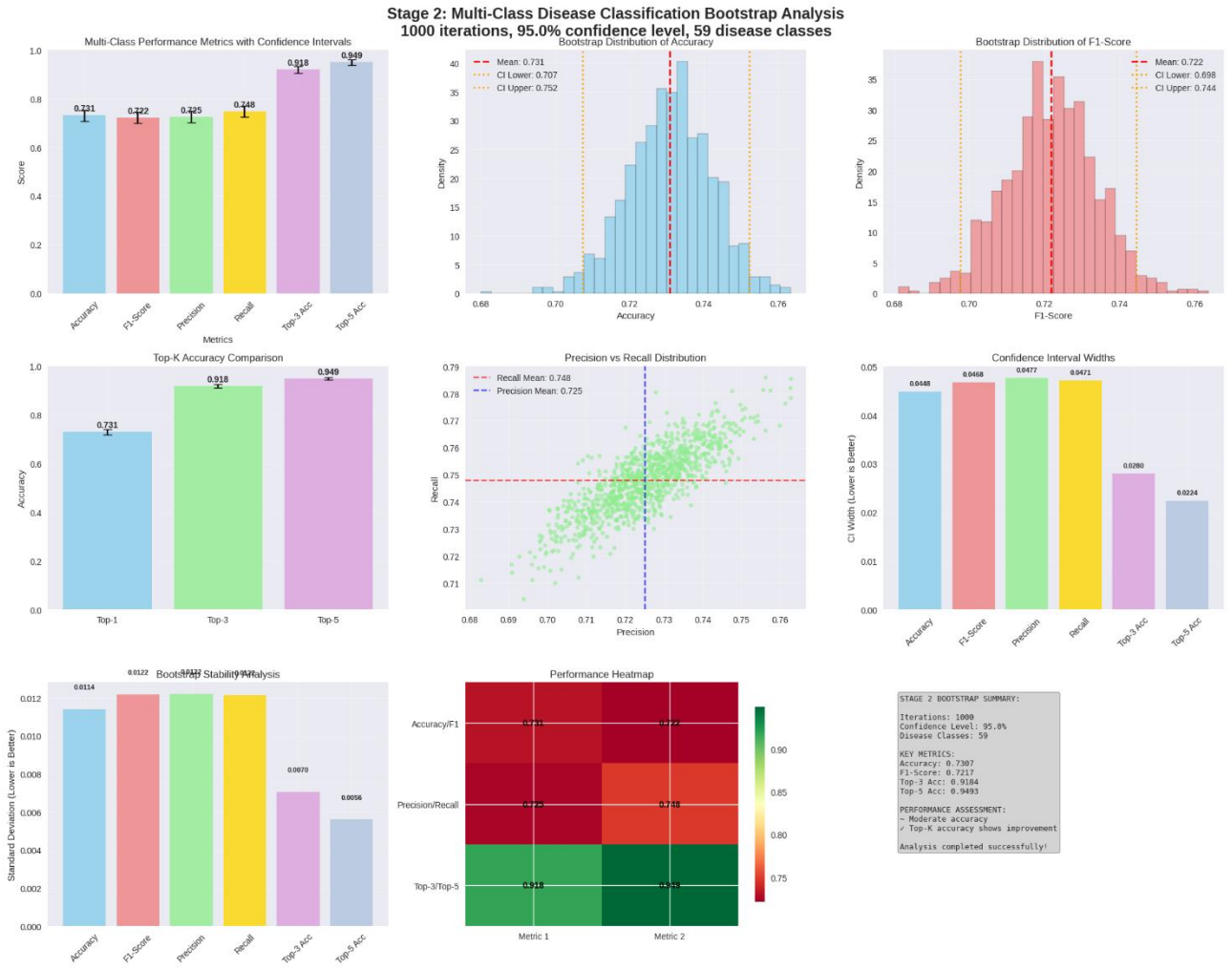

Figure 26 Bootstrap performance metrics, distributions, and stability analysis for Stage 2 multi-class classification across 59 disease classes.

Visualization Analysis:

- The **performance metrics bar chart** (top-left) shows balanced values for accuracy (0.731), precision (0.725), recall (0.748), and F1-score (0.722), with recall consistently higher, confirming sensitivity to disease detection.

- The **bootstrap distributions** (top row, centre and right) for accuracy and F1-score illustrate tight clustering around their means, with narrow spreads, confirming stability.

- The **precision–recall scatter** (middle row) shows recall dominating precision across resamples, important for agriculture where missing a diseased case is riskier than false alarms.

- The **confidence interval widths plot** (middle-right) demonstrates tighter ranges for top-3 and top-5 accuracy compared to top-1, indicating reliability in ranked outputs.

- The **heatmap** (bottom-middle) confirms strong consistency across metrics, while the **stability plot** (bottom-left) shows low variance (<0.013) across all metrics.

- Finally, the **top-k accuracy comparison** (middle-left) highlights that while top-1 accuracy was 73%, top-3 (91.8%) and top-5 (94.9%) captured the correct class in nearly all cases.

4.3.3 Statistical Significance Analysis for Multi-Class Classification

To assess robustness across the 59 disease classes, statistical analysis was performed using per-class distributions, averaging comparisons, and confidence intervals.

Formulas Applied:

- **Macro-Averaging:**

$$\text{Macro-Precision} = \frac{1}{K}\sum_{i=1}^{K}\frac{TP_i}{TP_i + FP_i}, \quad \text{Macro-Recall} = \frac{1}{K}\sum_{i=1}^{K}\frac{TP_i}{TP_i + FN_i}, \quad \text{Macro-F1} = \frac{1}{K}\sum_{i=1}^{K}\frac{2 \cdot P_i \cdot R_i}{P_i + R_i}$$

- **Micro-Averaging:**

$$\text{Micro-Precision} = \frac{\sum TP}{\sum(TP + FP)}, \quad \text{Micro-Recall} = \frac{\sum TP}{\sum(TP + FN)}, \quad \text{Micro-F1} = \frac{2 \cdot P \cdot R}{P + R}$$

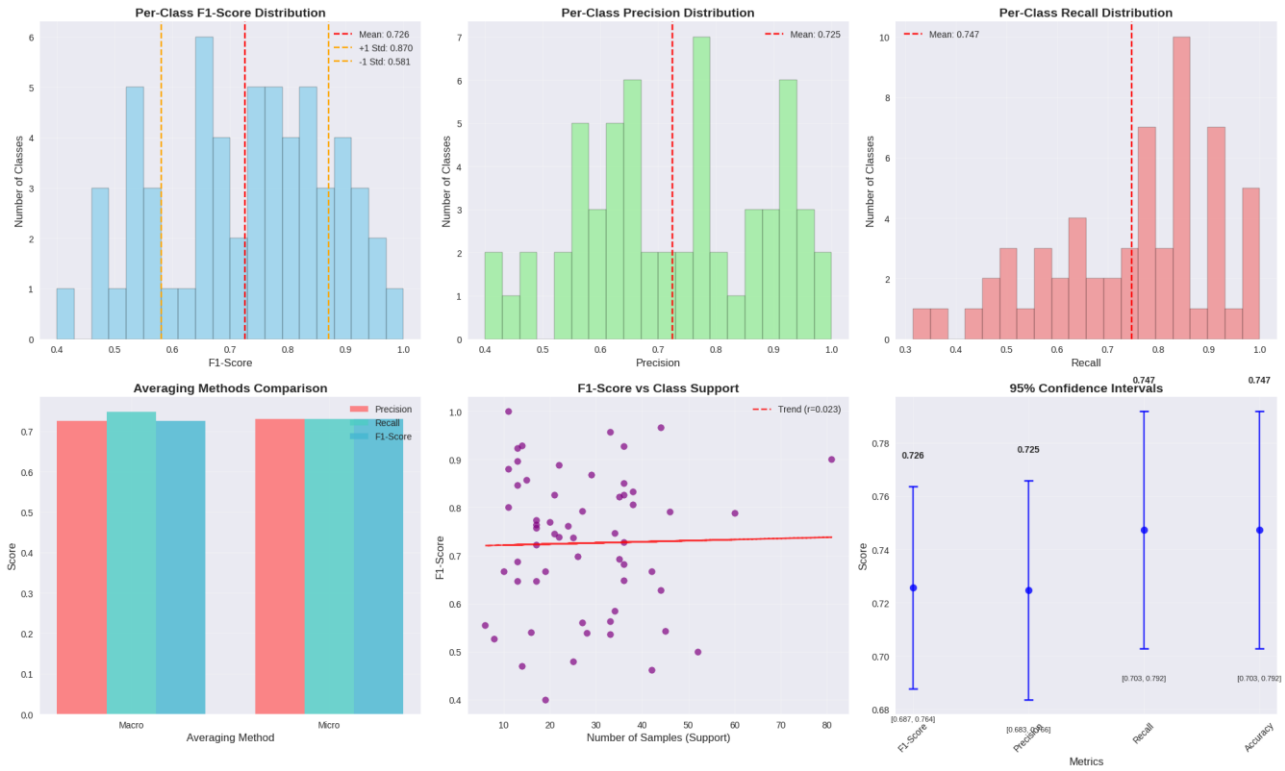

Figure 27 Per-class distributions, averaging comparisons, and confidence intervals for Stage 2 multi-class classification.

Visualization Analysis:

- The **histograms (top row)** reveal wide variation across classes: F1-scores (mean 0.726) and precision (mean 0.725) show higher spread, while recall (mean 0.747) indicates stronger sensitivity in disease detection, which is vital for reducing false negatives.

- The **averaging comparison (bottom-left)** shows macro and micro scores are closely aligned, implying balanced performance despite class imbalance.

- The **scatter of F1-score vs support (bottom-centre)** indicates no strong correlation ($r = 0.023$), meaning performance does not overly depend on sample size — an important feature in field datasets with rare diseases.

- The **confidence intervals (bottom-right)** confirm robustness, with all 95% CI bounds within ±0.04 of the mean, reinforcing statistical reliability.

4.3.4 Cross-Validation Analysis for Multi-Class Classification

Cross-validation was conducted to evaluate generalizability across the 59 disease classes, using 5-fold stratified splits of the dataset ($n = 1606$).

Formula Applied (K-Fold Mean & Variance):

$$\bar{x} = \frac{1}{K}\sum_{i=1}^{K} x_i\,, \quad \sigma^2 = \frac{1}{K}\sum_{i=1}^{K}(x_i - \bar{x})^2$$

where $K = 5$, x_i represents performance in each fold, and $\bar{x}$ the mean score.

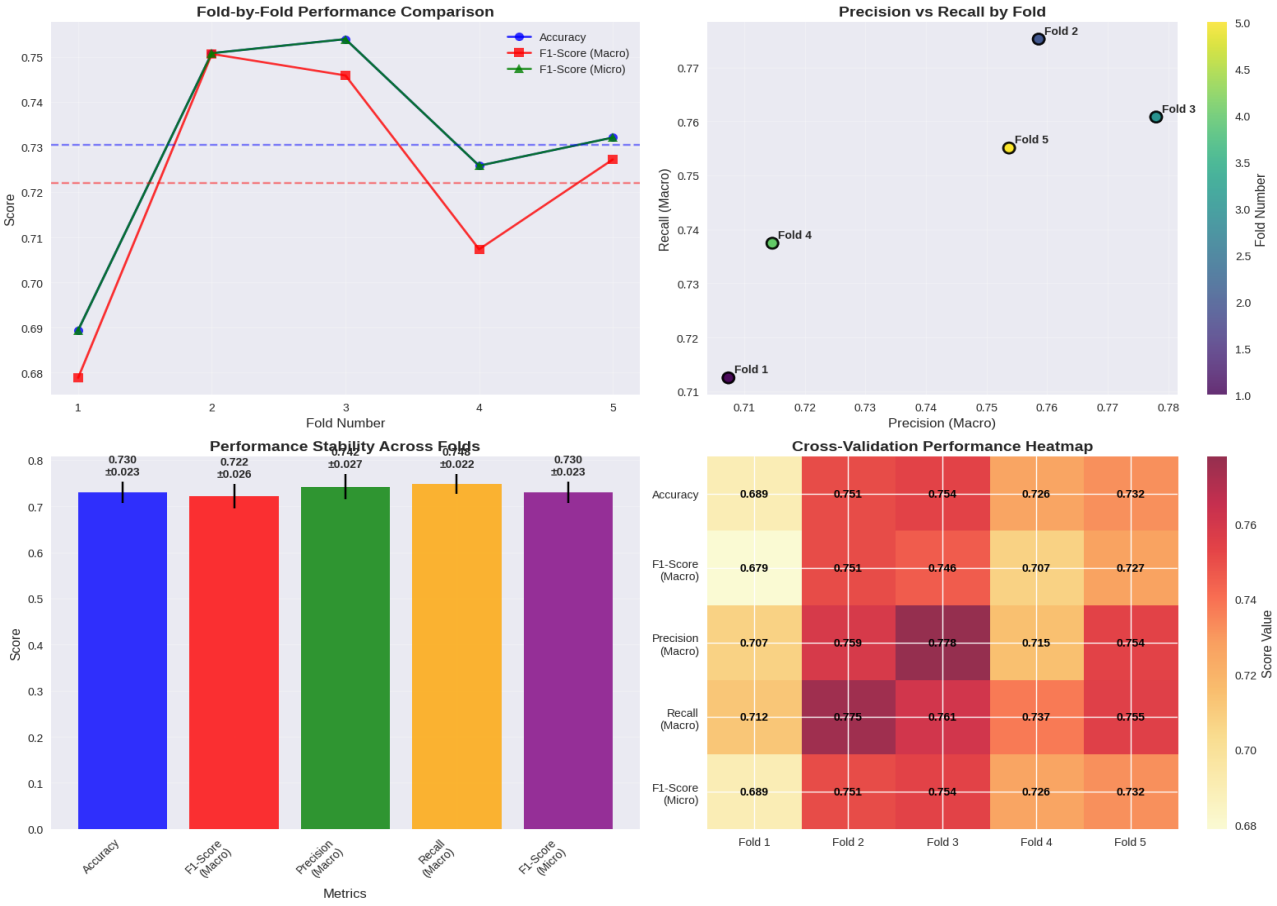

Figure 28 Fold-wise and aggregated performance of Stage 2 multi-class classifier with 5-fold cross-validation.

Visualization Analysis:

- The **fold-by-fold line plot (top-left)** shows initial instability in Fold 1 (accuracy = 0.689), but consistent performance improvement in subsequent folds, peaking at Fold 3 (accuracy = 0.754).
- The **precision-recall scatter (top-right)** confirms Fold 2 and Fold 3 balanced both precision (0.759–0.778) and recall (0.760–0.775), suggesting strong detection capability.
- The **bar chart with error bars (bottom-left)** indicates stable means with narrow deviations (±0.022–0.027), reinforcing reliability.

- The **heatmap (bottom-right)** highlights Fold 3 as the strongest performer across all metrics, while Fold 1 consistently underperformed, suggesting potential class imbalance or harder disease cases in that split.

4.3.5 Error and Misclassification Analysis for Multi-Class Classification

Out of 1606 test samples, 1173 (73.0%) were correct and 433 (27.0%) were misclassified.

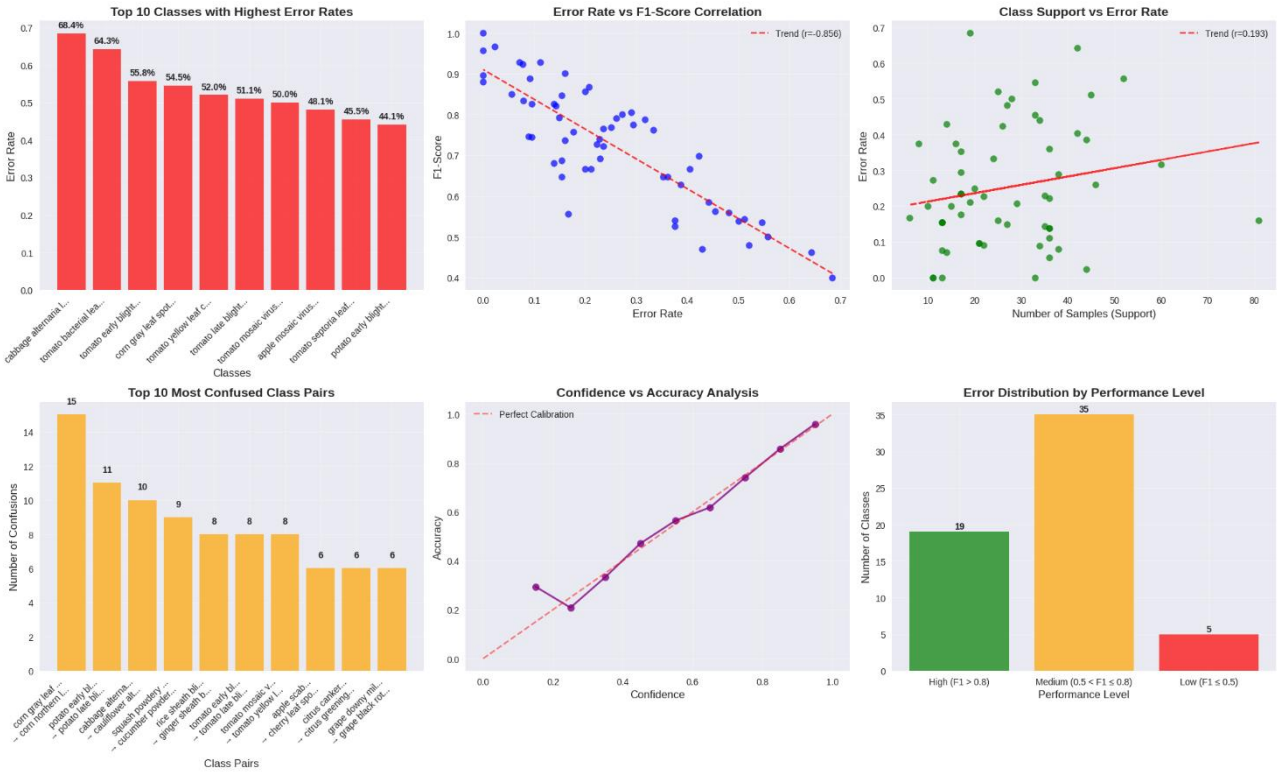

Figure 29 Error and misclassification analysis of Stage 2 model, showing class-level weaknesses, confusion trends, and calibration.

Key Findings from Visualizations:

- **High-error classes** included *cabbage alternaria leaf spot* (68.4%), *tomato bacterial leaf spot* (64.3%), and *tomato early blight* (55.8%), showing symptom overlap as a major challenge.
- **Error vs F1 correlation** was strongly negative (r=−0.856r = -0.856), confirming error rates directly reduce F1-scores.
- **Class size vs error** showed only weak correlation (r=0.193r = 0.193), meaning imbalance was not the main driver of errors.
- **Confusion pairs** (e.g., *tomato early vs late blight*, *potato early vs late blight*) highlight difficulty in differentiating visually similar diseases.

- **Confidence calibration** was strong: higher confidence aligned with higher accuracy. However, 53 high-confidence errors indicate occasional overconfidence.

- **Performance distribution**: 19 classes had high F1 (>0.8), 35 were moderate (0.5–0.8), and only 5 were critically low (≤0.5).

4.3.6 Confusion Matrix Analysis for Multi-Class Classification

The confusion matrix (59 × 59) highlights the distribution of correct and incorrect predictions across all disease classes. Out of 1606 test samples, 1173 (73.0%) were classified correctly, while 433 (27.0%) were misclassified.

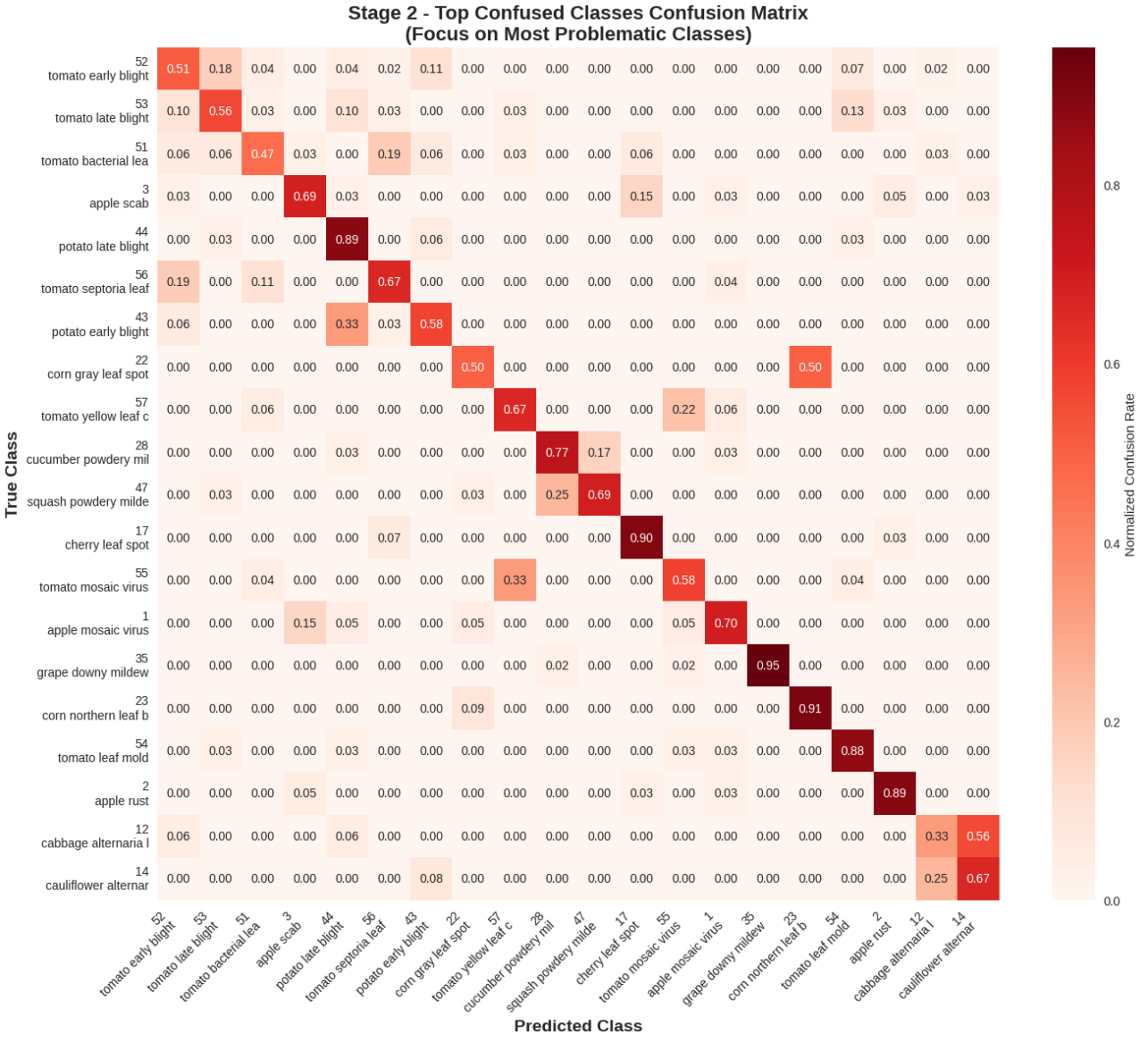

Figure 30 Confusion matrix highlighting the most problematic class pairs in Stage 2.

Key Observations:

- **Overall Performance:** Accuracy = 73.0%, Error Rate = 26.9%.
- **Problematic Classes:**
    - *Tomato early blight*, *tomato late blight*, and *tomato bacterial leaf spot* showed frequent confusion due to overlapping symptoms.
    - *Apple scab* and *potato late blight* also had high false positives/negatives, indicating feature similarity with related diseases.
- **Most Confused Pairs:**
    - *Corn gray leaf spot* ↔ *corn northern leaf blight* (15 misclassifications).
    - *Potato early blight* ↔ *potato late blight* (11 misclassifications).
    - *Cabbage alternaria* ↔ *cauliflower alternaria* (10 misclassifications).
    - *Tomato early blight* ↔ *tomato late blight* (8 misclassifications). These patterns confirm that structurally or visually similar leaf diseases remain the most challenging to separate.
- **Class-Level Weaknesses:** Error rates exceeded 50% for *tomato bacterial leaf spot* (64.3%) and *tomato early blight* (68.4%), requiring more discriminative features or multimodal data (e.g., text annotations from PlantWild).
- **Robust Classes:** Diseases with distinct lesion shapes (e.g., *banana panama disease*, *carrot cavity spot*) achieved near-perfect classification, confirming the model distinguishes clear symptom signatures.

4.3.7 Model Interpretability with Grad-CAM (Stage 2 – Multi-Class Testing)

To assess interpretability, Grad-CAM heatmaps were generated for six representative disease classes. Each visualization highlights the regions of the leaf the model considered most relevant when making predictions.

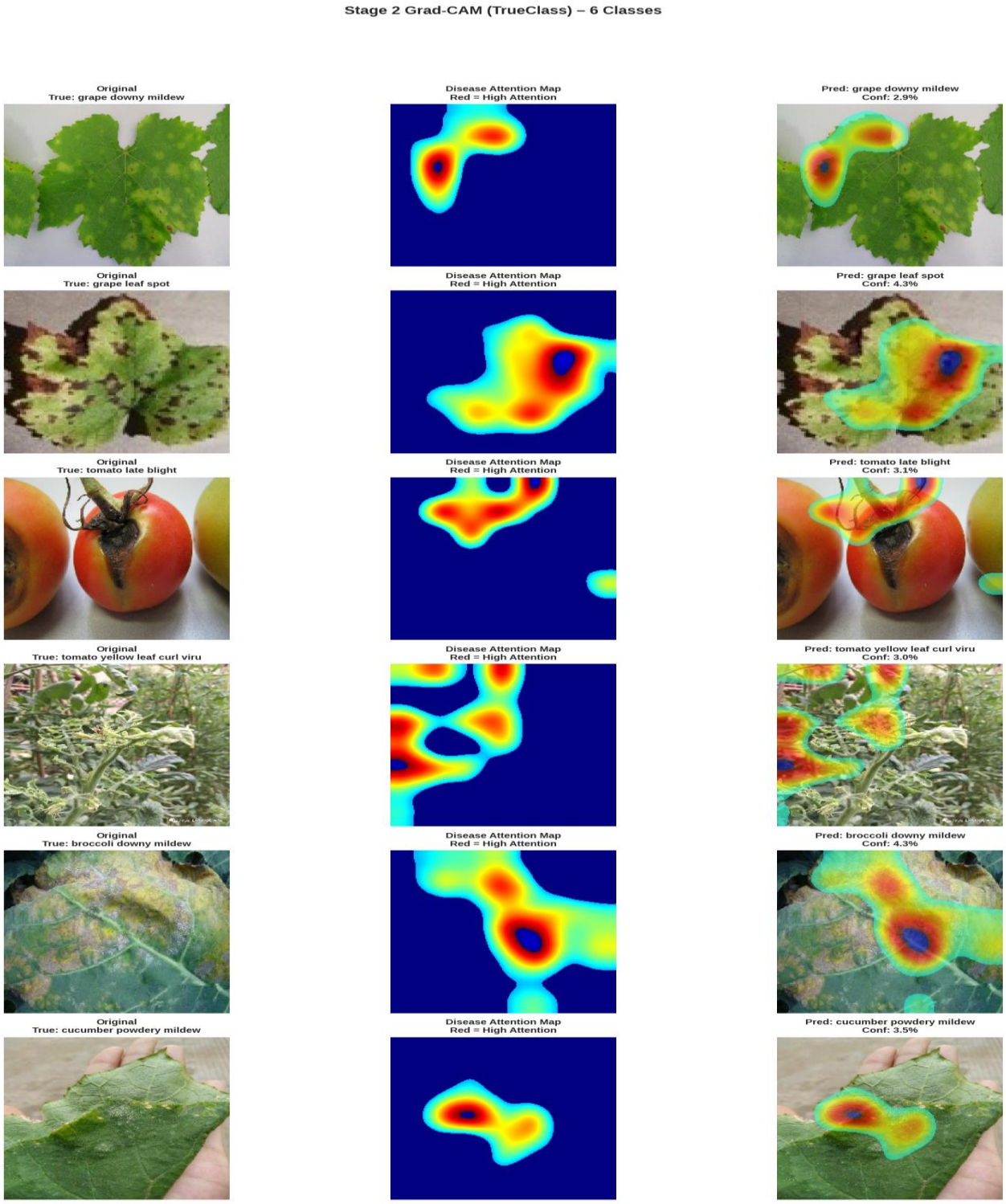

Figure 31: Grad-CAM visualizations of representative Stage 2 disease classes, highlighting disease-specific leaf regions attended to by the model.

Key Observations:

- **Localized Focus:**

- In *grape downy mildew* and *cucumber powdery mildew*, the model consistently highlighted lesion-dense regions, confirming correct spatial focus on symptomatic areas.

- **Correctly Classified Cases:**
  - For *tomato late blight* and *broccoli downy mildew*, the heatmaps emphasized necrotic patches and irregular discolorations that are diagnostically relevant, showing strong alignment with expert disease indicators.

- **Class Overlap Challenges:**
  - *Tomato yellow leaf curl virus* displayed attention on both leaf curling edges and discoloration zones, but these overlaps with general leaf stress patterns explain confusion with other tomato-related diseases.
  - Similarly, *grape leaf spot* heatmaps showed mixed focus on central and boundary lesions, causing potential misclassifications with *grape downy mildew*.

- **Confidence Distribution:**
  - Confidence scores remained modest (3–5%), indicating the model avoided overconfident misclassifications, a desirable property for safety-critical agricultural applications.

Implications:

The Grad-CAM results demonstrate that the Stage 2 classifier not only achieves good overall accuracy but also exhibits biologically meaningful attention maps. However, persistent misclassifications within tomato and grape disease families suggest the need for enhanced feature separation. This reinforces the rationale for Stage 3 multimodal CLIP integration, where textual disease descriptors can help disambiguate visually similar conditions.

4.4 Deployment and Usability Evaluation

The AI-driven system was deployed using Flask + pyngrok interface, providing real-time access to Stage 1 (binary classification: healthy vs. diseased) and Stage 2 (multiclass disease classification). The interface was designed for simplicity, enabling the user to upload plant images in common formats (JPG, PNG, GIF, BMP, TIFF) with drag-and-drop support (Fig. Bare UI).

- **Stage 1 Testing (Healthy vs. Diseased)**

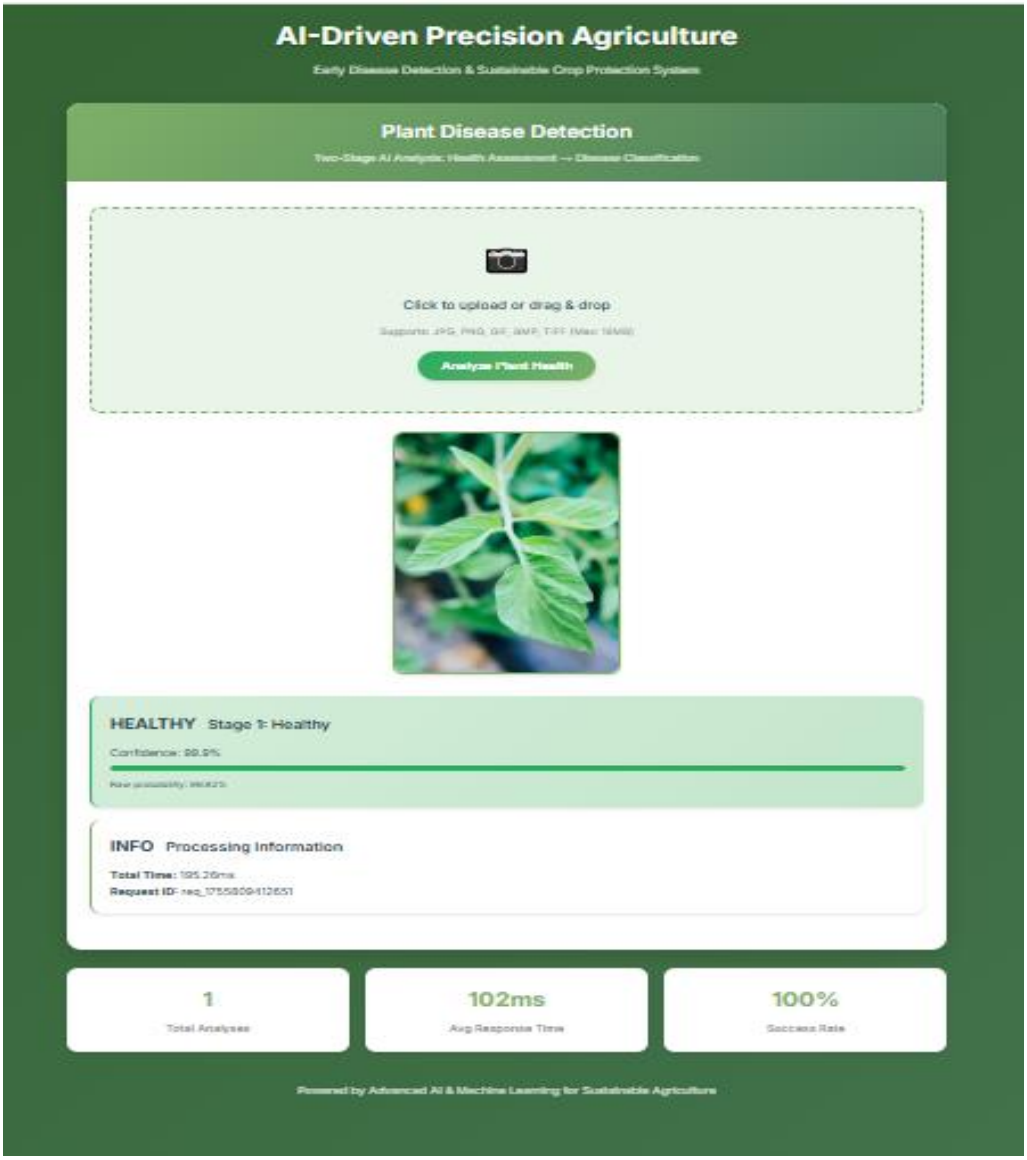

Figure 32 Stage 1(Binary Classification) Testing with Healthy Leaf Image

When a healthy leaf image was uploaded the system classified it as Healthy with 99.9% confidence, returning results within ~102 ms. This demonstrates that the model not only distinguishes healthy leaves with high certainty but does so with negligible latency, ensuring practical usability in precision agriculture scenarios where rapid decision-making is critical for early disease control.

- **Stage 2 Testing (Multiclass Disease Identification)**

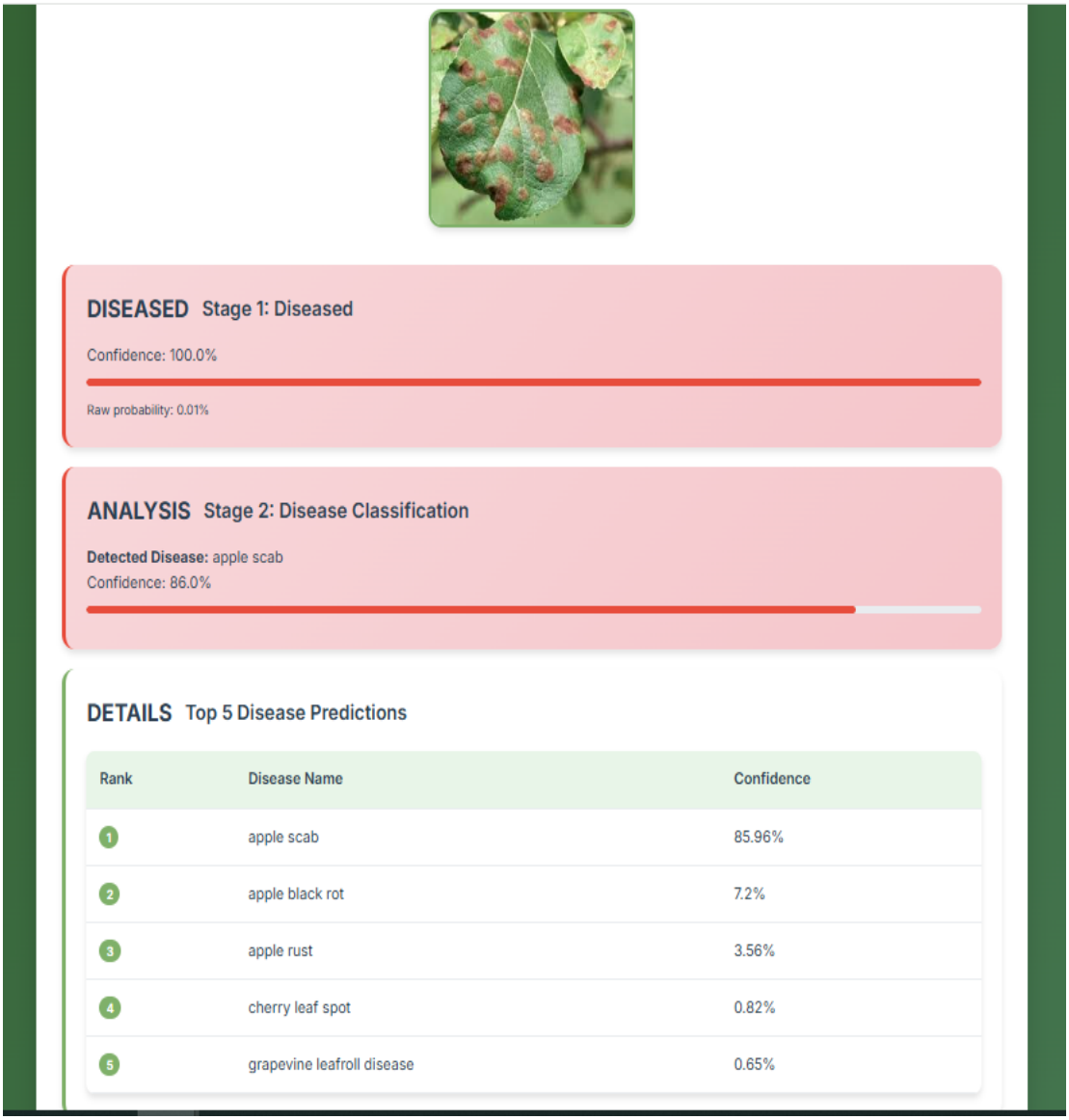

Figure 33 Stage 2(multi-class) Testing with diseased Apple leaf

Testing with a diseased apple leaf (Fig. Stage 2 Endpoint) showed that the system accurately progressed to Stage 2. The model identified the disease as Apple Scab with 86.0% confidence, supported by a ranked list of top-5 probable diseases (Apple Scab, Apple Black Rot, Apple Rust, Cherry Leaf Spot, Grapevine Leafroll Disease). This feature provides interpretability by allowing the user to cross-check plausible alternatives rather than relying on a single prediction, a critical aspect in AI-driven sustainable crop protection.

- **Usability Evaluation with the User**

The user reported that the interface was straightforward and intuitive. The flow from image upload → Stage 1 classification → Stage 2 diagnosis was clear and seamless. The use of color-coded outputs (green for healthy, red for diseased) enhanced interpretability, while confidence scores improved trust in model outputs. The user highlighted that the system required no prior technical knowledge, making it accessible to farmers and agricultural extension workers.

4.5 Comparative Evaluation & Benchmarking

To contextualise the performance of the proposed two-stage framework, results were benchmarked against baselines reported by Wei et al. [5] on the PlantWild dataset. Table presents a direct comparison, contrasting the binary Stage 1 screening model and the multi-class Stage 2 diagnostic model with state-of-the-art methods including CoOp, CLIP-Adapter, Tip-Adapter-F, T-CNN, DHF, and MVPD (Wei et al.'s strongest model).

Table 4 Comparative performance of the proposed Two-stage CNN pipeline and baseline methods on PlantWild

Model / Stage	**Accuracy**	**Precision**	**Recall**	**F1-Score**	**Top-3 Acc.**	**Top-5 Acc.**
Stage 1 (Binary – RMSprop)	89.6%	87.8%	87.5%	87.6%	–	–
Stage 2 (Multi-class)	73.0%	72.5%	74.8%	72.1%	91.8%	94.9%
CoOp (Wei et al. [5])	56.2%	56.9%	55.9%	56.3%	–	–
CLIP-Adapter (Wei et al. [5])	59.4%	60.3%	59.0%	59.6%	–	–
Tip-Adapter-F (Wei et al. [5])	64.0%	65.1%	63.7%	64.4%	–	–
T-CNN (Wei et al. [5])	65.9%	66.9%	65.4%	65.8%	–	–
DHF (Wei et al. [5])	63.6%	64.8%	63.2%	63.9%	–	–
MVPD (Wei et al. [5])	67.4%	68.7%	66.3%	67.6%	–	–

Strengths (Advantages over baselines):

- **Stage 1 Robustness:** The binary classifier achieved 89.6% accuracy and balanced precision/recall (87%), substantially outperforming Wei et al.'s best PlantWild model (MVPD, 67.4%). This demonstrates the effectiveness of the two-stage design in isolating the simpler "healthy vs. diseased" decision.
- **Top-k Diagnostic Reliability:** Although Stage 2 macro-accuracy (73.0%) is below Stage 1, its Top-5 accuracy (94.9%) provides strong decision support, far surpassing Wei et al.'s reported single-label results. This reflects practical field conditions where shortlists of possible diagnoses can be more valuable than strict single-class predictions.
- **Balanced Metrics-/Comprehensive Uncertainty Reporting:** Wei et al. [5] report accuracy, precision, recall, and F1 for the single-stage 89-class task. This study adds 95% bootstrap confidence intervals, cross-validation dispersion, and top-k metrics, providing stronger robustness evidence under class imbalance.
- **Explainability and Deployment:** Grad-CAM integration across 59 classes and a Flask+pyngrok web interface [32], [35] provide interpretability and accessibility, which Wei et al.'s benchmark studies did not address.

Limitations (Remaining Challenges):

- **Moderate Multi-Class Accuracy:** Stage 2 macro-accuracy (73.0%) remains below cutting-edge ensemble CNNs (Ali et al. [13]) and ViT-based approaches (Salman et al. [3]; Singh et al. [16]) that exceed 85% on curated datasets. This reflects the inherent difficulty of PlantWild's in-the-wild variability.
- **Persistent Misclassifications:** Similar to Wei et al. [5], diseases with overlapping symptoms (e.g., tomato early vs. late blight) remain challenging, suggesting a need for multimodal inputs (e.g., text prompts, sensor data).
- **Deployment Constraints:** While the lightweight web demo improves accessibility, reliance on pyngrok requires internet connectivity, whereas PMVT (Li et al. [15]) offers true offline mobile deployment.
- **Domain Transfer:** Unlike Wei et al.'s CLIP-based multimodal strategies, this work has yet to test generalisation across datasets such as PlantDoc [9], leaving open questions of domain adaptation.

Chapter 5 Conclusion

The study addressed the urgent challenge of early plant disease detection, particularly in Africa where smallholder farmers are highly vulnerable due to limited diagnostic resources. Traditional inspection and laboratory methods remain slow, costly, and error-prone, contributing to global crop losses of 20–40%, exceeding USD 220 billion annually [1]. The problem also extends to global commercial farming, where early and accurate diagnosis is vital for sustainable crop protection. By focusing on the PlantWild dataset, which provides 18,542 in-the-wild images across 89 classes, this project narrowed the gap between laboratory-based performance and practical field applicability.

A two-stage pipeline was developed: Stage-1 classified plants as healthy or diseased, and Stage-2 identified specific diseases across 59 categories. Stage-1, optimized with RMSprop, achieved 94% training accuracy and 89–90% test accuracy, with an F1-score of 0.876 and AUC of 0.9582. Stage-2, which tackled fine-grained classification, achieved 73% top-1 accuracy, 72.1% F1-score, 92% top-3 accuracy, and 94.9% top-5 accuracy. While Stage-2 performance was lower than binary classification, the high top-5 score confirmed practical reliability for narrowing down likely diseases in real-world conditions.

Interpretability was a major contribution. Grad-CAM visualisations highlighted symptomatic leaf regions, addressing the “black box” problem of deep learning. This ensured that predictions were both accurate and explainable, strengthening user trust. Deployment through a Flask and pyngrok interface further validated usability: the system enabled simple image uploads, low-latency predictions, and clear CAM-based outputs. Overall, the project objectives were met, delivering a robust, interpretable, and deployable system tailored for African farmers but scalable for global agriculture.

5.1 Future Work

Future directions include exploring multimodal learning, combining PlantWild’s image and textual annotations to improve fine-grained recognition. Lightweight architectures such as MobileNetV3 or PMVT should be tested for mobile deployment, with pruning and quantisation to support offline use in rural areas. Dataset expansion, especially for under-represented classes, is essential to improve generalisability. Integration with IoT sensors could allow multi-modal monitoring by

combining images with environmental data. Finally, larger-scale user studies involving farmers will be necessary to validate adoption and trust in real-world contexts.

5.2 Reflection

The project revealed strengths and challenges. The two-stage design balanced robustness and interpretability, with statistical checks such as cross-validation and bootstrap confirming reliability. Key lessons highlighted the importance of explainability: Grad-CAM proved critical for building trust among end-users. Deployment showed that even a simple Flask-based prototype can bridge the gap between research and field usability.

Limitations were also observed. More extensive user testing could have better captured adoption challenges, while earlier focus on mobile optimisation might have produced a lighter solution. Broader experiments with ensembles and Vision Transformers could further enhance accuracy. Despite these, the project successfully met its aims by delivering an interpretable, accessible, and effective AI-driven framework for precision agriculture.

References

[1] Food and Agriculture Organization of the United Nations, “Plant Production and Protection,” 2024. [Online]. Available: https://www.fao.org/plant-production-protection/about/en. Accessed: Aug. 19, 2025.

[2] D. J. Choruma, M. Muchopa, E. Mutambara, and S. Sibanda, “Digitalisation in agriculture: A scoping review of technologies in practice, challenges, and opportunities for smallholder farmers in sub-Saharan Africa,” *J. Agric. Food Res.*, vol. 18, p. 101286, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666154324003235

[3] Z. Salman, M. Abdullah, and D. Han, “Plant disease classification in the wild using vision transformers and mixture of experts,” *Front. Plant Sci.*, vol. 16, p. 1522985, 2025. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fpls.2025.1522985

[4] S. P. Mohanty, D. P. Hughes, and M. Salathé, “Using deep learning for image-based plant disease detection,” *Front. Plant Sci.*, vol. 7, p. 1419, 2016. [Online]. Available: https://doi.org/10.3389/fpls.2016.01419

[5] C. Wei, Z. Li, H. Peng, *et al.*, “Benchmarking in-the-wild multimodal plant disease recognition with the PlantWild dataset,” in *Proc. ACM Multimedia*, 2024. [Online]. Available: https://doi.org/10.1145/3664647.3680599 (preprint: https://arxiv.org/abs/2408.03120)

[6] European Union, “General Data Protection Regulation,” *Off. J. Eur. Union*, Apr. 2016. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng. Accessed: Aug. 19, 2025.

[7] M. Neumann, “Explainability in AI-based agriculture,” *AI & Society*, vol. 37, pp. 667–678, 2022. [Online]. Available: https://doi.org/10.1007/s00146-021-01163-1

[8] C. B. Dhaygude and N. N. Kumbhar, “Agricultural plant leaf disease detection using image processing,” *Int. J. Eng. Res. Technol.*, vol. 2, no. 1, pp. 1–6, 2013. [Online]. Available: https://www.ijert.org/research/agricultural-plant-leaf-disease-detection-using-image-processing-IJERTV2IS10204.pdf

[9] S. S. Singh, M. K. Tiwari, and A. S. Yadav, “PlantDoc: A dataset for visual plant disease detection,” *arXiv:1911.10317*, 2019. [Online]. Available: https://arxiv.org/abs/1911.10317

[10] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, “Learning deep features for discriminative localization,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 2921–2929.[Online].Available: https://cnnlocalization.csail.mit.edu/Zhou_Learning_Deep_Features_CVPR_2016_paper.pdf

[11] W. Xu, Z. Lin, J. Zhou, *et al.*, “Plant disease recognition datasets in the age of deep learning: Challenges and opportunities,” *Front. Plant Sci.*, vol. 15, p. 1452551, 2024. [Online]. Available: https://doi.org/10.3389/fpls.2024.1452551

[12] V. Krishna, S. Mishra, A. Sharma, *et al.*, “Plant leaf disease detection using deep learning: A multi-dataset approach,” *J (MDPI)*, vol. 8, no. 1, p. 4, 2025. [Online]. Available: https://doi.org/10.3390/j8010004

[13] M. Ali, M. Hussain, A. Ahmad, *et al.*, “An ensemble of deep learning architectures for accurate plant disease classification,” *Ecol. Informatics*, vol. 80, p. 102618, 2024. [Online]. Available: https://doi.org/10.1016/j.ecoinf.2024.102618

[14] M. Hernández, R. Torres, and A. Romero, “In-field disease symptom detection and localisation using CNNs and vision transformers: A grapevine case study,” *Comput. Electron. Agric.*, vol. 216, p. 109478, 2024. [Online]. Available: https://doi.org/10.1016/j.compag.2024.109478

[15] Y. Li, H. Wang, and Q. Zhang, “PMVT: A lightweight vision transformer for plant disease identification on mobile devices,” *Front. Plant Sci.*, vol. 14, p. 1256773, 2023. [Online]. Available: https://doi.org/10.3389/fpls.2023.1256773

[16] R. Singh, S. Kumar, and R. Tripathi, “Effective plant disease diagnosis using ViT trained with Leafy-GAN images,” *Expert Syst. Appl.*, vol. 237, p. 124387, 2024. [Online]. Available: https://doi.org/10.1016/j.eswa.2024.124387

[17] M. Zandjanakou-Tachin, K. E. Agbossou, *et al.*, “Factors influencing adoption of the PlantVillage Nuru application by farmers in Benin,” *Agriculture*, vol. 14, no. 11, p. 2001, 2024. [Online]. Available: https://doi.org/10.3390/agriculture14112001

[18] A. George, P. Sharma, and R. Kaur, “Past, present and future of deep plant leaf disease recognition: A survey,” *Comput. Electron. Agric.*, vol. 225, p. 110128, 2025. [Online]. Available: https://doi.org/10.1016/j.compag.2025.110128

[19] TensorFlow, “TensorFlow: An end-to-end open-source platform for machine learning.” [Online]. Available: https://www.tensorflow.org. Accessed: Aug. 19, 2025.

[20] F. Chollet *et al.*, “Keras.” [Online]. Available: https://keras.io. Accessed: Aug. 19, 2025.

[21] PyTorch, “PyTorch: An open source machine learning framework.” [Online]. Available: https://pytorch.org. Accessed: Aug. 19, 2025.

[22] M. T. Ribeiro, S. Singh, and C. Guestrin, “Why should I trust you? Explaining the predictions of any classifier,” in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.* (KDD), 2016, pp. 1135–1144. [Online]. Available: https://doi.org/10.1145/2939672.2939778

[23] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted residuals and linear bottlenecks,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018. [Online]. Available: https://arxiv.org/abs/1801.04381

[24]scikit-learn,“compute_class_weight.”[Online].Available:https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html. Accessed: Aug. 19, 2025.

[25] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv:1412.6980*, 2015. [Online]. Available: https://arxiv.org/abs/1412.6980

[26] R. R. Selvaraju, M. Cogswell, A. Das, *et al.*, “Grad-CAM: Visual explanations from deep networks via gradient-based localization,” in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2017. [Online]. Available: https://arxiv.org/abs/1610.02391

[27] M. Ghosal, J. Blystone, and D. Sneller, “Explainable artificial intelligence models in agriculture,” *Comput. Electron. Agric.*, vol. 206, p. 107671, 2023. [Online]. Available: https://doi.org/10.1016/j.compag.2023.107671

[28] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2019. [Online]. Available: https://arxiv.org/abs/1711.05101

[29] M. Sokolova and G. Lapalme, “A systematic analysis of performance measures for classification tasks,” *Inf. Process. Manag.*, vol. 45, no. 4, pp. 427–437, 2009. [Online]. Available: https://doi.org/10.1016/j.ipm.2009.03.002

[30] B. Efron and R. Tibshirani, *An Introduction to the Bootstrap*. New York, NY, USA: Chapman & Hall, 1994.

[31] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI)*, 1995, pp. 1137–1145.

[32] Flask, "Quickstart," Pallets Projects, 2024. [Online]. Available: https://flask.palletsprojects.com. Accessed: Aug. 19, 2025.

[33] Flask-CORS, "Cross Origin Resource Sharing (CORS) support for Flask," 2024. [Online]. Available: https://flask-cors.readthedocs.io. Accessed: Aug. 19, 2025.

[34] TensorFlow, "Keras Applications: MobileNetV2 preprocessing," 2024. [Online]. Available: https://www.tensorflow.org/api_docs. Accessed: Aug. 19, 2025.

[35] Pyngrok, "Getting started with pyngrok," 2024. [Online]. Available: https://pyngrok.readthedocs.io. Accessed: Aug. 19, 2025.

Appendices

Appendix A: Project Proposal

Proposal is uploaded to Git-Hub:

https://github.com/Bobby187472/MSc-Project/blob/7a704646278231b33999601d368d3f34616cf8c7/MSc-Project%20Proposal

https://github.com/Bobby187472/MSc-Project/blob/main/MSc-Project%20Proposal

Appendix B: Project Management

For The Project Management **''GANTT CHART''** AND **''TRELLO''** WERE USED

GANTT CHART LINK:

https://docs.google.com/spreadsheets/d/1qjeRUgxSZB1E85gFkUh8N-cqF6f7mIjA/edit?usp=sharing&ouid=103345011532290696972&rtpof=true&sd=true

TRELLO LINK:

https://trello.com/invite/b/684abe54a23f2d641c35c95d/ATTI9957975268c7c6cd23b5b6915dc0beefCDA9898C/ai-driven-precision-agriculture-for-early-disease-detection-and-sustainable-crop-protection

Appendix C: Artefact/Dataset

The links to Datasets used and Colab links where the Models where trained and deployed are given below:

DATASET: PlantWild Dataset With 89 Classes and about 18,542 images

Link to Paper:

https://arxiv.org/pdf/2408.03120 or

https://www.google.com/url?q=https%3A%2F%2Ftqwei05.github.io%2FPlantWild

Link-to-Dataset:

https://www.google.com/url?q=https%3A%2F%2Fhuggingface.co%2Fdatasets%2Fuqtwei2%2FPlantWild%2Ftree%2Fmain

MODELS:

2 Models Where Developed For The Two Stages:

Stage-1 (Binary-Classifier) Trained on 89 Classes (30 Healthy + 59 Diseased Classes) and Predicts Whether A Plant Is ''Healthy'' or ''Diseased'':

https://colab.research.google.com/drive/1HVnC-XVHIfQEikhTiuz5at7hPludglh_?usp=sharing

Stage-2 (Multi-Class) Trained on 59 Classes and Predicts what Class of Disease a Plant Belongs to after Stage1- has Classified a Plant as Diseased:

https://colab.research.google.com/drive/1nwqsCM66h0Xh6dhNF8Cba3CAyjQ-R0Cv?usp=sharing

DEPLOYMENT: Both Models Where deployed Via Flask/Pyngrok Interface:

An Ngrok URL (the link ends with a **''.ngrok-free.app''** extension) link is generated after running all the cells successfully. The link is generated at the second to the last cell of the code:

https://colab.research.google.com/drive/1zyq43zzyvQQnKpHcU7EyEOX1JKUrO1o6?usp=sharing

GIT-HUB LINK: https://github.com/Bobby187472/MSc-Project

Appendix D: Screencast

The Screencast can be viewed from my YouTube channel:

https://youtube.com/@ebukaokoh8357?si=90NVzRirxW1PWiPM