WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

YOLO-Based Rice Disease Detection For Precision Agriculture

Swejal Wasnik[1], Niharika Rokade[2], Khushbu Bagade[3], Gayatri Padole[4], Vaishnavi Vaidya[5], Prof. Yogesh B. Malode*[6]

Department of Information Technology, Priyadarshini Bhagwati College Of Engineering, Nagpur 440009, Maharashtra, India
*Corresponding Author (Mentor)

ABSTRACT :

Agricultural productivity is frequently reduced by leaf diseases, and their symptoms are difficult for non-specialists to tell apart by visual inspection. This paper presents LeafScan, an AI-based plant health detection system that analyses images of plant leaves and returns preliminary information about the plant category and its possible health condition. The system is organised into three cooperating layers: a React and Vite presentation layer offering registration, login, a dashboard, image upload with preview and a scan-history view; a Python FastAPI service layer that handles token-based authentication, receives uploaded images and exposes a scan-history API; and persistence in SQLite accessed through SQLAlchemy. YOLO11 classification model trained on a rice-leaf dataset of 5,932 images covering Bacterial Blight, Blast, Brown Spot . A local vision-language model, qwen2.5vl:3b served through Ollama, was explored for generating explanatory descriptions without sending images to a cloud service. The paper documents the requirements, design, implementation and testing status, and records the current limitations: the scan API still produced runtime errors in testing, the class mapping and prediction confidence of the models require validation, and the local vision-language model produced inconsistent output on some requests. Final accuracy figures are therefore not claimed. LeafScan is intended as a decision-support tool for preliminary screening and not as a replacement for professional agricultural diagnosis.

Keywords: Plant Disease Detection, Rice Leaf Disease, EfficientNet-B0, YOLO11, FastAPI, React, SQLite, Ollama, Vision-Language Model, Deep Learning.

I. INTRODUCTION

Plant diseases affect the growth, quality and yield of crops and are a continuing concern for farmers, gardeners and agricultural students [11]. Many diseases first appear as visible changes on the leaves, such as spots, discolouration, lesions or curling, but the same symptom can be produced by different causes. Recognising it reliably usually requires an experienced plant pathologist, and access to such expertise is often limited, especially for small growers.

Advances in deep learning and computer vision have made it possible to classify leaf images automatically. Convolutional neural networks such as EfficientNet [17] have reported high accuracy on public leaf-disease benchmarks such as PlantVillage [13][15], and recent systems combine classification with severity estimation and generative advice [1] or with vision-language models [3][6]. For rice, transfer-learning and feature-based approaches have been applied to diseases such as bacterial blight, blast, brown spot and tungro [7][10][12], and the YOLO11 model family now supports image classification alongside detection and segmentation [8]. However, high benchmark accuracy does not by itself show that a model is dependable inside a complete application used with real photographs [6].

LeafScan is developed as a simple web application in which a user can register, log in, upload a leaf image and view the analysis returned by the prediction workflow, while previous scans are retained in a history list. It is implemented with a React and Vite frontend [19][20], a FastAPI backend [21] with SQLite storage [22] and user authentication, YOLO11 classification models built with Train and Validation [8][23], and an optional local vision-language model (qwen2.5vl:3b) [5] served through Ollama [24].

The architecture separates the classifier from the explanation model so that either can be replaced or improved independently. Because the classifiers and the vision-language model can run on the user's own machine, uploaded images do not have to be sent to an external cloud service. The usefulness of the system nevertheless depends on the quality of the training data, correct class mapping, image quality and the successful operation of every component in the chain from upload to stored result.

The current implementation has known limitations. In testing, the scan API still produced runtime errors, the class mapping and prediction confidence of the models need validation, and the local vision-language model produced inconsistent output on some requests. For this reason the report does not claim a final accuracy figure; the presence of a feature in the design or code should not be interpreted as proof of reliability. LeafScan should be treated as a decision-support tool for preliminary screening rather than a substitute for professional agricultural diagnosis.

The main objective of LeafScan is to provide an integrated, browser-based approach to preliminary plant health screening by combining user authentication, leaf-image upload and validation, classifier-based prediction, an optional locally hosted explanation model and a persistent scan history. The overall contribution of the work is this integration within a single working prototype, with its practical effectiveness to be determined through further verification and systematic evaluation.

II. PROBLEM STATEMENT

Manual inspection of plant symptoms is time-consuming, and users often have limited access to agricultural experts. Different diseases can show similar visible symptoms, which makes identification difficult for non-specialists [11]. Users also need a convenient way to upload plant images, receive preliminary information and keep their previous analyses in one place.

The motivation for LeafScan is to make image-based plant health screening accessible through an ordinary web browser, to reduce the effort required for initial screening, and to demonstrate how deep learning, web technologies and a locally hosted vision-language model can be integrated into a single practical system. Since the models may misclassify images and the explanation model may be inconsistent [6], the limits of the system must also be tested and reported.

Problem Statement: To develop a web-based AI application that analyses plant leaf images and provides preliminary plant identification and health assessment, while evaluating its functionality, reliability and limitations.

III. LITERATURE REVIEW

The literature review examines recent deep-learning approaches to plant and rice leaf disease detection, the use of vision-language models for agricultural analysis, YOLO-based classification, and foundational work on transfer learning and EfficientNet. Recent publications are discussed first, followed by earlier foundational work, and the numbers correspond to the reference list at the end of the paper.

3.1 Recent Deep-Learning Frameworks for Plant Disease Detection (2026–2025)

Agrawal et al. [1] presented an integrated framework that pairs a VGG-19 classifier with Grad-CAM-based severity estimation and a generative-AI module for precautions and treatment recommendation, reporting 85% accuracy on a public dataset. Ranjan et al. [2] compared a baseline CNN with EfficientNetB0, MobileNetV2 and ResNet50 on about 87K leaf images in 38 classes; EfficientNetB0 gave the strongest results while MobileNetV2 had the lowest latency. Chaudhary [4] evaluated DenseNet and VGG19 features with SVM, Random Forest and Logistic Regression on PlantVillage, where DenseNet201 with SVM performed best.

3.2 Vision-Language Models in Agriculture (2026–2024)

Mahmood et al. [3] fine-tuned Qwen-2.5-VL-3B on about 11,000 expert-curated leaf images with verified explanations and reached 73.1% accuracy on a 1,000-image test set. The Qwen2.5-VL technical report [5] describes the model series, including the 3B variant that LeafScan runs locally through Ollama, and the earlier Qwen2-VL [9] introduced dynamic-resolution image handling for vision-language models. Ranario et al. [6] benchmarked vision-language models on 27 agricultural classification datasets (162 classes) and found that open models trail the best proprietary ones, with pest and damage identification the hardest task. These findings motivate treating the output of the local model in LeafScan as a supporting explanation rather than a diagnosis.

3.3 Rice Leaf Disease Detection and YOLO11 (2023–2024)

Ritharson et al. [7] applied transfer learning to a 5,932-image rice-leaf dataset covering bacterial blight, blast, brown spot, tungro and severity subtypes, and compared several pretrained networks. Aggarwal et al. [10] extracted features with 32 pretrained networks and classified rice diseases with machine-learning and ensemble classifiers, reaching 94% accuracy. Sudhesh et al. [12] studied ten transfer-learned CNNs on 3,416 images of four rice diseases, with DenseNet121 achieving 93.87%. ValidationYOLO11 [8] is an open-source model family supporting detection, segmentation and image classification, and is the basis of the rice classifier in LeafScan.

3.4 Reviews, EfficientNet and Foundational Work (20219–2023)

Mustofa et al. [11] reviewed ViT, CNN and YOLO-based leaf-disease models and the public datasets used. Bruno et al. [13] used two EfficientNet-B0 models with a trainable ensembling layer on PlantVillage to improve accuracy at low complexity, Bui et al. [14] applied Noisy-Student EfficientNet models with transfer learning, with B3 and B5 performing best on PlantVillage variants, and Atila et al. [15] trained EfficientNet B0–B7 on PlantVillage, with B5 and B4 giving the best accuracy on the original and augmented sets. Among earlier foundational work, Dosovitskiy et al. [16] introduced the Vision Transformer, Tan and Le [17] proposed compound scaling of depth, width and resolution, of which EfficientNet-B0 is the baseline network used in LeafScan, and Too et al. [18] compared fine-tuned VGG, Inception, ResNet and DenseNet models on PlantVillage, where DenseNets gave the best results.

3.5 Existing Approaches and Their Limitations

Existing approaches to leaf-disease identification fall into three groups: traditional visual inspection by farmers or experts; research prototypes applying convolutional networks to curated datasets [13][15][18]; and more recent work that attaches severity estimation or generative advice to a classifier [1] or uses vision-language models for open-ended diagnosis [3][6]. Their main limitations are:

1. Manual inspection is slow, subjective and dependent on the availability of an expert.
2. Many research models are evaluated only on laboratory-style images and may generalise poorly to real field photographs [6].
3. Several systems are research code only and do not provide a complete application with user accounts and stored history.
4. General-purpose vision-language models are not always reliable at disease identification and can give inconsistent explanations [6].
5. Cloud-based services require sending user images to a remote server, which raises privacy and connectivity concerns.

3.6 Proposed System and Comparison

LeafScan is proposed as an end-to-end web application rather than a stand-alone model. A React frontend lets the user register, log in, upload a leaf image and view the result. A FastAPI backend authenticates the user, preprocesses the image, calls the configured classifier (EfficientNet-B0 for plant and disease experiments, YOLO11 for rice diseases) and, optionally, a locally hosted qwen2.5vl:3b model through Ollama for an explanatory description. Scan information is stored in SQLite so that users can review their scan history. Table 1 compares the existing approaches with the proposed system.

Table 1: Comparison of existing approaches and the proposed LeafScan system

Aspect	Existing Approaches	Proposed LeafScan System
Method	Manual inspection; research CNN prototypes on curated datasets	End-to-end web application using EfficientNet-B0 and YOLO11 classifiers
Delivery	Often research code or stand-alone models	React + FastAPI + SQLite application with accounts, token authentication and scan history
Explanation	Severity or generative advice [1]; open-ended VLM diagnosis that may be inconsistent [3][6]	Optional local qwen2.5vl:3b via Ollama, presented only as supporting explanation
Privacy	Cloud services require sending images to a remote server	Classifiers and vision-language model can run on the user's own machine
Modularity	Classifier and explanation tightly coupled in many prototypes	Classifier and explanation model kept separate and replaceable
Safe handling	Not always addressed	Unsupported or low-confidence cases to be reported safely (design requirement; to be verified)

3.7 Summary of the Literature and Research Gap

The selected references cover plant and rice disease classification, EfficientNet and transfer learning, YOLO11, vision-language models and reviews of deep-learning methods [1]–[18]. They do not establish that a complete web application combining authentication, leaf-image validation, EfficientNet-B0 and YOLO11 classification, a locally hosted explanation model and stored scan history has been implemented and evaluated together in the manner of LeafScan. The project therefore focuses on implementing and testing this integrated workflow, with its practical effectiveness to be determined through testing rather than assumed in advance.

IV. RESEARCH GAP, AIM AND OBJECTIVES

4.1 Research Gap

Previous studies have explored deep-learning classification of leaf images, rice-disease recognition and the use of vision-language models for explanation [1][3][7][10]. Existing technical resources also provide tools for web development, model training and local model serving [8][19][21][23][24]. However, the selected references do not establish that these elements have been combined and evaluated in one application that provides user accounts, image validation, local inference and persistent scan history. This project therefore concentrates on integrating these components and on testing label mapping, prediction confidence, error handling and the behaviour of the local vision-language model. The effectiveness and reliability of the proposed system must be determined through implementation and evaluation.

4.2 Aim

The main aim of this project is to develop a web-based plant health detection system (LeafScan) that analyses plant leaf images and returns preliminary plant identification and health information through a structured, authenticated workflow with a stored scan history.

4.3 Objectives

The main objectives of LeafScan are:

1. To develop a web application for plant leaf image analysis.
2. To implement user registration and login functionality.
3. To prepare datasets and experiment with image-classification models for plant and disease recognition [8][17].
4. To predict plant categories and possible health conditions from leaf images.
5. To display understandable results together with general disease information.
6. To maintain a digital record of previous scans through a scan-history feature.

7. To explore a local vision-language model for additional, explanatory image analysis [5].
8. To prepare the application for portable setup and deployment after thorough testing.

4.4 Expected Outcome

The expected outcome is a web application prototype that integrates the implemented LeafScan features into a structured plant-scan workflow. Its actual reliability and effectiveness will be assessed through testing. Table 2 summarises the challenges addressed and the corresponding solutions.

Table 2: Challenges and proposed solutions in LeafScan

Challenge	**Proposed Solution**
Manual inspection is slow and subjective	Image-based AI screening of uploaded leaf images
Different diseases show similar symptoms	Trained classifiers (EfficientNet-B0, YOLO11) with confidence reporting
Limited access to agricultural experts	Browser-based tool requiring no technical knowledge
VLM explanations may be inconsistent	Optional, separate explanation module used only as supporting information
Uploaded images may raise privacy concerns	Local inference without sending images to a cloud service
Earlier results are lost	Scan history stored in SQLite and shown per user
Unauthorised access to scan features	Token-based authentication on protected endpoints

V. PROPOSED METHODOLOGY AND REQUIREMENTS

5.1 Proposed Methodology

The proposed LeafScan system follows a structured development approach consisting of requirement analysis, data preparation, model development, system design and implementation, and testing.

1. Requirement Analysis: The system requirements are identified for registration, login, image upload and validation, prediction, explanation, result display and scan history.
2. Dataset Preparation: PlantVillage and a separate rice-leaf dataset are checked, cleaned, split into training and validation sets, resized and normalised, with class distribution checked to avoid data leakage.
3. Model Development: EfficientNet-B0 [17] is trained with PyTorch [23] for plant and disease experiments, and a YOLO11 classification model [8] is trained on the rice dataset.
4. System Design and Application Development: The application is designed as a three-tier system and developed using React and Vite [19][20], FastAPI [21] and SQLite with SQLAlchemy [22].
5. Feature Integration: Classifier inference, the optional Ollama-hosted vision-language model [24] and scan-history storage are integrated into the scan endpoint.
6. Testing and Evaluation: The implemented features should be tested for functional correctness, validation of invalid inputs, label mapping, model behaviour and the complete frontend-to-backend flow. Performance results should be reported only after testing has been completed.

5.2 Functional Requirements

The main functional requirements of LeafScan are:

1. The system shall allow a new user to create an account.
2. The system shall authenticate registered users and issue a token for subsequent requests.
3. The system shall provide a dashboard from which the user can reach the scan and history features.
4. The user shall be able to select a leaf image, preview it and submit it for analysis.
5. The backend shall check the uploaded file and reject invalid, oversized or unsupported files.
6. The system shall predict a class and a confidence value using the configured classifier.
7. Where enabled, the system shall request an explanatory description from the local vision-language model.
8. The system shall present the prediction and any available disease information in an understandable form.
9. The system shall store scan records and allow the user to view previous scans.
10. Unsupported or low-confidence cases shall be reported safely rather than forcing a disease label.

5.3 Non-Functional Requirements

The proposed system should meet the following non-functional requirements:

1. Usability: The interface should be simple and easy to navigate.
2. Performance: Prediction should complete within a reasonable time on the development laptop; the vision-language step may be slower.
3. Reliability: Errors should be handled gracefully and reported with meaningful messages.
4. Security: User credentials must be stored securely and protected endpoints must require a valid token.
5. Privacy: Images should be processed locally where possible.
6. Maintainability: Frontend, backend, model and database code should be organised in separate modules.
7. Scalability: The architecture should allow additional crops, classes and models to be added.
8. Accuracy: Model outputs should be validated on labelled images; AI explanations may require independent verification [6].

5.4 Hardware and Software Requirements

Table 3 lists the hardware and software used during development and the minimum requirement suggested for running the system. Actual requirements depend on the final implementation and the models deployed.

Component	**Recorded Details / Requirement**
Development laptop	MSI GF63 Thin 11UCX
GPU	NVIDIA GeForce RTX 2050 (used for CUDA-enabled training)
Processor	Multi-core processor (CPU inference is possible but slower)
RAM	8 GB or more recommended for running the models and Ollama
Storage	Several GB for datasets, model weights and the Ollama model
Operating system	Windows 11
Python	Backend, model training and inference
FastAPI / Uvicorn	Backend API framework and server [21]
SQLite + SQLAlchemy	Local database and ORM [22]
PyTorch (CUDA)	Deep-learning framework for EfficientNet-B0 [23]
Ultralytics YOLO11	Classification training and inference for rice diseases [8]
Ollama + qwen2.5vl:3b	Local vision-language model service [5][24]
Node.js, React, Vite	Frontend development and build [19][20]
Axios	HTTP communication between frontend and backend
Web browser	Accessing and testing the application

Table 3: Hardware and software requirements

5.5 Feasibility Study

Technical feasibility: all components use open-source tools (React, FastAPI, SQLite, PyTorch, Ultralytics, Ollama), and the development laptop has a CUDA-capable GPU. Economic feasibility: no paid services are required, datasets and models are freely available and inference runs locally, so the cost is minimal. Operational feasibility: the web interface requires no technical knowledge from the user, but results must be presented as preliminary guidance. Schedule feasibility: the core interface, backend and model experiments were built within the project period, while end-to-end verification and deployment scripts remain to be completed.

VI. SYSTEM ARCHITECTURE AND DESIGN

6.1 System Architecture

LeafScan follows a three-tier design consisting of a React presentation tier, a FastAPI service tier containing the prediction services, and a SQLite data tier. The user interacts with the React/Vite frontend, which sends requests through Axios to the FastAPI backend at http://127.0.0.1:8000. The backend

validates the request, prepares the image and calls the classification service. If the vision-language workflow is enabled, it also calls the Ollama service (default port 11434). The classifier returns a class and confidence, and the backend assembles a structured response, stores the scan in SQLite through SQLAlchemy and returns the result to the frontend.

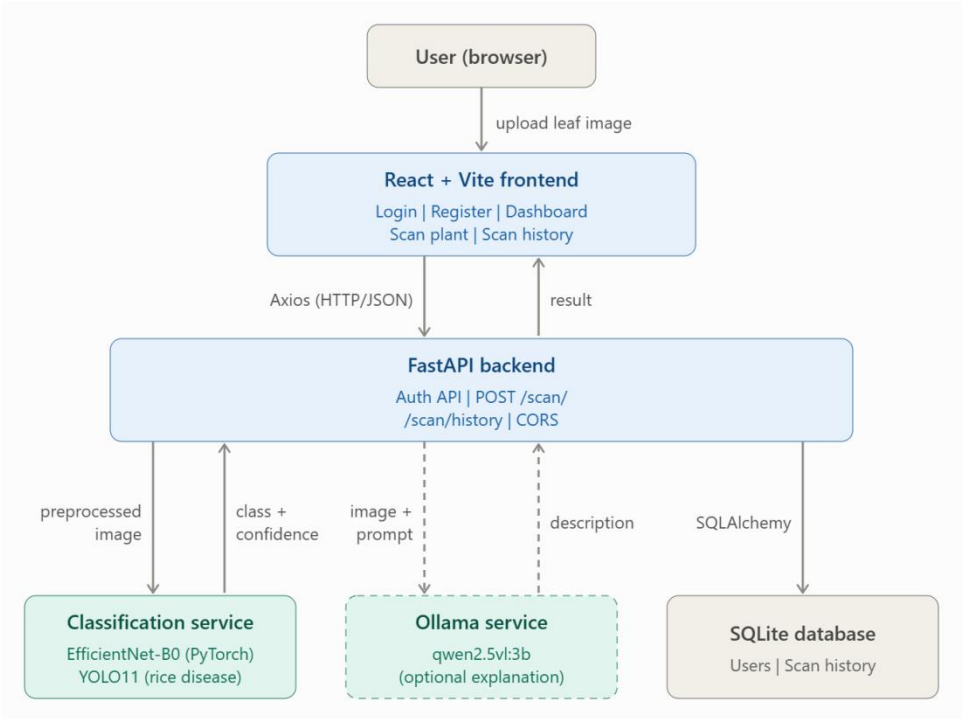


Figure 1: System Architecture of LeafScan

6.2 Main Architectural Components

1. Presentation Layer: React/Vite pages for registration, login, dashboard, image upload and preview, result display and scan history [19][20].
2. Backend Layer: FastAPI endpoints for authentication, image scanning (POST /scan/) and scan history (/scan/history), with CORS configured for the browser frontend [21].
3. Classification Layer: EfficientNet-B0 (PyTorch) and YOLO11 classification models that return a predicted class and confidence [8][17][23].
4. Explanation Layer: Optional Ollama service running qwen2.5vl:3b to produce a natural-language description of the leaf image [5][24].
5. Database Layer: SQLite accessed through SQLAlchemy, storing user profiles and scan records [22].
6. Security Layer: Token-based authentication and validation of uploaded files on protected endpoints.

6.3 Data Flow and Use Cases

A Data Flow Diagram shows how data enters the system, how it is processed and how outputs are returned. At Level 0 (Figure 2) the whole application is a single process exchanging data with the user. Level 1 (Figure 3) breaks the process into authentication, image validation and preprocessing, classification, explanation and result building, and storage of scan history. The primary actor in the use case diagram (Figure 4) is the User, who can register, log in, view the dashboard, upload a leaf image, view the prediction result, view scan history and log out.

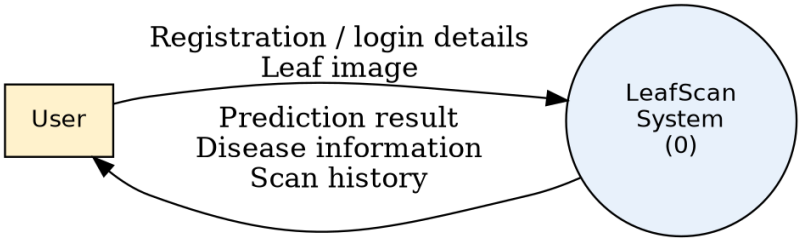


Figure 2: Data Flow Diagram (Level 0)

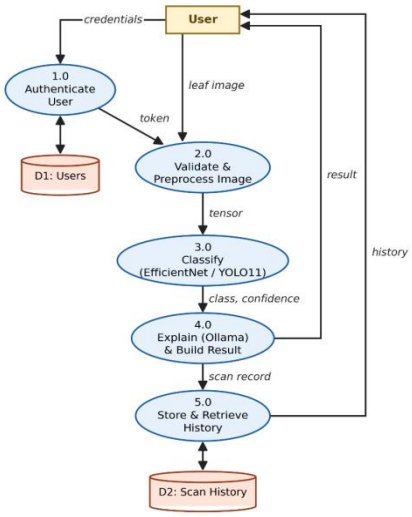


Figure 3: Data Flow Diagram (Level 1)

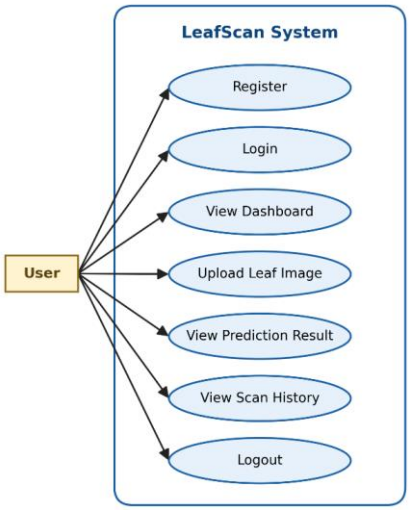


Figure 4: Use Case Diagram of LeafScan

6.4 Database Design

The database is a local SQLite file accessed through SQLAlchemy models. Two principal entities are used: Users, which stores account and profile information, and Scans, which stores each analysis performed by a user. One user can have many scans (one-to-many relationship), as shown in Figure 5. The exact schema must be confirmed against the current database before final submission.

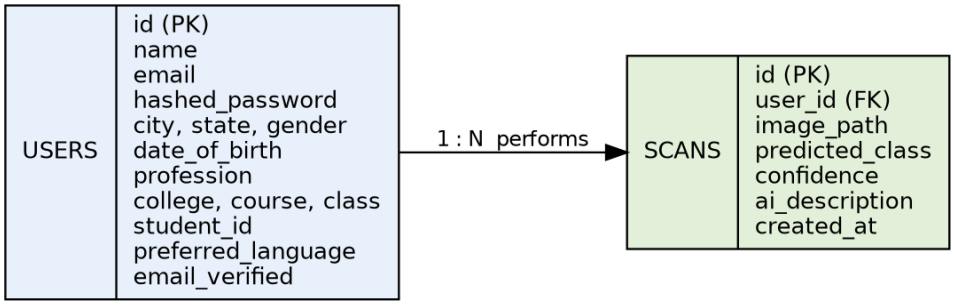


Figure 5: Entity Relationship Diagram

Table 4: Database entities and data items

Data Item	**Purpose**
User ID / Credentials	Identifies the registered user and supports authentication
Profile Fields	Name, email, city, state, gender, date of birth, profession, college, course, class, student ID, preferred language, email-verification status (as recorded)
Scan ID / User ID	Links each scan record to the user who performed it
Image Reference	Locates the uploaded leaf image (image path)
Predicted Class and Confidence	Stores the classifier output for the scan
AI Description	Stores the optional explanatory text, where implemented
Timestamp	Records when the scan was created

6.5 Sequence of a Scan Request

The sequence diagram (Figure 6) shows the message flow for one plant-scan request, including the optional call to the Ollama service and the database

write. The modular architecture allows individual components to be developed and tested separately before integration; the final capabilities must be verified against the implemented application.

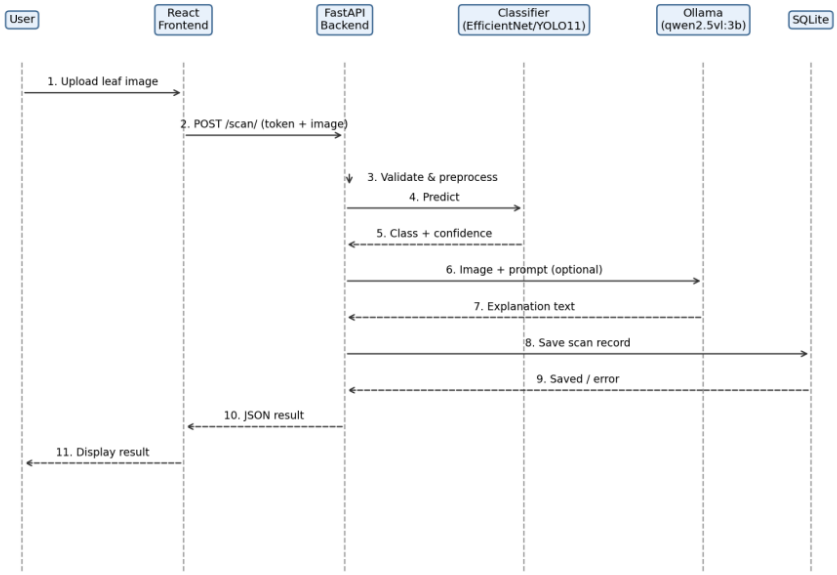


Figure 6: Sequence Diagram of a Plant Scan Request

VII. IMPLEMENTATION

LeafScan was implemented as a React and Vite frontend with Login, Register, Dashboard and Scan Plant pages; a FastAPI backend with SQLite, user authentication and a scan-history API; EfficientNet-B0 and YOLO11 classification models; a rice-disease classifier for Bacterial Blight, Blast, Brown Spot and Tungro; and an Ollama-hosted vision-language model (qwen2.5vl:3b). The implementation status of each feature must be confirmed through code inspection and testing.

7.1 Frontend Implementation

The frontend is a single-page application built with React and bundled by Vite [19][20]. The pages and components developed during the project are a splash/landing page; Login and Registration pages that collect credentials and profile details; a Dashboard from which the user navigates to scanning and history; a Scan Plant page with image selection, preview, loading state and result display; and a Scan History component listing earlier scans. Additional AI-related pages and components (AISuggest, AIChat, Projects) were created during the broader frontend work. HTTP communication is handled by Axios with the base URL http://127.0.0.1:8000, and an Axios interceptor adds the authorization token to outgoing requests when a token is available. The development server normally runs on port 5173. Routes and authentication behaviour should be re-confirmed in the final working copy.

7.2 Backend and Database Implementation

The backend is written in Python with FastAPI and served by Uvicorn on port 8000 [21]. Its main parts are authentication endpoints for registration and login, a user model with profile fields, the image scan endpoint POST /scan/, the scan-history endpoint /scan/history, a prediction service, an Ollama integration module (ollama_service.py) and CORS configuration so that the browser frontend can call the API. When an image arrives, the scan route validates it, preprocesses it for the model, obtains the class and confidence from the classifier, optionally requests an explanation from Ollama and returns a structured response. The vision-language workflow resizes and compresses the image, encodes it for the API request, sends a prompt with the image and parses the response.

SQLite is used as the local relational database and SQLAlchemy provides the object-relational mapping [22]. The user table was extended during development with profile and verification-related fields (Table 4); the current schema and any migrations must be verified before they are described as final. The scan-history feature stores each successful analysis and exposes it through /scan/history, and the Scan History component retrieves and lists these records for the logged-in user.

7.3 User Authentication

Authentication is token based. The user registers through the Register page and the backend stores the user in SQLite. At login the backend verifies the credentials and returns a token, which the frontend keeps and attaches to later requests through the Axios interceptor. Protected endpoints such as scanning and history use this token. Registration and login were implemented but must be re-tested against the current database (Section X).

7.4 Dataset Collection and Preparation

Two datasets were used (Table 5). PlantVillage was the main reference dataset for plant and disease classification [13][15][18], with approximately 54,304 images in 38 classes; the exact version and final training split should be confirmed against the files used for the final model. A separate rice-leaf dataset of 5,932 images in four classes (Bacterial Blight, Blast, Brown Spot and Tungro) [7] was prepared for the YOLO11 experiment and is reported separately from the PlantVillage experiment.

Table 5: Datasets used in the project

Dataset	Images	Classes	Used For
Rice leaf disease dataset	5,932 (4,747 train / 1,185 validation)	4	YOLO11 classification of Bacterial Blight, Blast, Brown Spot, Tungro

The data-preparation workflow was: download the dataset and verify its source and version; check image files, class folders and labels; remove unreadable or invalid images where appropriate; split the data into training and validation sets; resize images to the input size required by the model; apply normalisation and any training-time augmentation; and check the class distribution and avoid data leakage between the splits.

7.5 AI Model Development

7.5.1 EfficientNet-B0

EfficientNet-B0 is the baseline network of the EfficientNet family, which scales depth, width and resolution together [17]. EfficientNet variants have performed strongly on PlantVillage-style data [2][13][14][15]. In LeafScan it was used for plant-specific prediction experiments with PyTorch. The checkpoint referenced by the project is ai_model/models/leafscan_model.pth; it must be confirmed that it exists and matches the current class mapping before it is described as the production model.

7.5.2 YOLO11 Classification

YOLO11 [8] was explored as an additional approach and was trained on the rice dataset using the configuration in Table 6. Training logs and the final weights (runs/rice_yolo11n/weights/best.pt) should be checked before result values are quoted.

Parameter	Value
Model	YOLO11 classification (rice_yolo11n)
Training images	4,747
Validation images	1,185
Epochs	30
Image size	224
Batch size	16
Hardware	NVIDIA RTX 2050 GPU

Table 6: YOLO11 rice classification training configuration

7.5.3 Ollama Vision-Language Model

The local model qwen2.5vl:3b [5] was explored through Ollama [24]. Vision-language models can describe visible symptoms in natural language, but their disease identification is not always reliable [3][6]. In early LeafScan tests some responses were malformed or inconsistent, so the output must be validated and is presented only as a supporting explanation, never as a confirmed diagnosis.

7.6 Model Training Workflow

1. Collect and organise labelled images.
2. Validate labels and class distribution.
3. Create separate training and validation sets.
4. Configure the model, image size, batch size and number of epochs.
5. Train on the CPU or a compatible GPU (RTX 2050 in this project).
6. Evaluate on held-out validation images.
7. Review accuracy, precision, recall, F1-score and the confusion matrix.
8. Save the weights and document the class-to-index mapping.
9. Load the model in the backend and test it on known sample images.

A verified final accuracy figure was not available in the recorded project details; therefore no accuracy percentage is claimed in this paper.

7.7 Development Environment and Deployment

Table 7: Development environment

Item	Recorded Details
Operating system	Windows 11
Laptop / GPU	MSI GF63 Thin 11UCX; NVIDIA RTX 2050
Frontend	React + Vite (port 5173, typical Vite default)
Backend	FastAPI / Uvicorn (port 8000)
Database	SQLite
Deep learning	PyTorch with CUDA-enabled setup
Local vision model	Ollama qwen2.5vl:3b (API port 11434)

The project is currently run locally: the backend, frontend and Ollama are started separately on the ports above. Remote deployment and an automated startup script (for example START_LEAFSCAN.bat) for another Windows PC are planned after the end-to-end flow has been verified.

VIII. SYSTEM WORKING AND FUNCTIONAL MODULES

8.1 System Working

The LeafScan system follows these steps to analyse a leaf image:

1. Authentication: The user registers or logs in and receives a token.
2. Image Upload: The user selects a leaf image on the Scan Plant page and previews it.
3. Validation: The backend checks the file and rejects invalid, oversized or unsupported files.
4. Preprocessing and Classification: The image is preprocessed and the configured classifier returns a class and confidence.
5. Safe Handling: Unsupported or low-confidence cases are handled safely instead of forcing a disease label.
6. Explanation (optional): The local vision-language model is asked for a descriptive explanation.
7. Result Display: The frontend shows the prediction and available disease information.
8. Scan History: The scan is saved and can be reviewed later through the history view.

Actual functionality depends on implementation, model validation and the availability of the backend and Ollama services.

Table 8: LeafScan processing workflow

Stage	Action	Purpose
Stage 1	User Authentication	Registers or logs in the user and issues a token
Stage 2	Image Upload and Preview	Lets the user select and check a leaf image
Stage 3	Validation	Rejects invalid, oversized or unsupported files
Stage 4	Classification	Predicts the class and confidence using EfficientNet-B0 or YOLO11
Stage 5	Explanation (optional)	Requests a supporting description from qwen2.5vl:3b
Stage 6	Result Display	Presents the prediction and disease information
Stage 7	Scan History Storage	Saves the scan so earlier results can be reviewed

Figure 7 shows the flowchart of the system. Invalid files and low-confidence predictions are handled by dedicated branches so that the system does not force an incorrect disease label.

8.2 Functional Modules

1. Authentication Module: Handles registration, login and token issue and checking.
2. Image Upload and Validation Module: Supports image selection, preview and server-side file checks.
3. Prediction Module: Runs EfficientNet-B0 or YOLO11 inference and returns class and confidence [8][17].
4. Explanation Module: Optionally obtains a description from the local vision-language model through Ollama [5][24].
5. Result Display Module: Shows the prediction, confidence, treatment and prevention guidance.
6. Scan History Module: Stores and lists previous scans for the logged-in user.
7. Database Module: Manages user and scan records through SQLite and SQLAlchemy [22].

These modules describe the intended LeafScan functionality. Each feature must be tested before claiming successful implementation.

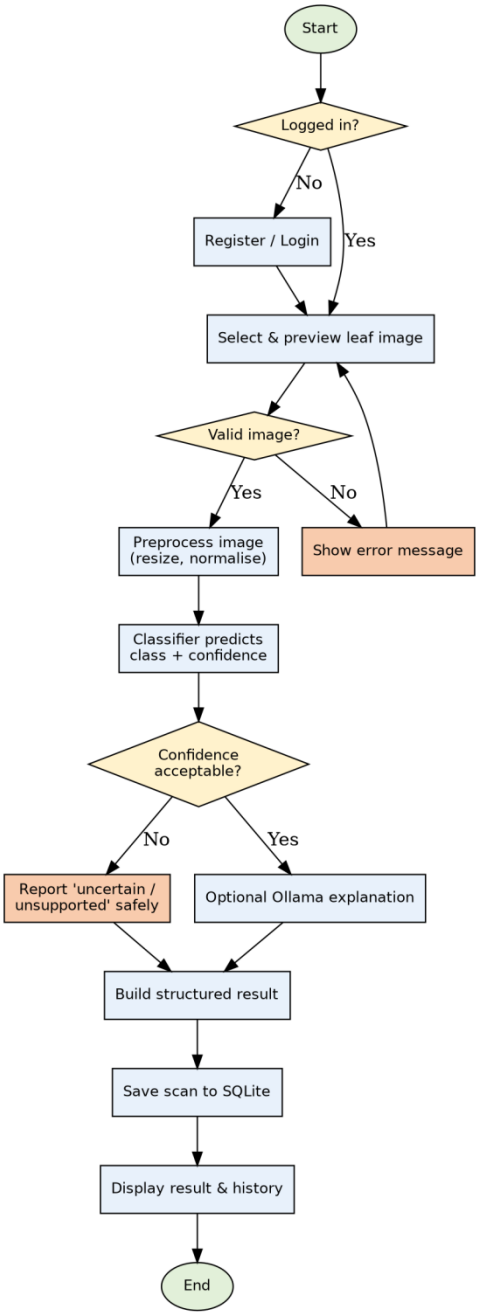


Figure 7: Flowchart of the LeafScan Process

IX. APPLICATION INTERFACE AND SCREENSHOTS

LeafScan provides a web interface for accessing registration, login, leaf-image upload, result display and scan history. The screenshots below show the main screens of the implemented application. The result screen shows an example output (Rice — Blast, reported with 100% confidence) and should not be read as evidence of model accuracy, since verified evaluation results are not yet available.

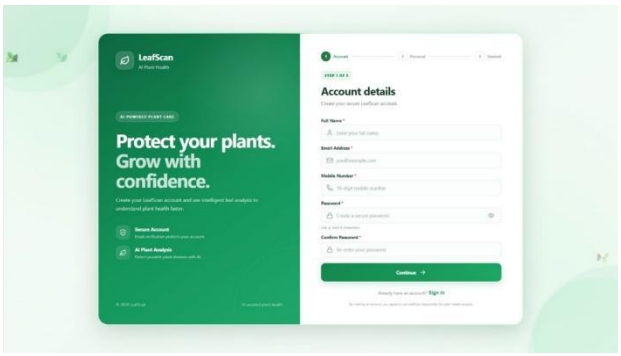


Figure 8: Registration Screen: new-user account details (Step 1 of 3).

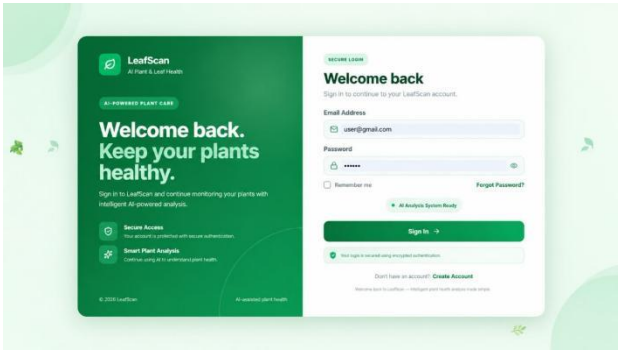


Figure 9: Login Screen: secure sign-in with email and password.

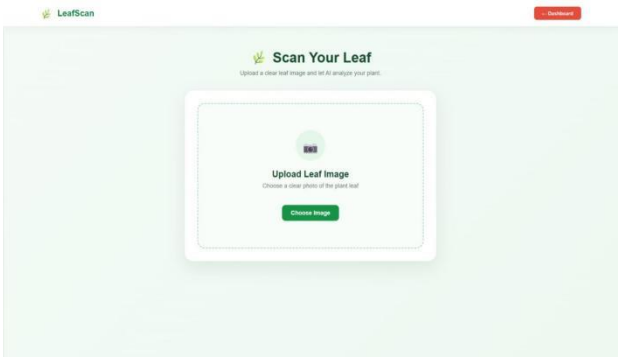


Figure 10: Scan Plant Screen: leaf-image upload area.

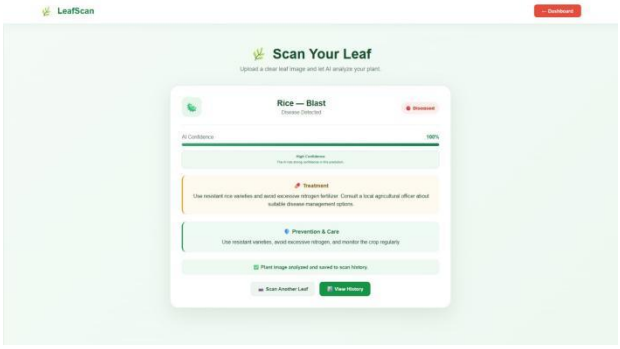


Figure 11: Scan Result Screen: prediction, confidence, treatment and prevention guidance.

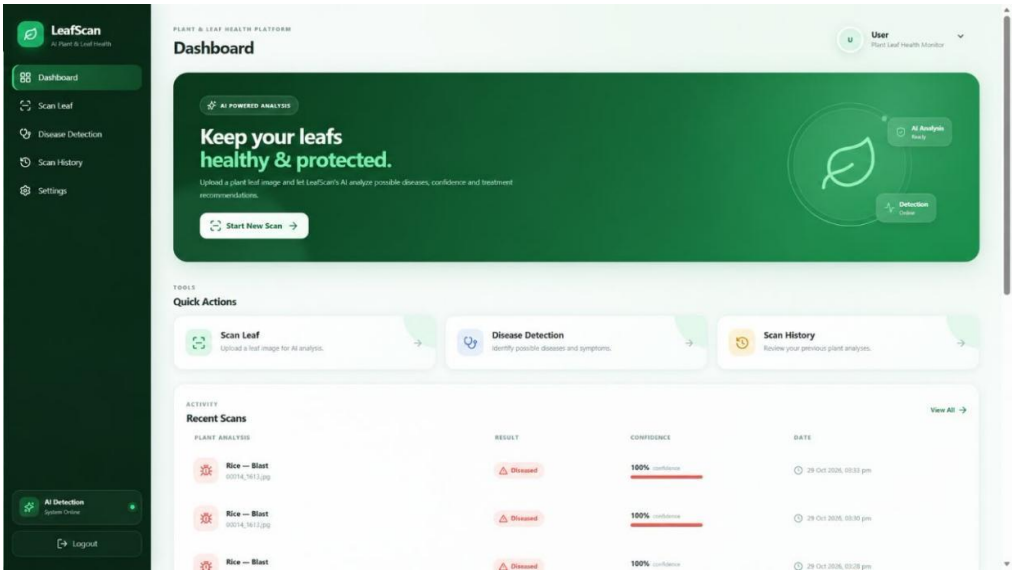


Figure 12: Dashboard Screen: quick actions and recent scans table.

Only screens and features that are available in the actual application should be included in the final paper.

X. VERIFICATION, DISCUSSION AND LIMITATIONS

10.1 Verification and Testing

LeafScan requires functional, black-box testing in which each page and API is exercised with valid and invalid inputs and the actual behaviour is compared with the expected result. Eighteen test cases have been defined: authentication (TC_AUTH_01–06: valid and duplicate registration, valid and wrong login, empty forms, access without token), upload and scan API (TC_SCAN_01–06: valid, unsupported, oversized and corrupted files, history and full flow), and model prediction and Ollama (TC_MOD_01–04 and TC_OLL_01–02: labelled samples, class mapping, rice images, low-confidence input, response parsing and service unavailability). Execution results have not yet been recorded for all test cases. Table 9 summarises the status recorded during development.

Table 9: Summary of testing status

Area	Recorded Status / Next Action
Frontend pages	Core pages developed; verify navigation in the current build
Authentication	Register/login APIs implemented; retest with the current database
Image upload	Scan endpoint implemented; test file validation and response
GPU	RTX 2050 detected in the development environment
Model predictions	Experimental; validate labels, confidence and checkpoint
Ollama	Tested; some responses malformed or inconsistent
Scan API	Runtime errors encountered; fix and retest
End-to-end flow	Requires full verification from upload to saved result

The errors recorded during development and the planned corrective actions are listed in Table 10.

Table 10: Recorded errors and planned corrective actions

Error / Issue Recorded	Planned Corrective Action
Unbound Local Error for the variable ’verified’	Initialise the variable on all code paths in the auth/scan logic
Indentation Error in app/api/scan.py	Correct indentation and re-run the server
HTTP 500 Internal Server Error on POST /scan/	Fix the errors above, add error handling and retest
Class-mapping and confidence issues in predictions	Verify class-to-index mapping against the checkpoint
Low disease confidence in a plant-specific Efficient Net prediction	Evaluate on labelled images; add confidence threshold handling

The recommended retest sequence is start the backend and confirm the root endpoint; test registration and login; test upload with valid, invalid, oversized and unsupported files; run inference on labelled samples; verify class mapping; record evaluation metrics; test scan-history persistence; and finally test the complete frontend-to-backend workflow. Actual results must be recorded after testing; successful verification must not be claimed without evidence. The evaluation results should be entered in Table 11 once available.

10.2 Discussion

LeafScan integrates authentication, image upload, classification, an optional local explanation model and scan history in one web application. Existing research shows that Efficient Net-based and transfer-learning models can classify leaf images with high reported accuracy on benchmark data [2][10][12][15], while vision-language models remain less reliable for disease identification [3][6]. The practical usefulness of LeafScan therefore depends on validating its class mapping, confidence handling and error handling, and on testing with real field photographs rather than laboratory-style images alone.

10.3 Limitations

1. Dataset dependence: Model performance depends on the quality and diversity of the training data; PlantVillage-style images may not represent field conditions [6].
2. Similar symptoms: Different diseases with similar visible symptoms can be misclassified.
3. Image quality: Lighting, background, blur and leaf condition can affect predictions.
4. Vision-language model: The local model may produce incorrect or malformed explanations.
5. Verification: Final accuracy and production readiness have not yet been established.
6. Treatment advice: Suggestions should be checked against trusted agricultural guidance.

These limitations should guide future improvements and evaluation of Leaf Scan, as summarised in Table 12.

Table 12: Limitations and future improvements

Current Limitation	**Possible Future Improvement**
Scan API runtime errors	Fix errors, add error handling and retest the full flow
Unverified class mapping and confidence	Validate mapping against checkpoints and add confidence thresholds
Inconsistent vision-language output	Validate and parse responses; consider fine-tuning on expert-verified data [3]
Limited crops and disease classes	Add more crops and classes using quality-checked datasets
Limited real-world evaluation	Evaluate on real field images and perform user trials
Local-only deployment	Remote deployment and automated setup scripts

XI. CONCLUSION AND FUTURE SCOPE

11.1 Conclusion

The proposed YOLO-based rice leaf disease detection system provides an efficient approach for identifying and classifying diseases through leaf image analysis. By utilizing deep learning and object detection techniques, the system enables rapid and reliable identification of visible disease symptoms. It can help farmers detect diseases at an early stage, supporting timely intervention and improved crop management. The integration of YOLO enhances the potential for automated disease monitoring while reducing the dependency on manual inspection. This approach can contribute to precision agriculture by promoting better crop health management and minimizing potential yield losses. Overall, the proposed system offers a promising solution for developing intelligent and efficient rice leaf disease detection systems.

11.2 Future Scope

LeafScan can be enhanced through the following improvements:

1. Add more crops and disease classes using quality-checked datasets.
2. Improve model performance and evaluate generalisation on real-world field images.
3. Implement confidence thresholds and unknown-class handling.
4. Validate generated treatment suggestions and provide trusted references; consider fine-tuning the vision-language model on expert-verified data as in [3].
5. Add regional-language output.
6. Improve scan history, downloadable reports and dashboard analytics.
7. Deploy the backend and frontend for remote access and build a mobile application.
8. Create automated setup and startup scripts for transfer to another Windows PC.

These improvements are future possibilities until implemented and tested.

XII. REFERENCES

1. A. Agrawal, A. Suri, and K. Verma, "An integrated deep learning framework for plant disease detection, severity analysis, and AI-based cure recommendation," in Proc. Conf. Bridging Engineering Disciplines with AI and Machine Learning (BEDAIML 2026), Atlantis Press, Jun. 2026, pp. 125–136, doi: 10.2991/978-94-6239-697-5_12.

2. T. Ranjan, O. Tigga, and J. Pal, "A study and analysis of plant leaf disease prediction using deep learning techniques," in Proc. Int. Conf. Emerging Trends and Technologies in Applications of Computer Science (ICETTACS 2026), Atlantis Press, Sep. 2026, pp. 121–132, doi: 10.2991/978-94-

6239-799-6_9.

3. H. Mahmood, Y. Yu, and R. Anwer, "AgriChain: Visually-grounded expert-verified reasoning for interpretable agricultural vision–language models," in Proc. 15th Language Resources and Evaluation Conf. (LREC), ELRA, 2026.
4. Md. J. Chaudhary, "An effective deep learning approach to plant disease detection," Undergraduate Conf. Intelligent Computing and Systems (UCICS 2025), Feb. 2025, doi: 10.13140/RG.2.2.18898.67522.
5. S. Bai et al., "Qwen2.5-VL technical report," arXiv:2502.13923, 2025.
6. E. Ranario et al., "Are vision-language models ready to zero-shot replace supervised classification models in agriculture?," arXiv:2512.15977, Dec. 2025.
7. P. I. Ritharson, K. Raimond, X. A. Mary, J. E. Robert, and A. J, "DeepRice: A deep learning and deep feature based classification of rice leaf disease subtypes," Artificial Intelligence in Agriculture, vol. 11, pp. 34–49, Mar. 2024.
8. G. Jocher and J. Qiu, "Ultralytics YOLO11," Ultralytics, 2024. [Online]. Available: https://github.com/ultralytics/ultralytics
9. P. Wang et al., "Qwen2-VL: Enhancing vision-language model's perception of the world at any resolution," arXiv:2409.12191, 2024.
10. M. Aggarwal, V. Khullar, N. Goyal, A. Singh, A. Tolba, E. B. Thompson, and S. Kumar, "Pre-trained deep neural network-based features selection supported machine learning for rice leaf disease classification," Agriculture, vol. 13, no. 5, p. 936, Apr. 2023, doi: 10.3390/agriculture13050936.
11. S. Mustofa, M. M. H. Munna, Y. R. Emon, and M. T. Ahad, "A comprehensive review on plant leaf disease detection using deep learning," arXiv:2308.14087, Aug. 2023.
12. K. M. Sudhesh, V. Sowmya, S. Kurian, and O. K. Sikha, "AI based rice leaf disease identification enhanced by dynamic mode decomposition," Engineering Applications of Artificial Intelligence, 2023.
13. A. Bruno et al., "Improving plant disease classification by adaptive minimal ensembling," Frontiers in Artificial Intelligence, vol. 5, Sep. 2022, doi: 10.3389/frai.2022.868926.
14. T. H. Bui, V. M. Hoang, and N. V. Nguyen, "Enhancing the performance of transferred EfficientNet models in leaf image-based plant disease classification," Journal of Plant Diseases and Protection, 2022.
15. Ü. Atila, M. Uçar, K. Akyol, and E. Uçar, "Plant leaf disease classification using EfficientNet deep learning model," Ecological Informatics, vol. 61, Art. no. 101182, Mar. 2021.
16. A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," in Proc. Int. Conf. Learning Representations (ICLR), 2021.
17. M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in Proc. 36th Int. Conf. Machine Learning (ICML), PMLR vol. 97, 2019, pp. 6105–6114.
18. E. C. Too, L. Yujian, S. Njuki, and L. Yingchun, "A comparative study of fine-tuning deep learning models for plant disease identification," Computers and Electronics in Agriculture, vol. 161, pp. 272–279, Jun. 2019.
19. Meta Open Source. (n.d.). React documentation. https://react.dev/
20. Vite. (n.d.). Vite documentation. https://vite.dev/guide/
21. FastAPI. (n.d.). FastAPI documentation. https://fastapi.tiangolo.com/
22. SQLAlchemy. (n.d.). SQLAlchemy documentation. https://docs.sqlalchemy.org/
23. PyTorch. (n.d.). PyTorch documentation. https://pytorch.org/docs/stable/index.html
24. Ollama. (n.d.). Ollama – run large language and vision-language models locally. https://github.com/ollama/ollama