Software Design Specification

Project Title:

Plant Disease Detection App (Agro Guard App)

Project Code:

PDDA - 041245

Internal Advisor:

Dr. Hussam Ali

External Advisor:

N/A

Project Manager:

Prof. Dr. Muhammad Iliyas

Project Team:

M Taha Mukhtar BSCS51F22S004 (Team Lead)
Hammad Ahmad BSCS51F22S012 (Team Member)
Usama Farooq BSCS51F22S045 (Team Member)

Submission Date:

December 15, 2025

Project Manager's Signature

Document Information

Category	Information
Customer	UOS – Department of Computer Science
Project	Plant Disease Detection App (Agro Guard App)
Document	Software Design Specification
Document Version	2.0
Identifier	PDD - 041245
Status	Approved
Author(s)	Muhammad Taha Mukhtar Hammad Ahmad Usama Farooq
Approver(s)	PM
Issue Date	October 20, 2025
Document Location	
Distribution	1. Advisor 2. PM 3. Project Office

Definition of Terms, Acronyms and Abbreviations.

Term	Description
AI	Artificial Intelligence
GPS	Global Positioning System
ML	Machine Learning
API	Application Programming Interface
MVVM	Model View View Model

Table of Contents

1. Introduction

1.1 Purpose of Document

This document provides a comprehensive design specification for the AI Plant Disease Detection mobile application. It is intended for the development team, project supervisors, and technical reviewers who will evaluate the system architecture and implementation approach. This document will serve as a blueprint for the development process, outlining the system architecture, design strategies, and technical considerations.

1.2 Project Overview

The AI Plant Disease Detection App is a mobile application designed to assist farmers in identifying plant diseases through image recognition technology. The application leverages artificial intelligence and machine learning to analyze photographs of crops and provide instant diagnosis of plant health conditions.

The system supports disease detection for multiple crop types including corn, oranges, potato, tomato, apple. Upon capturing or uploading an image of a plant, the application processes the image through a trained machine learning model to detect diseases. The application then provides farmers with detailed information about the identified disease, including descriptions and recommended treatments. Additionally, the app integrates weather information relevant to the farmer's location, helping them make informed decisions about crop management.

1.3 Scope

The system will:

- Provide a mobile application interface for capturing or uploading plant images
- Support disease detection for different crop plants such as corn, orange, tomato, potato.
- Utilize a trained machine learning model to classify plant diseases
- Provide detailed descriptions of identified diseases
- Suggest practical treatment options for detected plant diseases
- Display real-time weather information based on the user's geographical location

- Integrates an AI based chatbot to respond to user questions regarding crop disease and agriculture guidance

The system will NOT:

- Provide a marketplace for buying/selling agricultural products
- Support real-time chat or community features
- Provide financial or subsidy-related services
- Include e-commerce or payment gateway integration
- Offer personalized recommendations based on user history

2. Design Considerations

2.1 Assumptions and Dependencies

Assumptions:

1. **Device Capabilities:** Target users have smartphones with functional cameras capable of capturing clear images (minimum 8MP resolution recommended)
2. **Operating System:** The application will run on Android devices with minimum Android 8.0 OS version.
3. **Image Quality:** Users will capture images in adequate lighting conditions with the affected plant part clearly visible
4. **Storage Space:** Devices have sufficient storage space (minimum 100MB) to accommodate the application and ML model

Dependencies:

1. **Machine Learning Framework:** The application depends on TensorFlow Lite for on-device model inference
2. **Weather API Service:** Integration with a weather data provider for real-time weather information
3. **Location Services:** Device GPS/location services must be enabled for weather data retrieval
4. **Image Processing Libraries:** Dependencies on image preprocessing libraries for standardizing input to the ML model
5. **Dataset Availability:** Access to labeled plant disease datasets for model training and validation

2.2 Risks and Volatile Areas

Technical Risks:

1. **Model Performance Variability:** Disease detection accuracy may vary significantly based on image quality, lighting conditions, and disease progression stages. Images captured in poor lighting or at incorrect angles may lead to misclassification.
    - **Mitigation***:* Implement image quality checks, provide user guidelines for capturing images, and display confidence scores with results.

2. **Model Size and Performance:** Large ML models may cause slow inference times on older devices or consume excessive storage space.
    - **Mitigation:** Optimize model using quantization techniques, implement model compression, and test on various device specifications.

3. **API Dependency:** Weather API services may experience downtime, rate limiting, or changes in pricing/terms of service.
    - **Mitigation:** Implement graceful error handling, cache weather data, and design the system to allow easy switching between weather service providers.

Requirement-Related Risks:

1. **Scope Creep:** Stakeholders or users may request additional features (e.g., user accounts, social features, more crop types) during development.
- **Mitigation:** Maintain clear scope documentation and implement a modular architecture that allows future feature additions without major refactoring

2. **Remedy Information Accuracy:** Agricultural recommendations must be accurate and appropriate for local conditions; incorrect remedies could harm crops or be ineffective.
- **Mitigation:** Consult agricultural experts, validate remedy information with authoritative sources, and include disclaimers

Volatile Areas (Subject to Change):

1. **Supported Crop Types:** Initial scope includes six crops, but may expand based on stakeholder feedback or model availability
2. **Weather Information Display:** The extent and format of weather data presentation may evolve based on user feedback
3. **Disease Database:** Disease descriptions and remedies may require updates as new agricultural research becomes available
4. **ML Model Updates:** The disease detection model may need periodic retraining with new data, requiring a model update mechanism
5. **User Interface Design:** UI/UX may undergo iterations based on usability testing with actual farmers

Contingency Plans:

- Design modular architecture to accommodate new features without major rewrites
- Implement version control for ML models to allow rollback if new versions perform poorly
- Document all external dependencies and maintain alternative options where feasible

3. System Architecture

3.1 System Level Architecture

Agro Guard is a native Android mobile application built with **Kotlin** and **Jetpack Compose** that helps farmers identify plant diseases through AI-powered image analysis.
The system uses a **Clean Architecture** pattern with **MVVM (Model-View-ViewModel)**, offline-first capabilities using **Room Database**, and cloud integration via **Firebase**.

Core Components:

- **Android Application Layer:** UI (Jetpack Compose), Presentation (ViewModel/StateFlow), Domain (Use Cases), Data (Repositories).
- **Backend Services:** Firebase (Auth, Firestore, Storage), External ML API (Python/FastAPI), Chat Service (LLM Integration).
- **Cloud Services:** Firebase Platform (Auth, Firestore, Cloud Functions, Storage).
- **Local Storage:** DataStore (User preferences), Room Database (Cached detection history, plant models).

Key Subsystems:

- **Authentication System:** Firebase Authentication (Email/Password, Google Sign-In).
- **UI Subsystem:** Jetpack Compose (Declarative UI), Navigation Compose.
- **Image Processing:** CameraX (Image capture), Coil (Image loading), Coroutines (Async processing).
- **ML Inference:** Disease detection via external REST API (Retrofit) or local TFLite (optional).
- **Data Management:** Repository pattern mediating between Firebase and Room.

High-Level Architecture Diagram

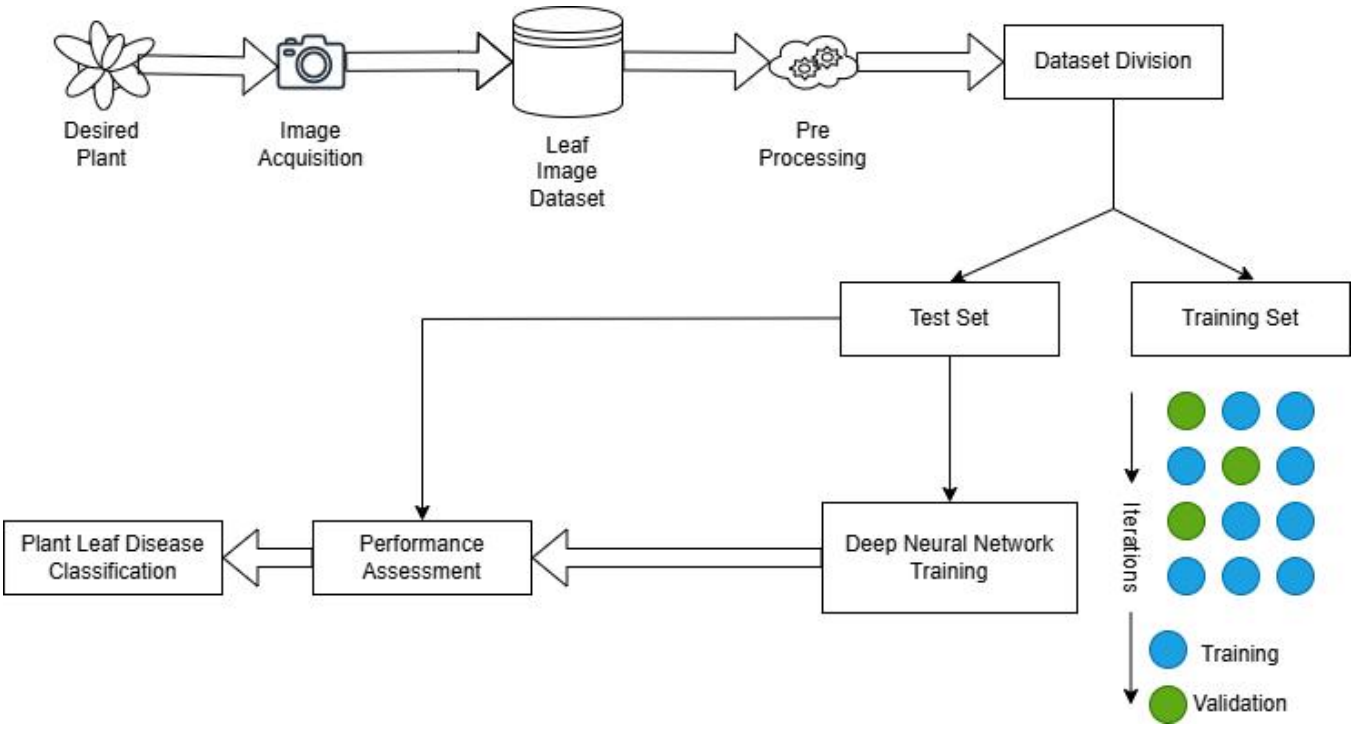


Figure 3.1

3.2 Sub-System / Component / Module Level Architecture

This section breaks down the major components of the Agro Guard system and their interactions using modern Android standards.
Mobile Application Components The Android application follows **Google's Guide to App Architecture**:

1. **UI Layer (Jetpack Compose):**
    - LoginScreen, SignupScreen, HomeScreen
    - CameraScreen (wrapping CameraX), GalleryScreen
    - ResultScreen, ChatScreen
2. **Presentation Layer (ViewModel):**
    - AuthViewModel: Manages login state (Loading, Success, Error).
    - DetectionViewModel: Handles image upload and result processing.
    - ChatViewModel: Manages conversation state.
3. **Domain Layer (Optional but recommended)**

- Use Cases: AuthenticateUserUseCase, DetectDiseaseUseCase, SyncDataUseCase.

4. **Data Layer (Repositories):**
    - AuthRepository: Wraps FirebaseAuth.
    - DiseaseRepository: Coordinates between Retrofit (Remote) and Room (Local).
    - ChatRepository: Manages chat history in Firestore.

Component Interaction Diagram (MVVM)

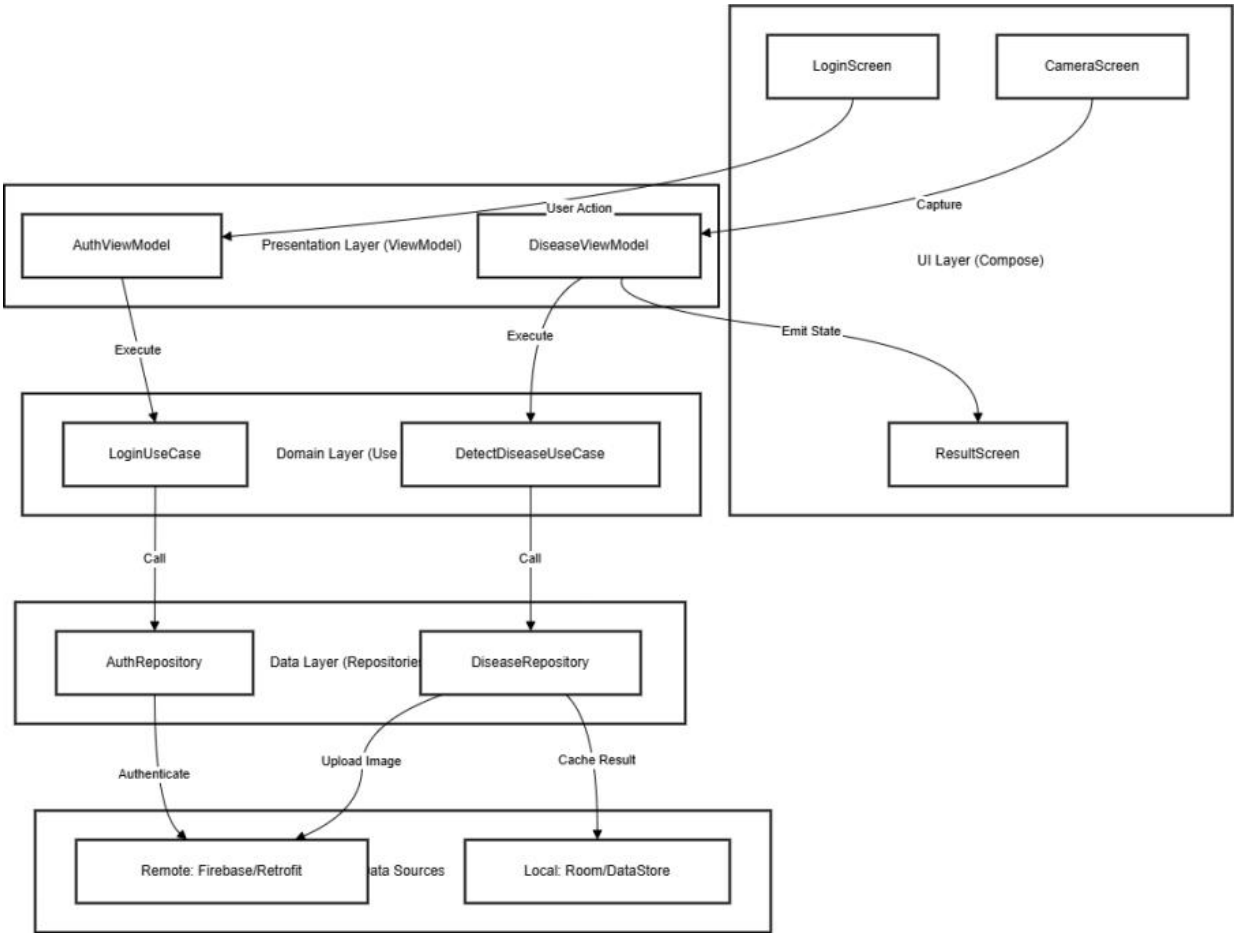


Figure 3.2

3.3 Sub-Component / Sub-Module Level Architecture (1…n)

3.3.1 Authentication System (Firebase)

The system uses **Firebase Authentication** for secure identity management.

- **Registration:** Email/Password validation handled via Kotlin Patterns.EMAIL_ADDRESS.

- **Session:** Managed automatically by the Firebase SDK. FirebaseAuth.getInstance().currentUser persists across app restarts.
- **Token Management:** Firebase handles token refresh automatically.
- **Data Integration:** Upon registration, a corresponding user document is created in the users collection in Firestore.

3.3.2 ML Inference Engine (Hybrid)

- **Cloud Inference (Primary):** Images are uploaded to Firebase Storage. The download URL is sent via **Retrofit** to the Python Backend.
- **Local Inference (Future/Optional):** The architecture supports integrating **TensorFlow Lite (TFLite)** directly into the Android app for offline inference using the org.tensorflow:tensorflow-lite library.

3.3.3 API Integration Manager (Retrofit)

All non-Firebase network calls (specifically to the custom ML backend) are handled by **Retrofit**.

- **Network Module (Hilt):** Provides a singleton instance of OkHttpClient and Retrofit.
- **Interceptors:** Adds auth tokens to headers if the custom backend requires them.
- **Error Handling:** try-catch blocks within Coroutines handle IOExceptions and HttpExceptions.

3.3.4 Image Processing Pipeline (Kotlin/Coroutines)

1. **Capture:** Uses **CameraX** API for lifecycle-aware camera control.
2. **Selection:** Uses ActivityResultContracts.PickVisualMedia for secure gallery access.
3. **Compression:** Images are compressed to JPEG (quality 80) using standard Android Bitmap APIs before upload.
4. **Asynchronous:** All image operations run on Dispatchers.IO to prevent UI freezing.

3.3.5 Database Schema (Firestore)

Firestore is a NoSQL document database. The schema is optimized for reads.

Collections:

- **users (Collection)**
    - Document ID: uid (from Auth)

    - Fields: email, name, createdAt, preferences { notifications: bool }

- **detections (Collection)**
    - Document ID: Auto-generated
    - Fields: userId (index), imageUrl, diseaseName, confidence, timestamp, plantType.
- **chats (Collection)**
    - Document ID: Auto-generated
    - Fields: userId, message, sender ("user" or "bot"), timestamp.

4. Design Strategies

4.1 Strategy 1…n

4.1.1 Authentication Strategy

- **Implementation:** Firebase Authentication.
- **Why:** Reduces backend boilerplate, provides free tier SMS/Email auth, and integrates seamlessly with Firestore security rules.

4.1.2 Data Sync Strategy (Offline-First)

- **Tool:** Firebase Firestore Offline Persistence.
- **Behavior:** The app writes to the local cache first. When the device reconnects to the internet, the Firestore SDK automatically synchronizes pending writes to the cloud. This requires zero extra code for basic offline support.

4.1.3 Clean Architecture with Hilt & MVVM

- **Objective:** Separation of concerns and testability.
- **DI: Hilt (Dagger)** is used for dependency injection (injecting Repositories into ViewModels, APIs into Repositories).
- **Async: Kotlin Coroutines** and **Flow** replace Promises and RxJava for handling asynchronous streams of data.

4.1.4 Migration to Native Android (Kotlin)

- **Performance:** Native code runs faster than the JS bridge in React Native, especially for image processing and heavy lists.
- **Camera:** CameraX offers more stable and device-agnostic control over hardware than React Native camera libraries.
- **UI:** Jetpack Compose offers a modern, declarative UI toolkit similar to React but type-safe and compiled.

4.1.5 Deployment Architecture

The Android app is distributed via Google Play Console. The backend is split between Firebase (Serverless) and a GPU Instance (ML).

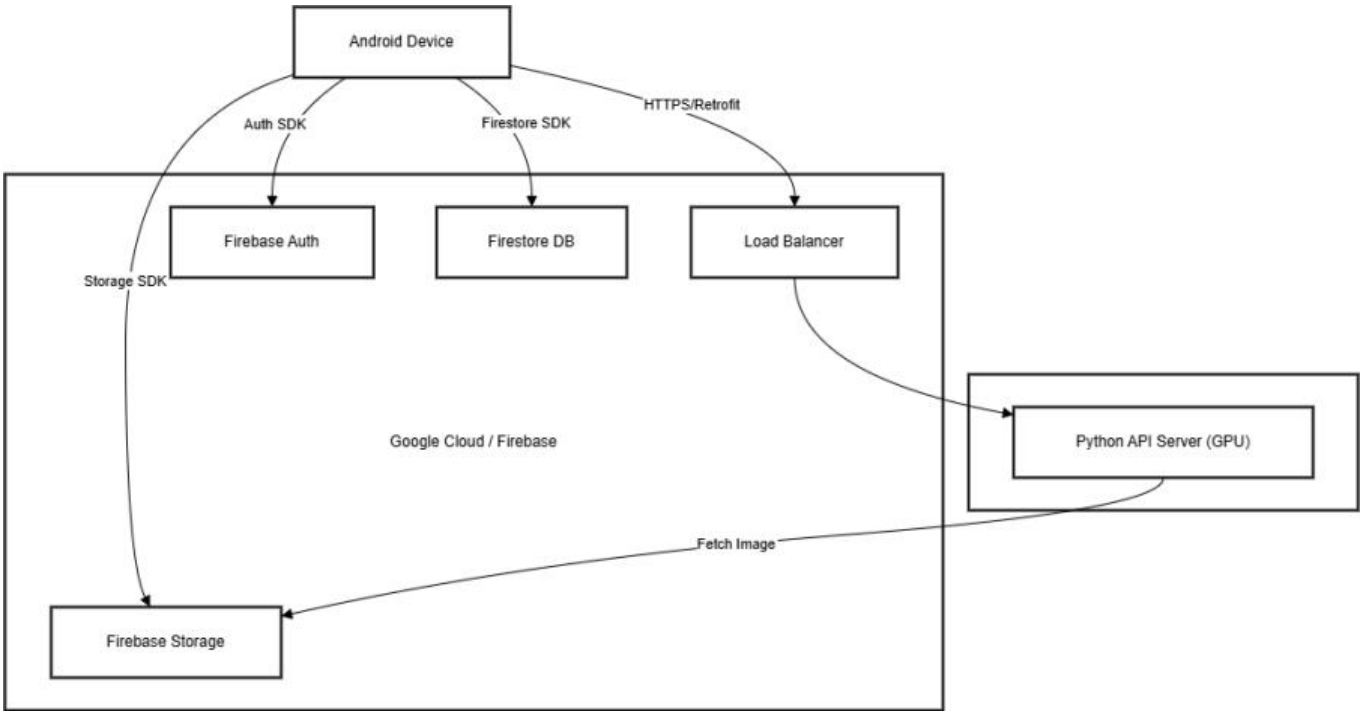


Figure 4.1.5

5. Detailed System Design

The detailed system design provides a comprehensive architectural blueprint of the AI Assistant for Plant Disease Diagnosis, presenting multiple design perspectives that define system structure, behavior, and data management.

Class Diagram: The system defines classes across multiple modules including User Management, Disease Detection, ChatBot, Weather Services, and Location. Each class specifies relevant attributes and methods, while UML relationships illustrate interactions and dependencies between system components.

Sequence Diagram: The sequence diagrams illustrate six core system workflows: Registration, Login, Weather Display, Disease Detection, ChatBot Interaction, and Logout. Each diagram presents step-by-step message flows, including parameter details exchanged between the interacting system components.

State Transition Diagram: The state transition diagram maps the application states and user navigation flow from launch through various features, highlighting decision points, loop-back states, and error-handling transitions.

Logical Data Model: The logical data model presents the conceptual data structure of the system, identifying key entities such as User, Disease, Treatment, and DetectionResult, along with the relationships among them.

Physical Data Model: The physical data model defines the actual database implementation using Android SQLite, specifying tables, columns, data types, primary and foreign keys, and integrity constraints.

Detailed GUIs: The detailed GUI designs provide visual representations of all major application screens, ensuring a consistent, intuitive, and user-friendly experience.

5.1 Class Diagram:

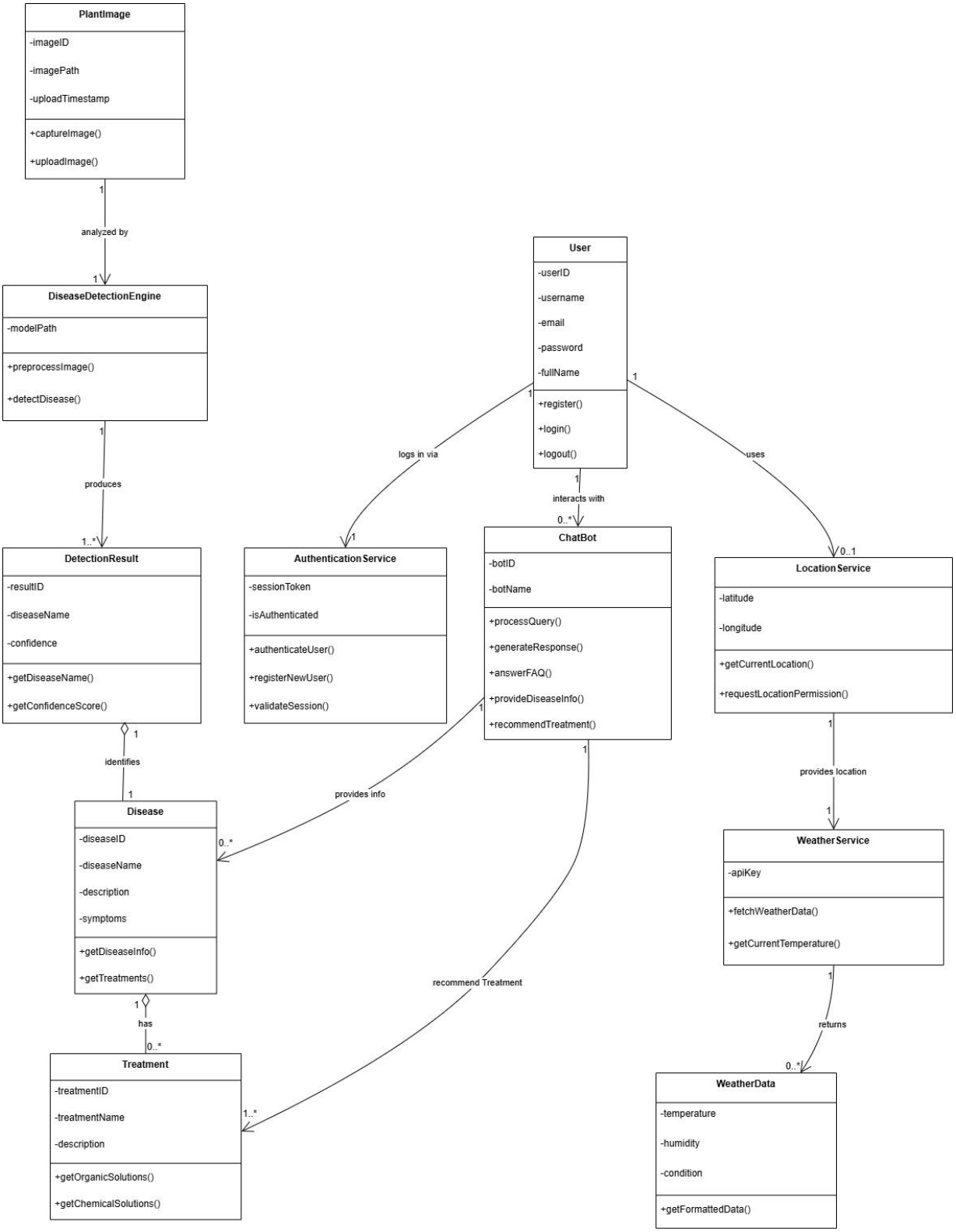


Figure 5.1

5.2 Sequence Diagram:

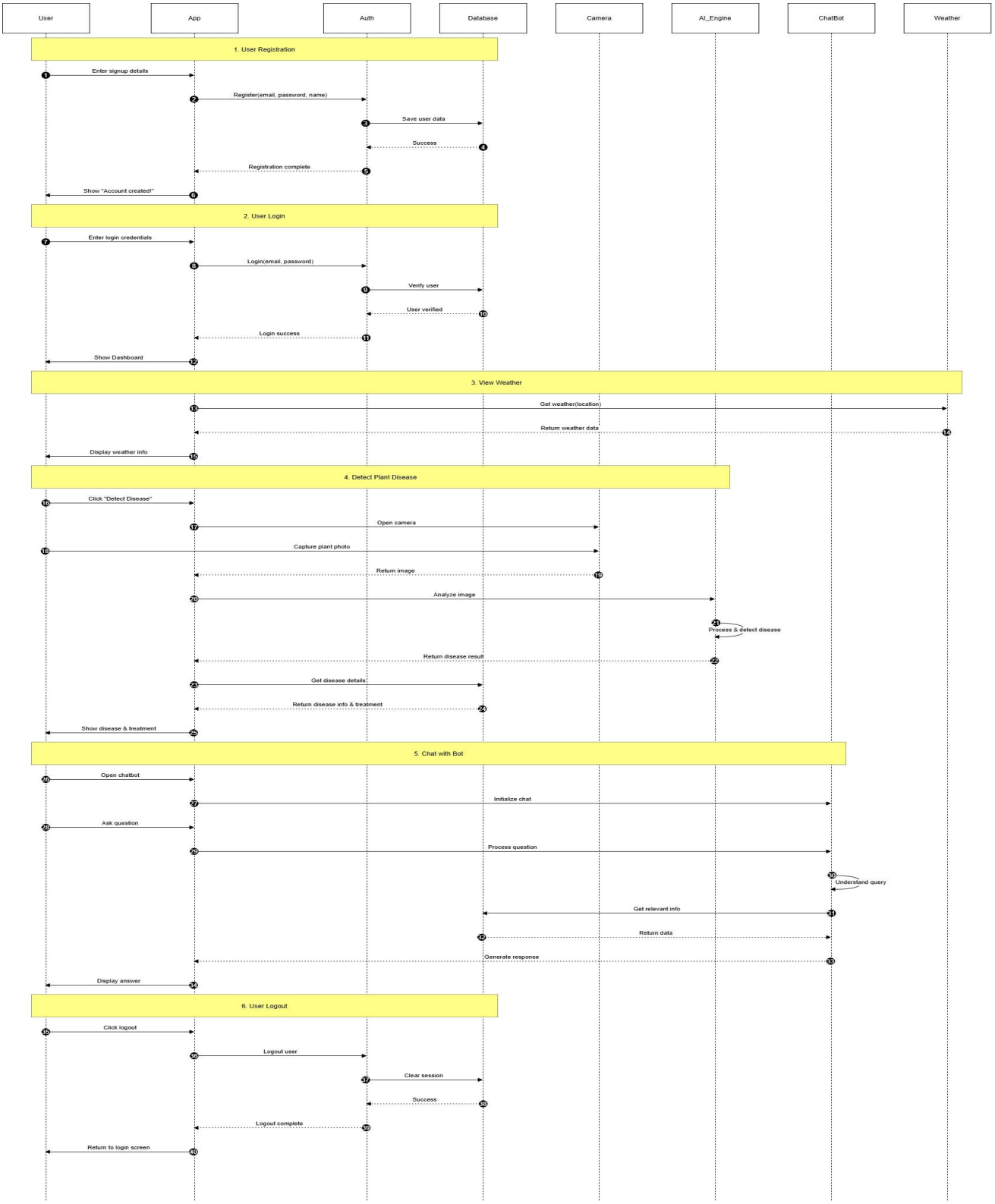


Figure 5.2

5.3 State Transition Diagram:

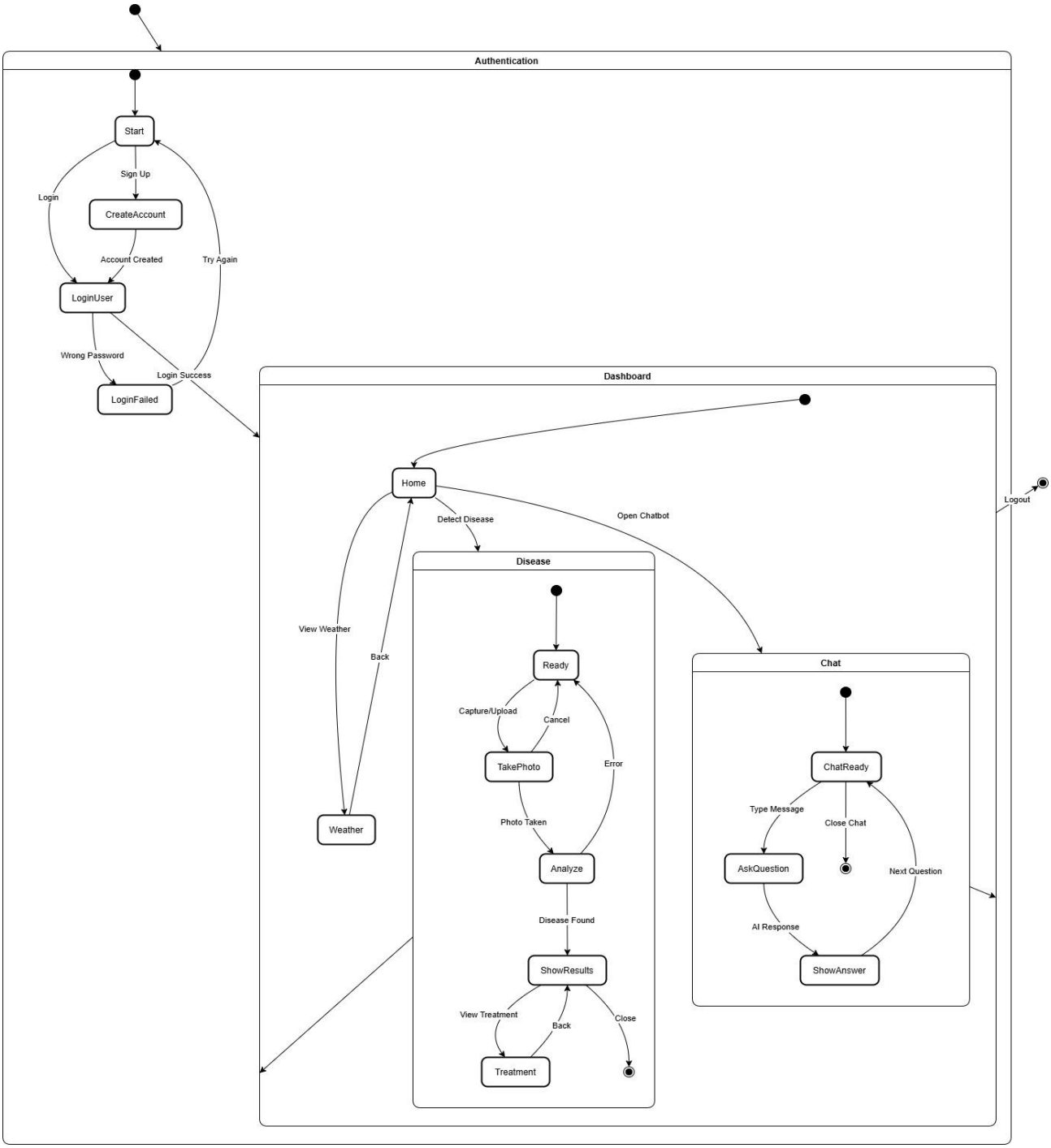
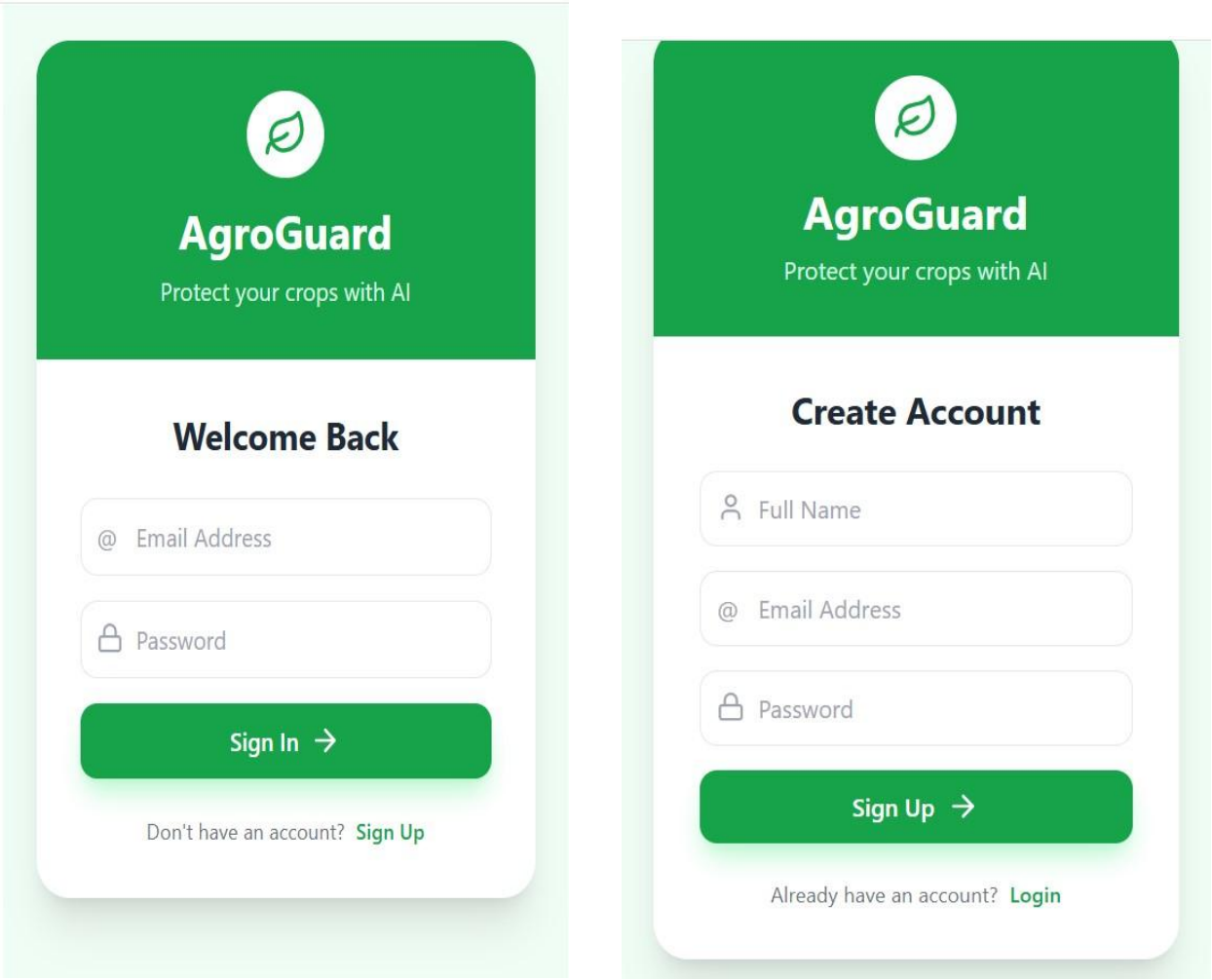


Figure 5.3

5.4 E/R Diagram:

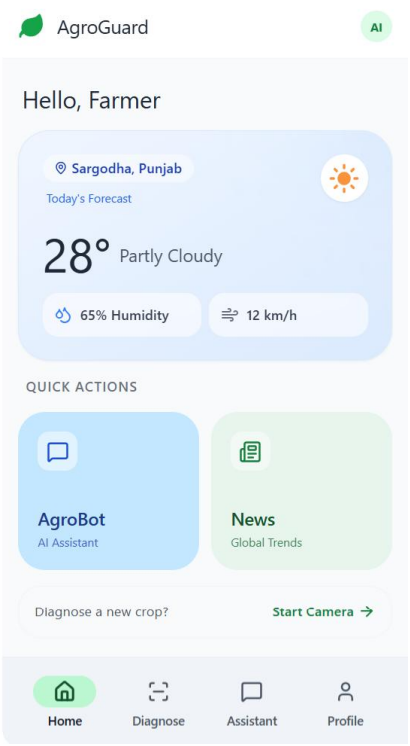
Figure 5.4

5.5 Graphical User Interface:

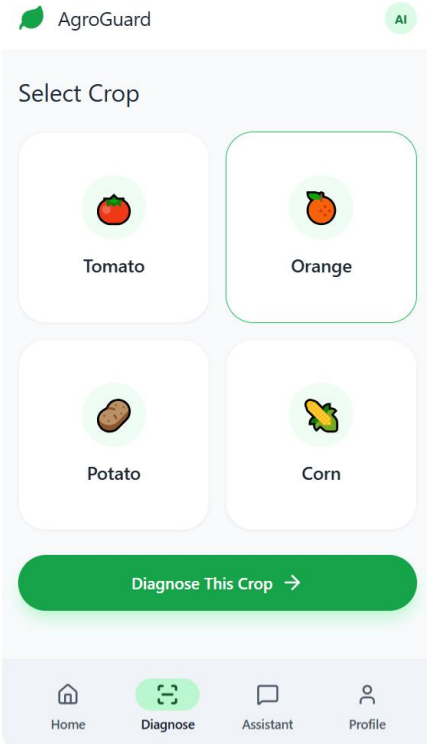


(a) Login Screen (b) Account Creation Screen

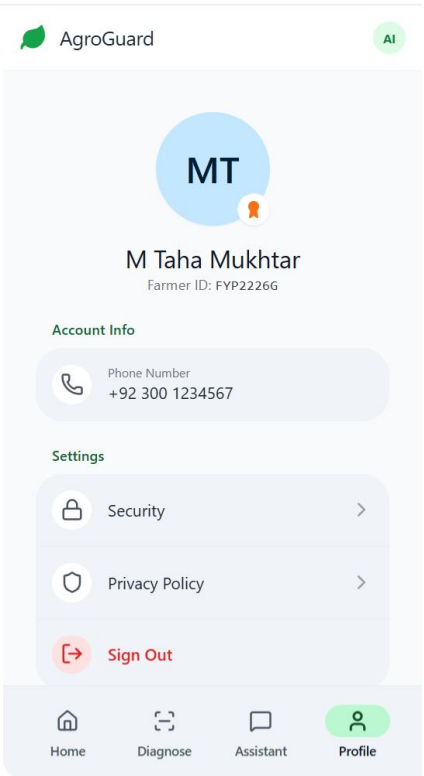
Authentication Interface



(a) Home Screen



(b) Dignose Screen



(c) Profile Screen

6. References

Ref. No.	Document Title	Date of Release/ Publication	Document Source
FYP2226G	Project Proposal	Oct 10, 2025	https://github.com/mtahamukhtar2-ui/FYP/blob/3dbcf4303c0d487f63b785c9511108fd356f04bb/FYP%20Proposal.pdf
FYP2226G	SRS	Oct 20, 2025	https://github.com/mtahamukhtar2-ui/FYP/blob/3dbcf4303c0d487f63b785c9511108fd356f04bb/SRS%20V%203.0.pdf
PDDA - 041245	Diagrams	Dec 15,2025	https://github.com/mtahamukhtar3-star/FInal-Diagrams.git