

## **Agri Management System: Crop Disease Detection and Prediction**

### **EXPERIMENT 1 :**

#### **Aim :-**

To list down the following for the chosen problem statement:

- Functional requirements
- Non-functional requirements
- Security features
- Technology Stack

#### **Problem Statement :-**

Agriculture is the backbone of many economies, particularly in developing nations like India, where a significant portion of the population depends on farming for livelihood. However, farmers often face challenges such as crop diseases, fluctuating market prices, and lack of localized farming recommendations. Traditional methods of detecting crop diseases are time-consuming, require expert intervention, and are not always accessible to small-scale farmers.

This project aims to develop an integrated web-based Agri Management System that leverages machine learning and real-time data APIs to:

- Detect crop diseases from images using deep learning models.
- Recommend suitable remedies.
- Visualize real-time market prices of agricultural products.
- Suggest the most appropriate crops for cultivation based on location, weather, and soil conditions.

## **Functional Requirements**

### 1. User Authentication and Authorization

- Users can register and log in as farmers, agronomists, or administrators.
- Role-based access to features (e.g., only admins can manage data sources).

### 2. Crop Disease Detection

- Users can upload images of crop leaves.
- System uses trained CNN model (via TensorFlow/Keras) to detect crop diseases.
- Displays disease name and probability of detection.

### 3. Remedy Suggestion

- Based on detected disease, the system provides suitable remedies.
- Suggestions include pesticides, organic treatments, and best practices.

### 4. Product Price Visualization

- Integrates with India Gov API to fetch real-time agricultural product prices.
- Visualizes prices using charts for better understanding (using JS libraries).
- Filters available based on location and crop type.

### 5. Crop Recommendation System

- Uses KNN model to recommend the most suitable crops.
- Factors include user location, weather (via OpenWeatherMap API), and soil conditions (via India Gov API).
- Provides rationale for the recommendation.

### 6. Interactive Location Mapping

- Uses Leaflet API to show farms, weather patterns, and soil data on a map.
- Helps visualize crop suitability geographically.

## 7. Responsive User Interface

- Built with HTML, CSS, and JavaScript for mobile and desktop compatibility.
- Dashboard view with access to all major functionalities.

## **Non-Functional Requirements**

### 1. Performance Requirements

- The system should respond to image uploads and return disease predictions within 5–10 seconds.
- Crop recommendation and product price visualization should load within 3 seconds under normal conditions.

### 2. Scalability

- The system should support multiple users concurrently, including simultaneous image uploads and API calls.
- Backend architecture should allow future scaling (e.g., more ML models, more regions/crops).

### 3. Availability

- The system should maintain 99% uptime during active agricultural seasons.
- Weather and price APIs should have fallback handling for temporary outages.

### 4. Reliability

- System must provide consistent outputs for the same input (e.g., same image → same disease result).
- All key operations should be logged and recoverable in case of failure.

## 5. Usability

- The system should be easy to navigate, especially for farmers with minimal tech experience.
- Interfaces should use intuitive language, icons, and error-free feedback.

## 6. Security

- Secure login system with password encryption.
- Input validations and backend protection against common attacks (SQL injection, XSS).
- Access control based on user roles (farmer, admin, agronomist).

## 7. Maintainability

- Code should be modular and well-documented to allow easy updates.
- Configuration files should separate API keys and database connections for easier maintenance.

## 8. Portability

- Web application should be compatible with all modern browsers (Chrome, Firefox, Edge, Safari).
- Interface should be responsive on both mobile and desktop devices.

## 9. Interoperability

System should integrate seamlessly with third-party APIs like:

- OpenWeatherMap API
- India Gov Price API
- Leaflet Maps API

## **Security Concerns**

### 1. User Authentication and Data Protection

Concern: Unauthorized access to user accounts (farmers/admins).

Mitigation:

- Use hashed and salted passwords (e.g., Django's default `PBKDF2` hashing).
- Secure login sessions using HTTPS.
- Implement session timeout and logout functionality.

### 2. Input Validation and Sanitization

Concern: Vulnerability to SQL Injection, XSS, and CSRF attacks.

Mitigation:

- Use Django's built-in ORM to avoid raw SQL queries.
- Auto-escape HTML content in templates.
- Enable Django's CSRF protection middleware.

### 3. API Key Exposure

Concern: Exposure of sensitive API keys (OpenWeatherMap, India Gov API, etc.) in frontend code.

Mitigation:

- Store API keys in environment variables.
- Use a backend service to fetch external API data, never expose keys on the client-side.

### 4. Model & Image Upload Security

Concern: Malicious files being uploaded for disease detection.

Mitigation:

- Limit accepted file types (e.g., JPEG, PNG).
- Use virus/malware scanners on uploaded files.
- Restrict upload size and apply image format validation.

## 5. Data Privacy and Access Control

Concern: Leakage of sensitive agricultural data and personal user data.

Mitigation:

- Role-based access control (RBAC) to ensure data visibility is limited by user roles.
- Limit administrative privileges and protect admin routes.

## 6. Man-in-the-Middle Attacks

Concern: Data interception over insecure connections.

Mitigation:

- Enforce HTTPS across the application using SSL/TLS certificates.

## 7. Error Handling and Logging

Concern: Detailed error messages revealing system structure or code logic.

Mitigation:

- Disable debug mode in production.
- Log errors securely without exposing stack traces to users.

## 8. Third-Party API Reliability & Security

Concern: Dependency on APIs which could be insecure or unstable.

Mitigation:

- Validate and sanitize all API responses.
- Add retry/fallback logic for failed API calls.

## **Technology Stack:**

### **Frontend:**

The frontend of the Agri Management System is developed using HTML5, CSS3, and JavaScript, ensuring a responsive and intuitive user interface. It provides a clean dashboard and easy navigation for farmers, administrators, and experts.

To enhance interactivity and visualization, Leaflet.js is integrated to display maps showing user locations, crop zones, and weather data dynamically.

### **Backend:**

The backend is powered by Django, a high-level Python web framework that handles routing, form validation, user authentication, and server-side logic.

Django communicates with a relational database such as SQLite, PostgreSQL, or MySQL to manage and store data related to users, crops, diseases, remedies, and feedback.

APIs are integrated into the backend to fetch live weather and price data securely.

### **Machine Learning:**

The system utilizes TensorFlow and Keras for implementing machine learning models.

A Convolutional Neural Network (CNN) is used to detect crop diseases from uploaded leaf images with high accuracy.

A K-Nearest Neighbors (KNN) algorithm is employed to recommend the most suitable crops for cultivation based on parameters like user location, weather data, and soil conditions.

### **APIs and External Services:**

- OpenWeatherMap API: Provides real-time weather data for different locations.
- India Government API : Supplies up-to-date agricultural product price information.
- Leaflet API: Enables map-based visualizations of crop zones and climate overlays.

### **Development Tools:**

- VS Code : For writing and managing the codebase.
- Git & GitHub: For version control and collaborative development.
- Postman: For testing API integrations.
- Virtualenv: For managing Python environments and dependencies.



## EXPERIMENT 2:

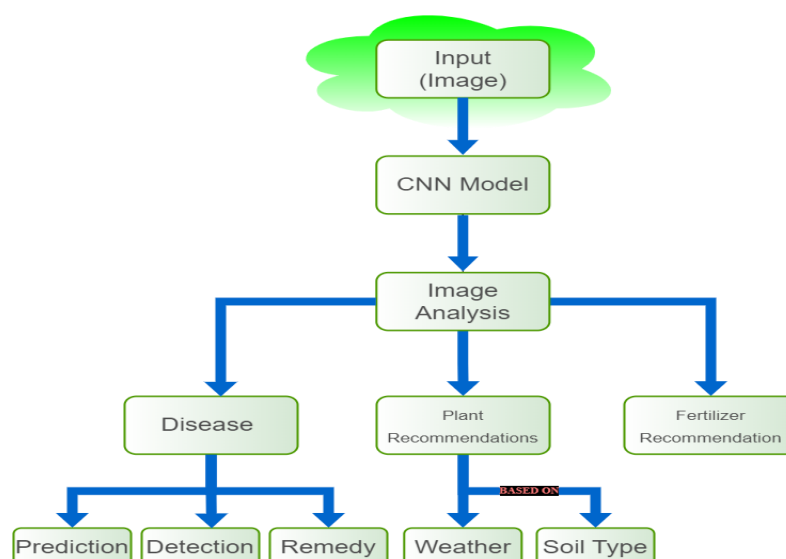
## Aim

To list down the fixed and variable cost elements for the chosen problem statement

**Fixed cost and Variable cost: -**

FIXED COSTS	VARIABLE COSTS
AI model development (CNN for disease, KNN for crop recommendation)	Cloud storage as number of users and image uploads increases
Platform development (Registration, Login, Dashboard, UI)	API usage costs (weather data, commodity prices) as usage scales
Third-party tools and software libraries (e.g., TensorFlow, Django)	Programming costs for updates and adding new features
Initial database setup (crop data, remedies, user info)	Personalisation for users (e.g., tailoring crop recommendations to specific needs)
API integration setup (Gov commodity API, weather APIs)	Model inference and processing costs per user (if hosted on cloud)
Initial cloud server and hosting setup	Maintenance and bug fixes over time

### Work Flow:



## EXPERIMENT 3:

### Aim

To prioritize top 3 features from the functionalities, code, test and implement them.

Top 3 Prioritized features are

### 1. Crop Disease Detection and Remedy Management

- This feature is vital for **early identification of diseases** in crops through leaf image analysis using a CNN model. It provides **instant remedies from a curated database**, helping them to
  - Reduce crop loss
  - Apply targeted treatment
  - Save time and resources
- Many farmers struggle to identify crop diseases in time. This feature acts as a **digital plant doctor**, empowering them with timely, accurate disease diagnosis and remedy suggestions.

### 2. Commodity Price Visualization

- This feature fetches **real-time commodity prices** from government API and presents them in a **simple bar chart** for easy understanding. It helps farmers to
  - Decide **when and where to sell**
  - Analyse price trends visually
  - Make **profitable market decisions**
- Farmers often lack access to timely price data. Visualizing this data helps them **avoid middlemen exploitation** and maximize their earnings.

### **3. Crop Recommendation System**

- Based on **location-specific weather data** (fetched via APIs) and soil nutrients (NPK), this feature uses a **KNN model** to recommend the most suitable crops for the conditions. It helps in
  - Optimize land usage
  - Improve crop yield
  - Support sustainable farming practices
- Farmers may not always know which crop suits current environmental and soil conditions. This intelligent recommendation system guides them to **choose the right crop at the right time**.

### **CODE SNIPPET:**

#### **Disease Detection:**

```
def detect_disease(image_path):  
    if not os.path.exists(image_path):  
        return HttpResponse("Image file not found.", status=404)  
  
    else:  
        image=tf.keras.preprocessing.image.load_img(image_path,target_size=(128,128))  
        input_arr=tf.keras.preprocessing.image.img_to_array(image)  
        input_arr=np.array([input_arr])  
        print(image_path)  
        disease=modelTest.fun(image_path)  
        disease_info = read(disease)
```

```
data = {  
    "dname": disease_info[0],  
    "dcause": disease_info[1],  
    "dtype": disease_info[2],  
    "dremedy": disease_info[3] }
```

```
print(input_arr.shape)  
#createRemedy()  
context = {  
    "image_url": image_path,  
    "predicted_disease": disease,  
    "data": data }
```

```
return context
```

### **Price Visualization:**

```
def price_visualization(request):  
    # Retrieve selected commodities from the session  
    selected_commodities = request.session.get('selected_commodities', [])  
    print("Selected Commodities in Visualization:", selected_commodities) #  
    Debugging
```

```
if not selected_commodities:  
    return HttpResponseBadRequest("No commodities selected.")
```

```
# Fetch and process data
```

```
data = fetch_data()
```

```
if data:
```

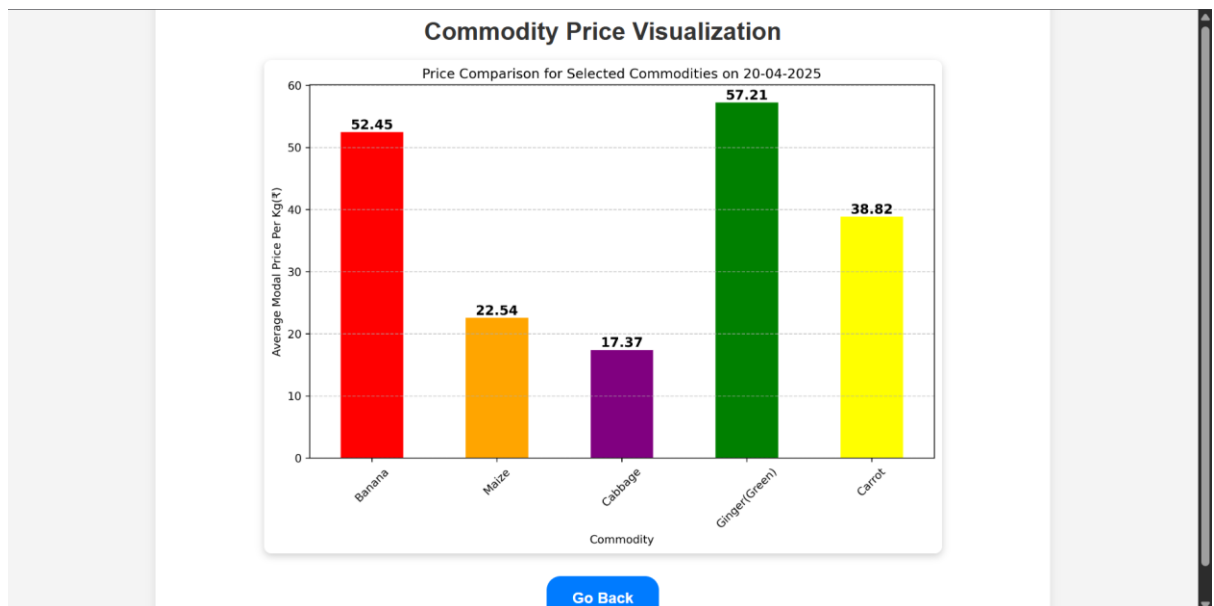
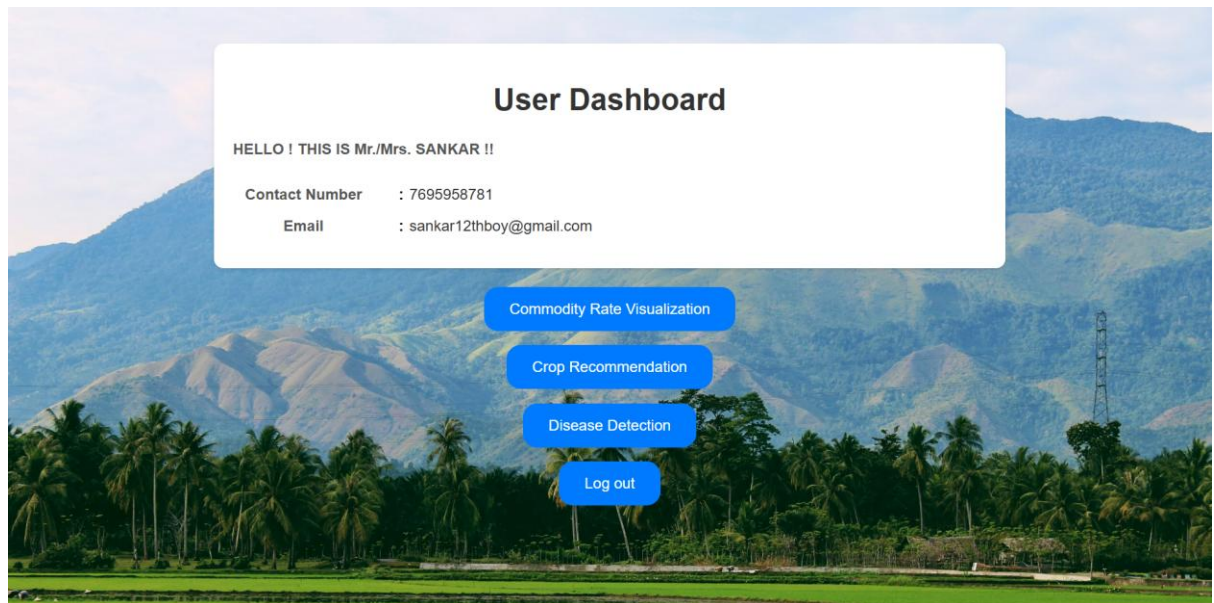
```
processed_df = process_data(data, selected_commodities)
plot_data(processed_df, selected_commodities)

# Pass the selected commodities to the template
context = {
    'selected_commodities': selected_commodities,
    'image_path': 'detector/static/commodity_prices.png' # Path to the saved
image
}
return render(request, 'myapp/price_visualization.html', context)
```

### **Show Remedy:**

```
def show_remedy(request):
    if request.method == "POST":
        disease_name = request.POST.get("dname")
        # Fetch remedy from database (assuming you have a model for it)
        disease = Remedy.objects.get(dname=disease_name)
        return render(request, "myapp/remedy.html", {"data": disease})
```

## OUTPUT SNAPSHOT :



### Enter Soil Parameters

Nitrogen (kg/ha):  
50

Phosphorus (kg/ha):  
30

Potassium (kg/ha):  
40

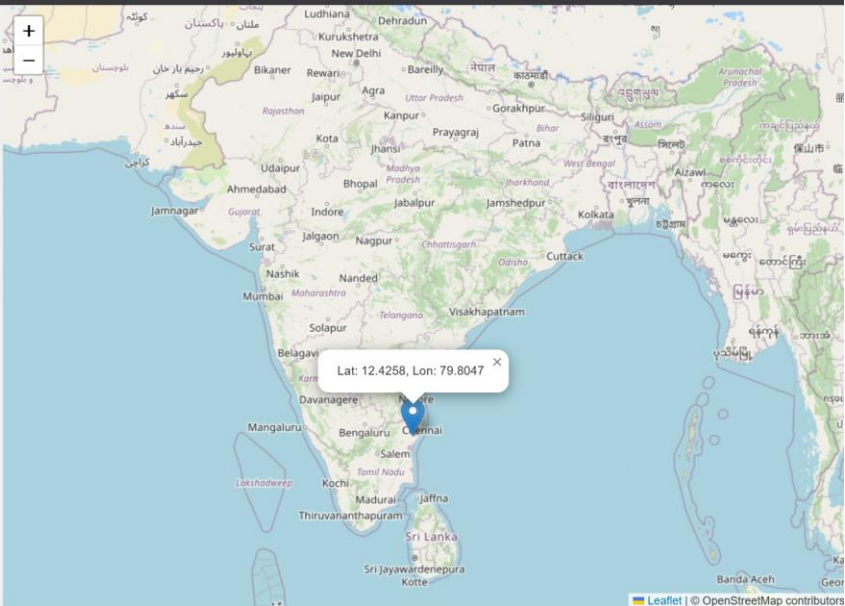
Soil pH:  
6.5

Submit Data

Cancel

Back

Click on the map to choose a location, then submit.

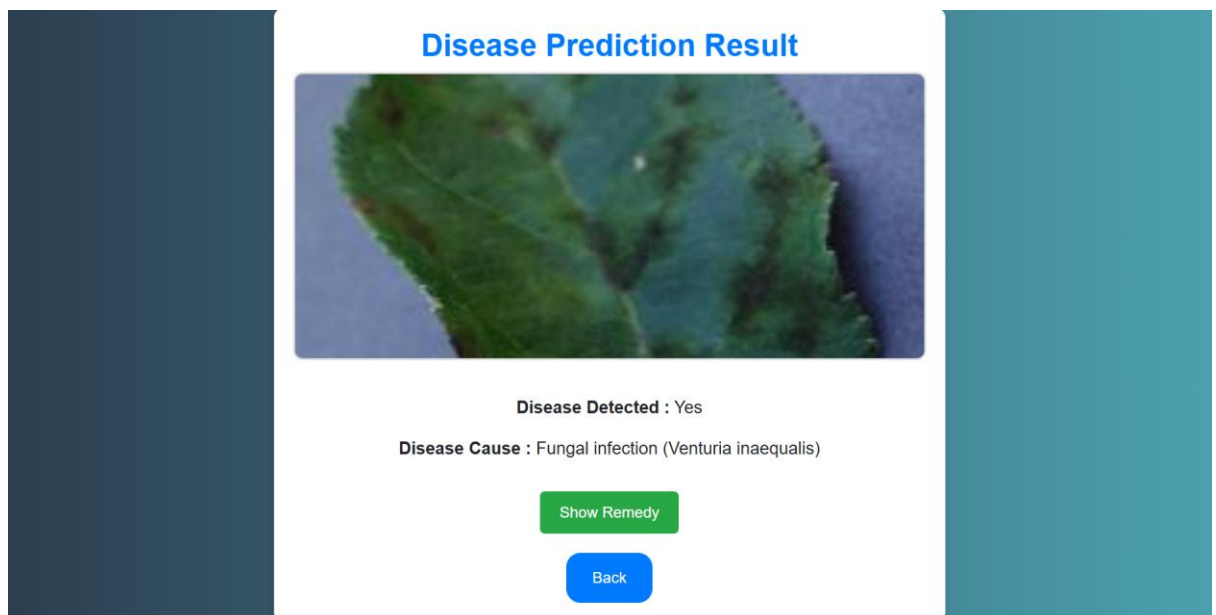
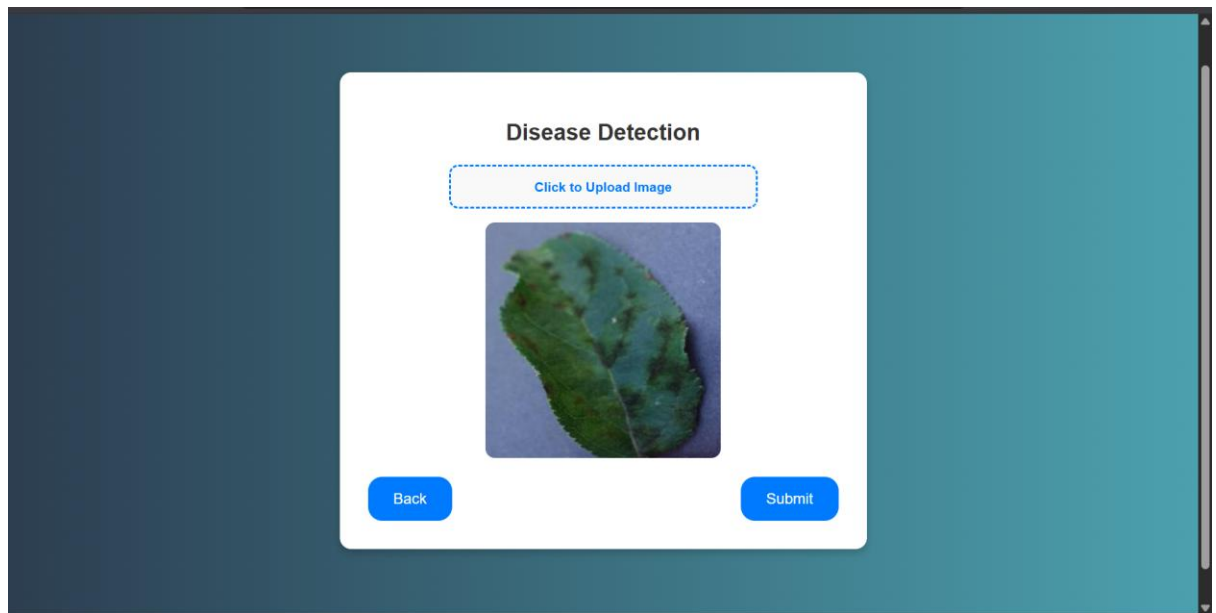


### Crop Prediction Result

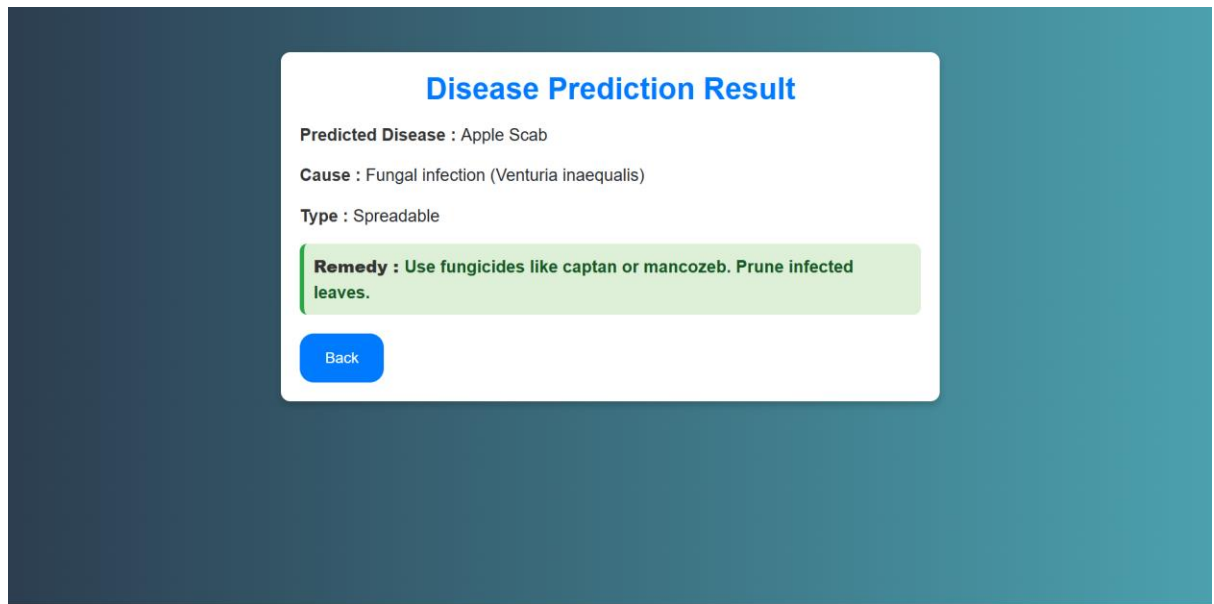
🌡 Temperature:	28.03 °C
💧 Humidity:	81%
☁ Rainfall:	44.0 mm
🌱 Nitrogen:	50 kg/ha
🌱 Phosphorus:	30 kg/ha
🌱 Potassium:	40 kg/ha
🌱 Soil pH:	6.5

🌱 Predicted Crop: papaya

Go Back







#### LEARNING OUTCOMES:

- Able to understand how the software works.
- Learnt technology stacks for frontend and backend.