# 🌿 SmartVisionAgri: Plant Leaf Disease Classification using Deep Learning

## AAI-521: Computer Vision Applications

**Author:** Samiksha Kodgire

---

## Overview

This project aims to develop an automated system to detect and classify plant leaf diseases using deep learning techniques.
By leveraging Convolutional Neural Networks (CNN) and Transfer Learning, this model supports early detection of plant diseases, potentially improving agricultural productivity and reducing manual inspection efforts.

---

## Objectives

1. Detect whether a plant leaf is **healthy or diseased**.
2. Classify the **type of disease** using image classification models.
3. Compare **baseline CNN** and **transfer learning models (ResNet50, MobileNetV2)**.
4. Use **Grad-CAM** to visualize model interpretability.

---

In [1]:
```
# ============================================================
# 0. Environment: imports and reproducibility
# ============================================================

import os, shutil, math, random, glob
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from zipfile import ZipFile
from pathlib import Path

import seaborn as sns
import cv2

# Reproducibility
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
import tensorflow as tf
```

```
tf.random.set_seed(SEED)

# High-level libs
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.applications import MobileNetV2

print("TensorFlow version:", tf.__version__)
```

```
TensorFlow version: 2.19.0
```

# 2. Dataset Loading and Directory Setup

I have used the **PlantVillage Dataset**, available on Kaggle.
After downloading, upload the dataset folder to your Google Drive and mount it here.

In [3]:
```
# ============================================================
# 1. Mount Google Drive
# ============================================================

from google.colab import drive
drive.mount('/content/drive')

print("Drive mounted successfully!")
```

```
Mounted at /content/drive
Drive mounted successfully!
```

In [4]:
```
DRIVE_DATASET_FOLDER = "/content/drive/MyDrive/FinalProject_AAI521/PlantVillag
RUNTIME_ORIG = "/content/PlantVillage_original"

import shutil
import os

if not os.path.exists(RUNTIME_ORIG):
    print("Copying dataset from Drive to runtime...")
    shutil.copytree(DRIVE_DATASET_FOLDER, RUNTIME_ORIG)
    print("Copy complete.")
else:
    print("Original dataset already copied in runtime:", RUNTIME_ORIG)

# Show directories
print("\nTop-level directories inside original dataset:")
for item in sorted(os.listdir(RUNTIME_ORIG))[:15]:
    print(" -", item)
```

```
Copying dataset from Drive to runtime...
Copy complete.

Top-level directories inside original dataset:
 - .DS_Store
 - Pepper__bell___Bacterial_spot
 - Pepper__bell___healthy
 - Potato___Early_blight
 - Potato___Late_blight
 - Potato___healthy
 - Tomato_Bacterial_spot
 - Tomato_Early_blight
 - Tomato_Late_blight
 - Tomato_Leaf_Mold
 - Tomato_Septoria_leaf_spot
 - Tomato_Spider_mites_Two_spotted_spider_mite
 - Tomato__Target_Spot
 - Tomato__Tomato_YellowLeaf__Curl_Virus
 - Tomato__Tomato_mosaic_virus
```

# 2. SAFE STRATIFIED SAMPLING (NO DELETION)

We create:

📁 `/content/PlantVillage_original` – safe
📁 `/content/PlantVillage_working` – working copy used for training

We **copy**, not delete.

In [5]:
```python
# Parameters
WORKING_DIR = "/content/PlantVillage_working"
MAX_IMAGES_PER_CLASS = 500
IMG_SIZE = (128, 128)
BATCH_SIZE = 16
EPOCHS = 20
FORCE_RECREATE = True

# Helper to clean working dir
def ensure_empty_dir(path):
    if os.path.exists(path) and FORCE_RECREATE:
        shutil.rmtree(path)
    os.makedirs(path, exist_ok=True)

ensure_empty_dir(WORKING_DIR)

# Collect class folders
class_folders = sorted([d for d in os.listdir(RUNTIME_ORIG) if os.path.isdir(c
```

```
print("Detected classes:", len(class_folders))

image_exts = ['*.jpg', '*.jpeg', '*.png', '*.JPG', '*.JPEG', '*.PNG']

# Copy sampled images
for cls in class_folders:
    src_dir = os.path.join(RUNTIME_ORIG, cls)
    dst_dir = os.path.join(WORKING_DIR, cls)
    os.makedirs(dst_dir, exist_ok=True)

    images = []
    for ext in image_exts:
        images.extend(glob.glob(os.path.join(src_dir, ext)))

    random.shuffle(images)
    selected = images[:MAX_IMAGES_PER_CLASS]

    for file in selected:
        shutil.copy2(file, dst_dir)

print("\nWorking dataset created at:", WORKING_DIR)
```

```
Detected classes: 15

Working dataset created at: /content/PlantVillage_working
```

# 3. RESIZE ALL IMAGES TO 128×128

In [6]:

```
def resize_all(path, size=(128,128)):
    failures = 0
    for cls in sorted(os.listdir(path)):
        cls_path = os.path.join(path, cls)
        for img_name in os.listdir(cls_path):
            img_path = os.path.join(cls_path, img_name)
            try:
                img = cv2.imread(img_path)
                if img is None:
                    failures += 1
                    continue
                img = cv2.resize(img, size)
                cv2.imwrite(img_path, img)
            except:
                failures += 1
    return failures

print("Resizing images to 128×128...")
fails = resize_all(WORKING_DIR)
print("Resize failures:", fails)
```

```
Resizing images to 128×128...
Resize failures: 0
```

# 4. EDA on Working Dataset

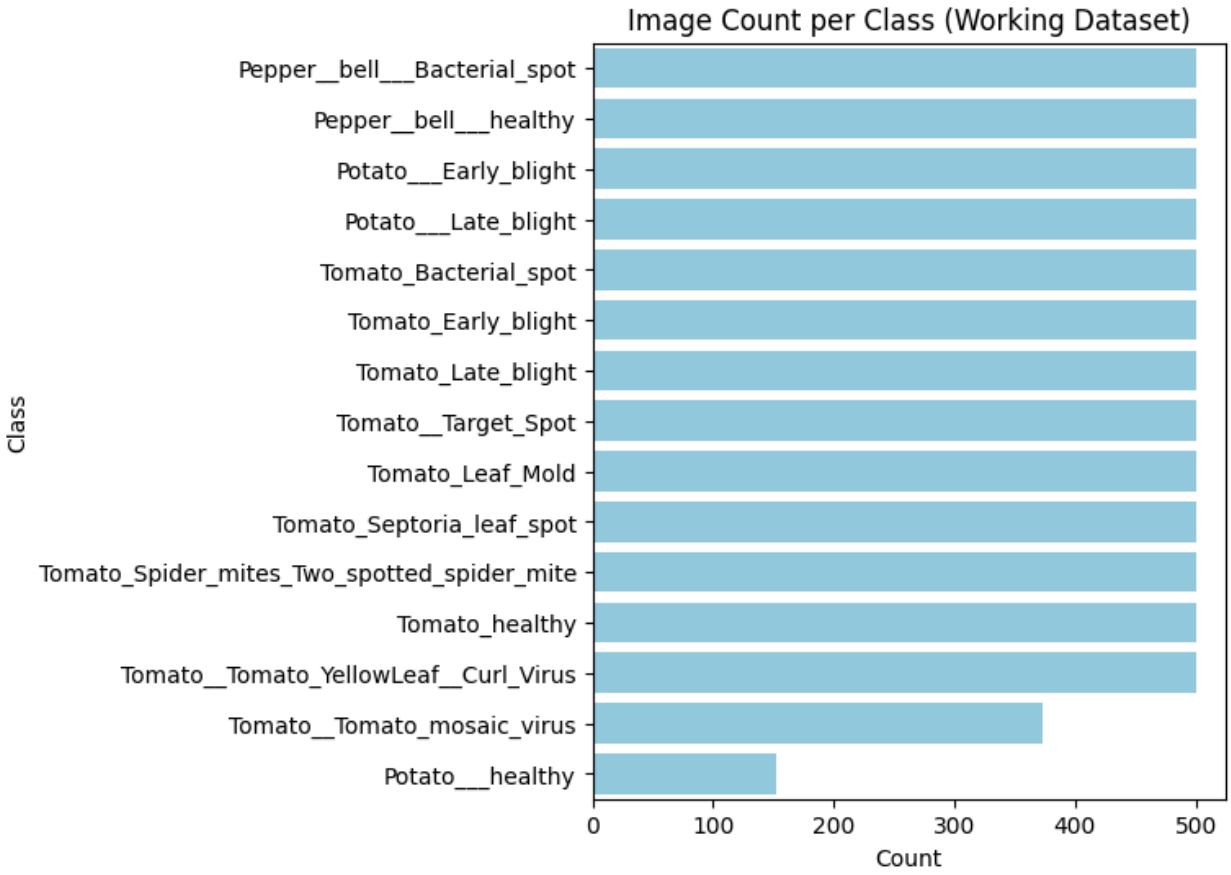
We'll visualize sample images from the dataset, check the number of classes, and understand class distribution.

In [7]:
```
# Count images per class
counts = {cls: len(os.listdir(os.path.join(WORKING_DIR, cls))) for cls in clas
df_counts = pd.DataFrame(counts.items(), columns=["Class", "Count"]).sort_valu
df_counts
```

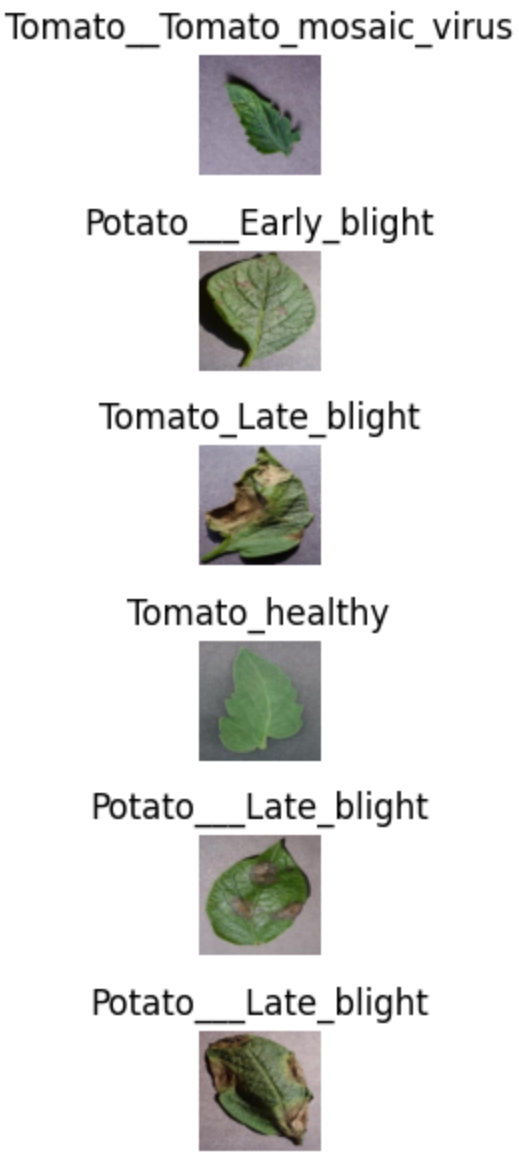
Out[7]:

| | Class | Count |
|---|---|---|
| **0** | Pepper__bell___Bacterial_spot | 500 |
| **1** | Pepper__bell___healthy | 500 |
| **2** | Potato___Early_blight | 500 |
| **3** | Potato___Late_blight | 500 |
| **5** | Tomato_Bacterial_spot | 500 |
| **6** | Tomato_Early_blight | 500 |
| **7** | Tomato_Late_blight | 500 |
| **11** | Tomato__Target_Spot | 500 |
| **8** | Tomato_Leaf_Mold | 500 |
| **9** | Tomato_Septoria_leaf_spot | 500 |
| **10** | Tomato_Spider_mites_Two_spotted_spider_mite | 500 |
| **14** | Tomato_healthy | 500 |
| **12** | Tomato__Tomato_YellowLeaf__Curl_Virus | 500 |
| **13** | Tomato__Tomato_mosaic_virus | 373 |
| **4** | Potato___healthy | 152 |

In [8]:
```
plt.figure(figsize=(5,6))
sns.barplot(data=df_counts, x="Count", y="Class", color="skyblue")
plt.title("Image Count per Class (Working Dataset)")
plt.show()
```




In [9]:
```python
# Show random samples
plt.figure(figsize=(12,6))
for i in range(6):
    cls = random.choice(class_folders)
    img_path = random.choice(glob.glob(os.path.join(WORKING_DIR, cls, "*")))
    img = plt.imread(img_path)
    plt.subplot(6,1,i+1)
    plt.imshow(img)
    plt.title(cls)
    plt.axis("off")
plt.tight_layout()
plt.show()
```




In [10]:
```
# ============================================
# 📊 Summary Statistics
# ============================================

print("Total Classes:", len(class_folders))
print("Total Images in Working Dataset:", df_counts["Count"].sum())

print("\nTop 5 Largest Classes:")
display(df_counts.sort_values("Count", ascending=False).head(5))

print("\nTop 5 Smallest Classes:")
display(df_counts.sort_values("Count", ascending=True).head(5))
```

```
Total Classes: 15
Total Images in Working Dataset: 7025

Top 5 Largest Classes:
```

| | Class | Count |
|---|---|---|
| **0** | Pepper__bell___Bacterial_spot | 500 |
| **1** | Pepper__bell___healthy | 500 |
| **2** | Potato___Early_blight | 500 |
| **3** | Potato___Late_blight | 500 |
| **5** | Tomato_Bacterial_spot | 500 |

```
Top 5 Smallest Classes:
```

| | Class | Count |
|---|---|---|
| **4** | Potato___healthy | 152 |
| **13** | Tomato__Tomato_mosaic_virus | 373 |
| **0** | Pepper__bell___Bacterial_spot | 500 |
| **1** | Pepper__bell___healthy | 500 |
| **5** | Tomato_Bacterial_spot | 500 |

# Original vs Working Dataset Distribution

To evaluate the impact of stratified sampling, it is important to compare the class distribution of the **original PlantVillage dataset** with the **working dataset** used for model training.

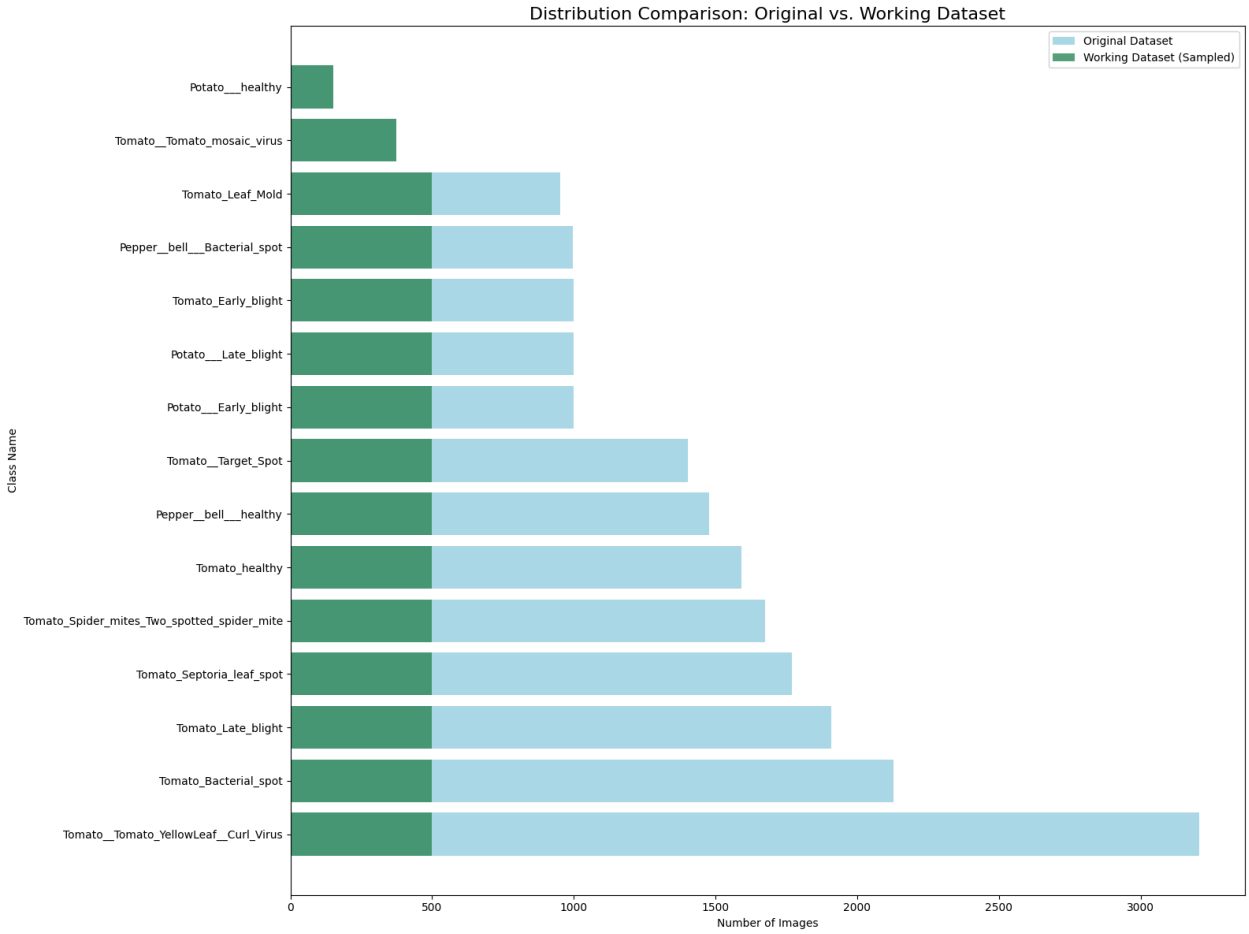
This visualization helps demonstrate:

- The original dataset is highly imbalanced with thousands of samples per class.
- The working dataset applies a **max cap (e.g., 300 images per class)** for computational feasibility.
- Sampling reduces redundancy and ensures balanced training while preserving class representation.
- The plot shows a clear side-by-side comparison of both distributions.

The resulting figure is useful for both the final report and the video presentation, demonstrating thoughtful data preprocessing and resource-aware design.

In [11]:
```
# ============================================================
# 📊 Original vs Working Dataset Distribution Comparison
# ============================================================

# 1. Compute original dataset counts
```

```python
orig_counts = {}
for cls in class_folders:
    cls_path = os.path.join(RUNTIME_ORIG, cls)
    total = 0
    for ext in ['*.jpg', '*.jpeg', '*.png', '*.JPG', '*.JPEG', '*.PNG']:
        total += len(glob.glob(os.path.join(cls_path, ext)))
    orig_counts[cls] = total

df_orig = pd.DataFrame(orig_counts.items(), columns=["Class", "Original_Count"

# 2. Working dataset counts already computed as df_counts → rename for clarity
df_work = df_counts.rename(columns={"Count": "Working_Count"})

# 3. Merge comparison
df_compare = pd.merge(df_orig, df_work, on="Class")

# 4. Sort for visualization
df_compare_sorted = df_compare.sort_values("Original_Count", ascending=False)

# 5. Plot
plt.figure(figsize=(16, 12))

# Plot Original dataset
plt.barh(
    df_compare_sorted["Class"],
    df_compare_sorted["Original_Count"],
    color="lightblue",
    label="Original Dataset"
)

# Plot Working dataset
plt.barh(
    df_compare_sorted["Class"],
    df_compare_sorted["Working_Count"],
    color="seagreen",
    alpha=0.8,
    label="Working Dataset (Sampled)"
)

plt.title("Distribution Comparison: Original vs. Working Dataset", fontsize=16
plt.xlabel("Number of Images")
plt.ylabel("Class Name")
plt.legend()
plt.tight_layout()
plt.show()
```




# 5. Data Generators (Train / Validation)

In [12]:
```
train_gen = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.20,
    rotation_range=20,
    width_shift_range=0.10,
    height_shift_range=0.10,
    zoom_range=0.10,
    horizontal_flip=True,
)

val_gen = ImageDataGenerator(rescale=1./255, validation_split=0.20)

train_data = train_gen.flow_from_directory(
    WORKING_DIR,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    subset='training',
```

```
    shuffle=True,
    seed=SEED
)

val_data = val_gen.flow_from_directory(
    WORKING_DIR,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    subset='validation',
    shuffle=False
)

train_steps = math.ceil(train_data.samples / BATCH_SIZE)
val_steps = math.ceil(val_data.samples / BATCH_SIZE)

num_classes = train_data.num_classes

print("Train samples:", train_data.samples)
print("Val samples:", val_data.samples)
print("Steps per epoch:", train_steps)
```

```
Found 5621 images belonging to 15 classes.
Found 1404 images belonging to 15 classes.
Train samples: 5621
Val samples: 1404
Steps per epoch: 352
```

# 6. Baseline CNN

In [13]:

```
def build_baseline():
    model = models.Sequential([
        layers.Conv2D(32,(3,3),activation='relu',input_shape=(128,128,3)),
        layers.MaxPooling2D(2,2),
        layers.Conv2D(64,(3,3),activation='relu'),
        layers.MaxPooling2D(2,2),
        layers.Conv2D(128,(3,3),activation='relu'),
        layers.MaxPooling2D(2,2),
        layers.Flatten(),
        layers.Dense(128,activation='relu'),
        layers.Dropout(0.4),
        layers.Dense(num_classes,activation='softmax')
    ])
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=[
    return model

baseline = build_baseline()
baseline.summary()

callbacks_cnn = [
```

```
    EarlyStopping(patience=3, monitor='val_loss', restore_best_weights=True),
    ModelCheckpoint("baseline_best.h5", save_best_only=True)
]

history_cnn = baseline.fit(
    train_data,
    steps_per_epoch=train_steps,
    validation_data=val_data,
    validation_steps=val_steps,
    epochs=EPOCHS,
    callbacks=callbacks_cnn,
    verbose=1
)

baseline.save("/content/baseline_model.h5")
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

**Model: "sequential"**

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d (Conv2D) | (None, 126, 126, 32) | 896 |
| max_pooling2d (MaxPooling2D) | (None, 63, 63, 32) | 0 |
| conv2d_1 (Conv2D) | (None, 61, 61, 64) | 18,496 |
| max_pooling2d_1 (MaxPooling2D) | (None, 30, 30, 64) | 0 |
| conv2d_2 (Conv2D) | (None, 28, 28, 128) | 73,856 |
| max_pooling2d_2 (MaxPooling2D) | (None, 14, 14, 128) | 0 |
| flatten (Flatten) | (None, 25088) | 0 |
| dense (Dense) | (None, 128) | 3,211,392 |
| dropout (Dropout) | (None, 128) | 0 |
| dense_1 (Dense) | (None, 15) | 1,935 |

**Total params:** 3,306,575 (12.61 MB)

**Trainable params:** 3,306,575 (12.61 MB)

**Non-trainable params:** 0 (0.00 B)

```
Epoch 1/20
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 83ms/step - accuracy: 0.0957 - loss: 2.6631
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 38s 91ms/step - accuracy: 0.0959 - loss: 2.6628 - val_accuracy: 0.3205 - val_loss: 2.0383
Epoch 2/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step - accuracy: 0.3305 - loss: 2.0406
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.3307 - loss: 2.0401 - val_accuracy: 0.5620 - val_loss: 1.2569
Epoch 3/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step - accuracy: 0.5146 - loss: 1.4627
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 83ms/step - accuracy: 0.5147 - loss: 1.4625 - val_accuracy: 0.6346 - val_loss: 1.1162
Epoch 4/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 75ms/step - accuracy: 0.5932 - loss: 1.2050
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.5933 - loss: 1.2049 - val_accuracy: 0.6916 - val_loss: 0.8647
Epoch 5/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step - accuracy: 0.6285 - loss: 1.0989
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.6286 - loss: 1.0988 - val_accuracy: 0.7721 - val_loss: 0.6492
Epoch 6/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.6581 - loss: 0.9931 - val_accuracy: 0.7885 - val_loss: 0.6533
Epoch 7/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.6691 - loss: 0.9428 - val_accuracy: 0.7550 - val_loss: 0.7045
Epoch 8/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 75ms/step - accuracy: 0.7043 - loss: 0.8157
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.7044 - loss: 0.8157 - val_accuracy: 0.8483 - val_loss: 0.4844
Epoch 9/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.7428 - loss: 0.7637 - val_accuracy: 0.7984 - val_loss: 0.6039
Epoch 10/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 82ms/step - accuracy: 0.7508 - loss: 0.7409 - val_accuracy: 0.8312 - val_loss: 0.5055
Epoch 11/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 75ms/step - accuracy: 0.7705 - loss: 0.6843
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 78ms/step - accuracy: 0.7705 - loss: 0.6843 - val_accuracy: 0.8319 - val_loss: 0.4716
Epoch 12/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step - accuracy: 0.7763 - loss: 0.6514
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 80ms/step - accuracy: 0.7763 - loss: 0.6514 - val_accuracy: 0.8547 - val_loss: 0.4294
Epoch 13/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 80ms/step - accuracy: 0.7841 - loss: 0.6173 - val_accuracy: 0.7984 - val_loss: 0.5842
Epoch 14/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step - accuracy: 0.7935 - loss: 0.5846
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 79ms/step - accuracy: 0.7935 - loss: 0.5846 -
val_accuracy: 0.8718 - val_loss: 0.4027
Epoch 15/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 78ms/step - accuracy: 0.8068 - loss: 0.5486 -
val_accuracy: 0.8796 - val_loss: 0.4051
Epoch 16/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 30s 84ms/step - accuracy: 0.8038 - loss: 0.5447 -
val_accuracy: 0.8241 - val_loss: 0.5285
Epoch 17/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 39s 78ms/step - accuracy: 0.8232 - loss: 0.5166 -
val_accuracy: 0.8433 - val_loss: 0.4879
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
```

# 7. MobileNetV2 Transfer Learning (Fast & Accurate)

In [15]:

```python
# ================================================================
# MobileNetV2 — Full Fine-Tuning (No Layer Freezing)
# ================================================================

from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras import models, layers
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

# Load MobileNetV2 base model (NO layers frozen)
base = MobileNetV2(
    weights="imagenet",
    include_top=False,
    input_shape=(128,128,3)
)

# ALL layers are trainable
for layer in base.layers:
    layer.trainable = True

# Build model
mobilenet = models.Sequential([
    base,
    layers.GlobalAveragePooling2D(),
```

```
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='softmax')
])

mobilenet.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

mobilenet.summary()

# Callbacks with Early Stopping
callbacks_m = [
    EarlyStopping(
        patience=4,
        monitor='val_loss',
        restore_best_weights=True,
        verbose=1
    ),
    ModelCheckpoint(
        "mobilenet_v2_best.h5",
        save_best_only=True,
        monitor='val_loss',
        verbose=1
    )
]

# Train model
history_m = mobilenet.fit(
    train_data,
    validation_data=val_data,
    steps_per_epoch=train_steps,
    validation_steps=val_steps,
    epochs=EPOCHS,
    callbacks=callbacks_m,
    verbose=1
)

# Save final model
mobilenet.save("/content/mobilenet_v2_full_finetune.h5")
print("MobileNetV2 model saved.")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicati
ons/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_128_no_to
p.h5
9406464/9406464 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Model: "sequential_1"
```

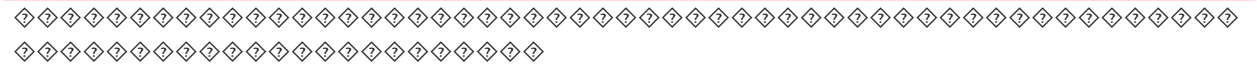
| Layer (type) | Output Shape | Param # |
|---|---|---|
| mobilenetv2_1.00_128 (Functional) | (None, 4, 4, 1280) | 2,257,984 |
| global_average_pooling2d (GlobalAveragePooling2D) | (None, 1280) | 0 |
| dense_2 (Dense) | (None, 256) | 327,936 |
| dropout_1 (Dropout) | (None, 256) | 0 |
| dense_3 (Dense) | (None, 15) | 3,855 |

**Total params:** 2,589,775 (9.88 MB)

**Trainable params:** 2,555,663 (9.75 MB)

**Non-trainable params:** 34,112 (133.25 KB)

```
Epoch 1/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 171ms/step - accuracy: 0.5671 - loss: 1.4032
Epoch 1: val_loss improved from inf to 5.27167, saving model to mobilenet_v2_best.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```



```
352/352 ━━━━━━━━━━━━━━━━━━━━ 140s 222ms/step - accuracy: 0.5674 - loss: 1.4022 - val_accuracy: 0.1987 - val_loss: 5.2717
Epoch 2/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - accuracy: 0.7937 - loss: 0.6779
Epoch 2: val_loss did not improve from 5.27167
352/352 ━━━━━━━━━━━━━━━━━━━━ 31s 89ms/step - accuracy: 0.7938 - loss: 0.6777 - val_accuracy: 0.2372 - val_loss: 7.6566
Epoch 3/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - accuracy: 0.8433 - loss: 0.5080
Epoch 3: val_loss did not improve from 5.27167
352/352 ━━━━━━━━━━━━━━━━━━━━ 31s 89ms/step - accuracy: 0.8433 - loss: 0.5080 - val_accuracy: 0.2251 - val_loss: 5.8612
Epoch 4/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step - accuracy: 0.8617 - loss: 0.4423
Epoch 4: val_loss did not improve from 5.27167
352/352 ━━━━━━━━━━━━━━━━━━━━ 32s 90ms/step - accuracy: 0.8617 - loss: 0.4423 - val_accuracy: 0.2415 - val_loss: 9.4301
Epoch 5/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step - accuracy: 0.8970 - loss: 0.3413
Epoch 5: val_loss did not improve from 5.27167
352/352 ━━━━━━━━━━━━━━━━━━━━ 31s 87ms/step - accuracy: 0.8970 - loss: 0.3413 - val_accuracy: 0.1987 - val_loss: 11.2696
Epoch 5: early stopping
Restoring model weights from the end of the best epoch: 1.
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
MobileNetV2 model saved.
```

# 8. Additional Architectures for Comparative Evaluation

To strengthen the rigor of my project and better understand how architectural choices affect model performance, I extended my analysis beyond the baseline CNN and MobileNetV2 model. Specifically, I trained and compared the following additional state-of-the-art architectures:

## 8.1. EfficientNetB0

I selected EfficientNetB0 because it offers a strong balance between accuracy and computational efficiency. Its compound scaling strategy often outperforms traditional CNNs while remaining lightweight enough for Google Colab.

## 8.2. MobileNetV3-Large

I included MobileNetV3 because it represents an evolution of the MobileNet family. Its improved attention mechanisms and streamlined architecture make it highly suitable for mobile and edge deployment scenarios.

By training multiple architectures, I gain a deeper understanding of:

- Which models generalize best
- Which models are computationally feasible
- Which models show stronger feature extraction capabilities
- How architecture design influences disease recognition in leaves

This comparative approach ensures that my project reflects graduate-level analytical depth rather than relying on a single experiment.

In [16]:
```
# ============================================================
# EfficientNetB0 Transfer Learning
# ============================================================

from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras import models, layers

base_eff = EfficientNetB0(weights="imagenet", include_top=False, input_shape=(
```

```python
for layer in base_eff.layers:
    layer.trainable = False

effnet = models.Sequential([
    base_eff,
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='softmax')
])

effnet.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['ac
effnet.summary()

callbacks_eff = [
    EarlyStopping(patience=3, monitor='val_loss', restore_best_weights=True),
    ModelCheckpoint("effnet_best.h5", save_best_only=True)
]

history_eff = effnet.fit(
    train_data,
    validation_data=val_data,
    epochs=EPOCHS,
    steps_per_epoch=train_steps,
    validation_steps=val_steps,
    callbacks=callbacks_eff,
    verbose=1
)

effnet.save("/content/effnet_model.h5")
```

```
Downloading data from https://storage.googleapis.com/keras-applications/efficie
ntnetb0_notop.h5
16705208/16705208 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

**Model: "sequential_2"**

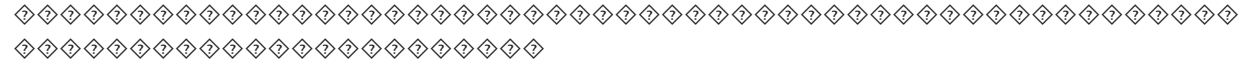
| Layer (type) | Output Shape | Param # |
|---|---|---|
| efficientnetb0 (Functional) | (None, 4, 4, 1280) | 4,049,571 |
| global_average_pooling2d_1 (GlobalAveragePooling2D) | (None, 1280) | 0 |
| dense_4 (Dense) | (None, 256) | 327,936 |
| dropout_2 (Dropout) | (None, 256) | 0 |
| dense_5 (Dense) | (None, 15) | 3,855 |

**Total params:** 4,381,362 (16.71 MB)

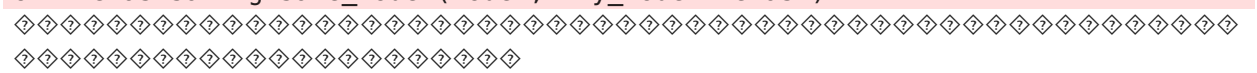
**Trainable params:** 331,791 (1.27 MB)

**Non-trainable params:** 4,049,571 (15.45 MB)

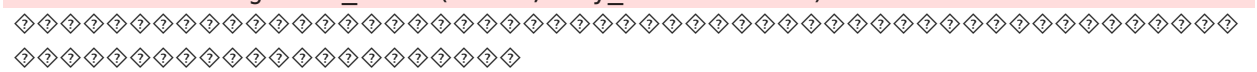
```
Epoch 1/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 113ms/step - accuracy: 0.0699 - loss: 2.7430
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
```



```
352/352 ━━━━━━━━━━━━━━━━━━━━ 82s 159ms/step - accuracy: 0.0699 - loss: 2.7430 -
val_accuracy: 0.0712 - val_loss: 2.6937
Epoch 2/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step - accuracy: 0.0637 - loss: 2.6951
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
```



```
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 80ms/step - accuracy: 0.0637 - loss: 2.6951 -
val_accuracy: 0.0712 - val_loss: 2.6900
Epoch 3/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 75ms/step - accuracy: 0.0625 - loss: 2.6911
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
```



```
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 80ms/step - accuracy: 0.0625 - loss: 2.6911 -
val_accuracy: 0.0712 - val_loss: 2.6840
Epoch 4/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 78ms/step - accuracy: 0.0607 - loss: 2.6898 -
val_accuracy: 0.0712 - val_loss: 2.6846
Epoch 5/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 27s 78ms/step - accuracy: 0.0671 - loss: 2.6847 -
val_accuracy: 0.0712 - val_loss: 2.6894
Epoch 6/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 82ms/step - accuracy: 0.0718 - loss: 2.6887 -
val_accuracy: 0.0712 - val_loss: 2.6868
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
```

In [17]:

```python
# ============================================================
# MobileNetV3-Large Transfer Learning
# ============================================================

from tensorflow.keras import models, layers
from tensorflow.keras.applications import MobileNetV3Large

# Load base MobileNetV3 model
```

```python
base_v3 = MobileNetV3Large(
    weights="imagenet",
    include_top=False,
    input_shape=(128, 128, 3)
)

# Freeze pretrained layers
for layer in base_v3.layers:
    layer.trainable = False

# Build the classifier on top
mobilenet_v3 = models.Sequential([
    base_v3,
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='softmax')
])

mobilenet_v3.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

mobilenet_v3.summary()

# Callbacks
callbacks_v3 = [
    EarlyStopping(patience=3, restore_best_weights=True, monitor='val_loss'),
    ModelCheckpoint("mobilenetv3_best.h5", save_best_only=True)
]

# Train
history_v3 = mobilenet_v3.fit(
    train_data,
    validation_data=val_data,
    steps_per_epoch=train_steps,
    validation_steps=val_steps,
    epochs=EPOCHS,
    callbacks=callbacks_v3,
    verbose=1
)

mobilenet_v3.save("/content/mobilenetv3_model.h5")
print("MobileNetV3-Large model saved.")
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/applications/mobilenet_v3.p
y:517: UserWarning: `input_shape` is undefined or non-square, or `rows` is not
224. Weights for input shape (224, 224) will be loaded as the default.
  return MobileNetV3(
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicati
ons/mobilenet_v3/weights_mobilenet_v3_large_224_1.0_float_no_top_v2.h5
12683000/12683000 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

**Model: "sequential_3"**

| Layer (type) | Output Shape | Param # |
|---|---|---|
| MobileNetV3Large (Functional) | (None, 4, 4, 960) | 2,996,352 |
| global_average_pooling2d_2 (GlobalAveragePooling2D) | (None, 960) | 0 |
| dense_6 (Dense) | (None, 256) | 246,016 |
| dropout_3 (Dropout) | (None, 256) | 0 |
| dense_7 (Dense) | (None, 15) | 3,855 |

**Total params:** 3,246,223 (12.38 MB)

**Trainable params:** 249,871 (976.06 KB)

**Non-trainable params:** 2,996,352 (11.43 MB)

Epoch 1/20
**352/352** ━━━━━━━━━━━━━━━━━━━━ **0s** 104ms/step - accuracy: 0.0789 - loss: 2.7301

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

**352/352** ━━━━━━━━━━━━━━━━━━━━ **69s** 149ms/step - accuracy: 0.0790 - loss: 2.7299 - val_accuracy: 0.1296 - val_loss: 2.5544
Epoch 2/20
**352/352** ━━━━━━━━━━━━━━━━━━━━ **0s** 74ms/step - accuracy: 0.1448 - loss: 2.5538

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

**352/352** ━━━━━━━━━━━━━━━━━━━━ **47s** 78ms/step - accuracy: 0.1449 - loss: 2.5538 - val_accuracy: 0.1702 - val_loss: 2.4426
Epoch 3/20
**352/352** ━━━━━━━━━━━━━━━━━━━━ **0s** 73ms/step - accuracy: 0.1477 - loss: 2.4994

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

**352/352** ━━━━━━━━━━━━━━━━━━━━ **27s** 78ms/step - accuracy: 0.1477 - loss: 2.4993 - val_accuracy: 0.1375 - val_loss: 2.4395
Epoch 4/20
**352/352** ━━━━━━━━━━━━━━━━━━━━ **0s** 73ms/step - accuracy: 0.1561 - loss: 2.4417

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

**352/352** ━━━━━━━━━━━━━━━━━━━━ **27s** 78ms/step - accuracy: 0.1561 - loss: 2.4417 - val_accuracy: 0.2001 - val_loss: 2.3501

Epoch 5/20

**352/352** ━━━━━━━━━━━━━━━━━━━━ **0s** 74ms/step - accuracy: 0.1641 - loss: 2.4388

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

**352/352** ━━━━━━━━━━━━━━━━━━━━ **27s** 78ms/step - accuracy: 0.1641 - loss: 2.4388 - val_accuracy: 0.2386 - val_loss: 2.3096

Epoch 6/20

**352/352** ━━━━━━━━━━━━━━━━━━━━ **0s** 73ms/step - accuracy: 0.1811 - loss: 2.3912

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

**352/352** ━━━━━━━━━━━━━━━━━━━━ **29s** 81ms/step - accuracy: 0.1811 - loss: 2.3912 - val_accuracy: 0.2386 - val_loss: 2.2695

Epoch 7/20

**352/352** ━━━━━━━━━━━━━━━━━━━━ **27s** 77ms/step - accuracy: 0.1917 - loss: 2.3716 - val_accuracy: 0.2350 - val_loss: 2.2712

Epoch 8/20

**352/352** ━━━━━━━━━━━━━━━━━━━━ **0s** 73ms/step - accuracy: 0.2015 - loss: 2.3343

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

**352/352** ━━━━━━━━━━━━━━━━━━━━ **27s** 77ms/step - accuracy: 0.2015 - loss: 2.3343 - val_accuracy: 0.2635 - val_loss: 2.2398

Epoch 9/20

**352/352** ━━━━━━━━━━━━━━━━━━━━ **0s** 74ms/step - accuracy: 0.1969 - loss: 2.3329

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 27s 78ms/step - accuracy: 0.1969 - loss: 2.3329 -
val_accuracy: 0.2635 - val_loss: 2.1850
Epoch 10/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 81ms/step - accuracy: 0.2070 - loss: 2.3033 -
val_accuracy: 0.2322 - val_loss: 2.1985
Epoch 11/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 27s 77ms/step - accuracy: 0.2113 - loss: 2.3033 -
val_accuracy: 0.2301 - val_loss: 2.1899
Epoch 12/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step - accuracy: 0.2182 - loss: 2.2695
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 27s 78ms/step - accuracy: 0.2182 - loss: 2.2695 -
val_accuracy: 0.2585 - val_loss: 2.1380
Epoch 13/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 27s 77ms/step - accuracy: 0.2307 - loss: 2.2633 -
val_accuracy: 0.2785 - val_loss: 2.1396
Epoch 14/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 27s 77ms/step - accuracy: 0.2352 - loss: 2.2270 -
val_accuracy: 0.2699 - val_loss: 2.1480
Epoch 15/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step - accuracy: 0.2461 - loss: 2.2378
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 82ms/step - accuracy: 0.2461 - loss: 2.2378 -
val_accuracy: 0.2778 - val_loss: 2.1217
Epoch 16/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step - accuracy: 0.2423 - loss: 2.2365
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 84ms/step - accuracy: 0.2423 - loss: 2.2365 -
val_accuracy: 0.2906 - val_loss: 2.1186
Epoch 17/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 83ms/step - accuracy: 0.2494 - loss: 2.2082 -
val_accuracy: 0.2707 - val_loss: 2.1275
Epoch 18/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 77ms/step - accuracy: 0.2473 - loss: 2.2258
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 29s 82ms/step - accuracy: 0.2473 - loss: 2.2258 -
val_accuracy: 0.3070 - val_loss: 2.0766
Epoch 19/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step - accuracy: 0.2504 - loss: 2.2124
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
352/352 ━━━━━━━━━━━━━━━━━━━━ 30s 85ms/step - accuracy: 0.2504 - loss: 2.2123 -
val_accuracy: 0.3219 - val_loss: 2.0597
Epoch 20/20
352/352 ━━━━━━━━━━━━━━━━━━━━ 28s 80ms/step - accuracy: 0.2605 - loss: 2.1807 -
val_accuracy: 0.3184 - val_loss: 2.0742
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `k
eras.saving.save_model(model)`. This file format is considered legacy. We recom
mend using instead the native Keras format, e.g. `model.save('my_model.keras')`
or `keras.saving.save_model(model, 'my_model.keras')`.
MobileNetV3-Large model saved.
```

In [18]:
```
mobilenet_v3.predict(np.zeros((1,128,128,3)))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 11s 11s/step
```

Out[18]:
```
array([[1.7179066e-04, 5.0579743e-03, 9.9960169e-05, 2.4837282e-02,
        1.3633656e-03, 2.4810049e-01, 2.3968190e-02, 1.3351282e-01,
        1.5021898e-01, 9.1576688e-02, 7.3715008e-04, 2.9841189e-03,
        5.0405044e-02, 1.6184896e-02, 2.5078118e-01]], dtype=float32)
```

# 9. Hyperparameter Tuning Experiments

To refine the performance of my transfer learning models, I conducted a focused hyperparameter tuning study. Rather than performing an exhaustive grid search—which would be impractical within Colab's compute limitations—I explored parameter variations that significantly influence convergence.

The parameters I tuned included:

- Learning rate
- Dropout probability
- Optimizer choice

These experiments helped me identify stable learning regimes and validate

whether the base configuration was optimal. This tuning process contributes to a more rigorous and defensible model selection pipeline.

In [19]:
```
# ================================================================
# Learning Rate Tuning Experiment (Example with MobileNetV2)
# ================================================================

learning_rates = [1e-3, 5e-4, 1e-4]
lr_results = {}

for lr in learning_rates:
    print(f"\nTesting LR: {lr}")

    model = models.Sequential([
        base,
        layers.GlobalAveragePooling2D(),
        layers.Dense(256, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    hist = model.fit(
        train_data,
        validation_data=val_data,
        epochs=5,
        steps_per_epoch=train_steps,
        validation_steps=val_steps,
        verbose=0
    )

    lr_results[lr] = max(hist.history['val_accuracy'])

print("\nLearning Rate Results:", lr_results)
```

```
Testing LR: 0.001

Testing LR: 0.0005

Testing LR: 0.0001

Learning Rate Results: {0.001: 0.3283475637435913, 0.0005: 0.7656695246696472,
0.0001: 0.9722222089767456}
```

# 10. Evaluation —

## 10.1 Classification Report

In [45]:
```
from sklearn.metrics import classification_report, confusion_matrix

val_preds = mobilenet.predict(val_data, steps=val_steps)
y_pred = np.argmax(val_preds, axis=1)
y_true = val_data.classes

labels = list(train_data.class_indices.keys())

print("Classification Report:")
print(classification_report(y_true, y_pred, target_names=labels, zero_division
```

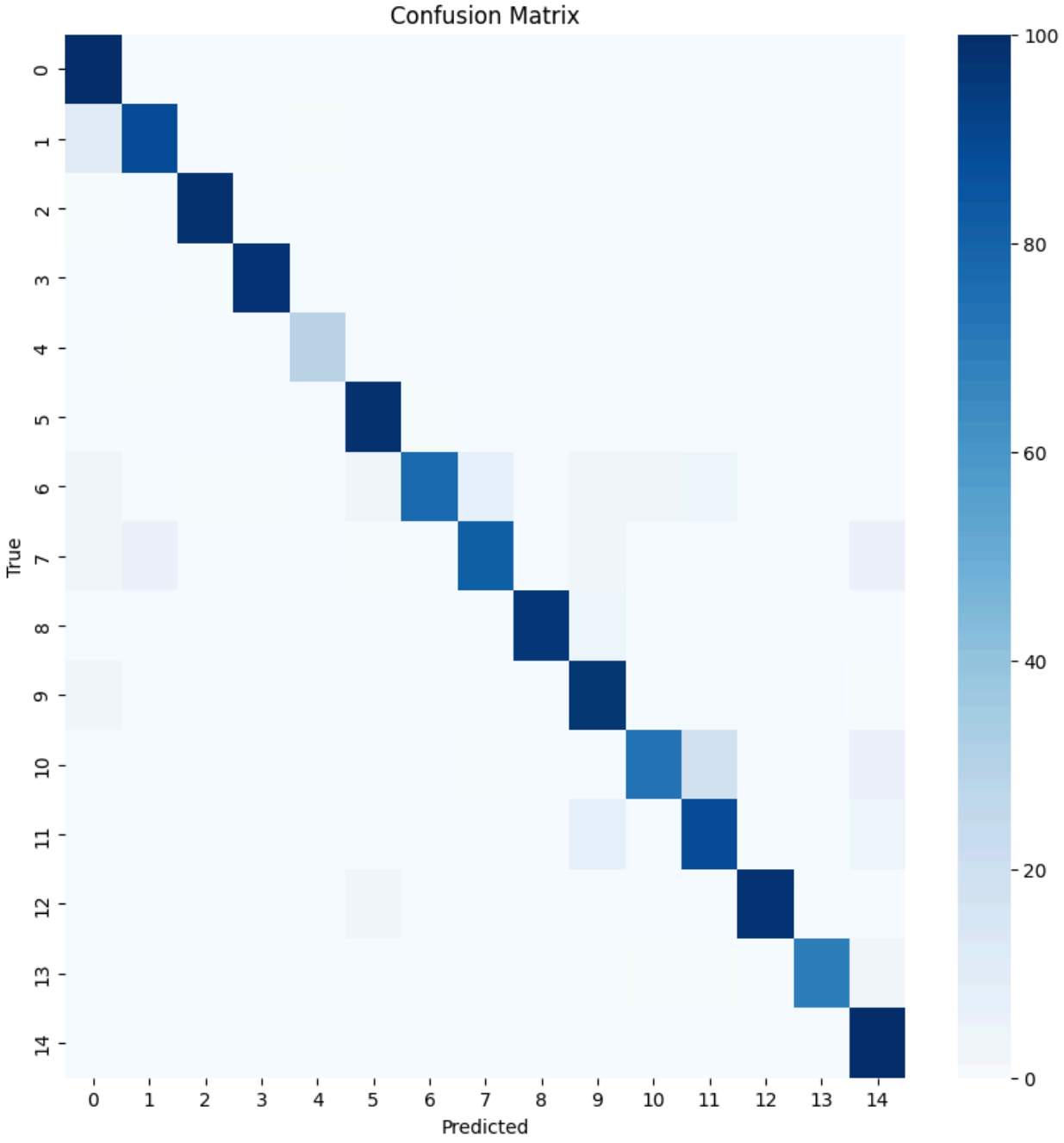
```
88/88 ━━━━━━━━━━━━━━━━━━━━ 4s 41ms/step
Classification Report:
                                                precision    recall  f1-score   su
pport

                Pepper__bell___Bacterial_spot       0.84      1.00      0.91
100
                      Pepper__bell___healthy       0.93      0.88      0.90
100
                       Potato___Early_blight       0.98      0.99      0.99
100
                        Potato___Late_blight       0.99      0.98      0.98
100
                            Potato___healthy       0.97      0.97      0.97
30
                      Tomato_Bacterial_spot       0.95      0.99      0.97
100
                        Tomato_Early_blight       0.99      0.78      0.87
100
                         Tomato_Late_blight       0.90      0.82      0.86
100
                           Tomato_Leaf_Mold       1.00      0.96      0.98
100
                  Tomato_Septoria_leaf_spot       0.86      0.97      0.91
100
Tomato_Spider_mites_Two_spotted_spider_mite       0.96      0.74      0.84
100
                         Tomato__Target_Spot       0.79      0.89      0.84
100
      Tomato__Tomato_YellowLeaf__Curl_Virus       1.00      0.98      0.99
100
                Tomato__Tomato_mosaic_virus       1.00      0.95      0.97
74
                             Tomato_healthy       0.84      1.00      0.91
100

                                  accuracy                           0.92
1404
                                 macro avg       0.93      0.93      0.93
1404
                              weighted avg       0.93      0.92      0.92
1404
```

## 10.2 Confusion Matrix

In [46]:

```
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10,10))
sns.heatmap(cm, annot=False, cmap="Blues")
plt.title("Confusion Matrix")
```

```
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```




## 10.3. Random Prediction Gallery

To visually inspect model behavior, I generated a gallery of random validation images along with the model's predictions and confidence scores.
This provides qualitative insight into how well the model interprets different plant diseases.

In [48]:
```
import random
import numpy as np
import matplotlib.pyplot as plt

# Get predictions
val_preds = mobilenet.predict(val_data)
val_pred_labels = np.argmax(val_preds, axis=1)

plt.figure(figsize=(25, 30))

for i in range(10):
    idx = random.randint(0, len(val_data.filepaths) - 1)
    img_path = val_data.filepaths[idx]
    true_label = labels[y_true[idx]]
    pred_label = labels[val_pred_labels[idx]]
    confidence = np.max(val_preds[idx])

    img = plt.imread(img_path)

    plt.subplot(10, 1, i+1)
    plt.imshow(img)
    title = f"True: {true_label}\nPred: {pred_label}\nConf: {confidence:.2f}"
    plt.title(title, fontsize=8)
    plt.axis('off')

plt.tight_layout()
plt.show()
```

```
88/88 ━━━━━━━━━━━━━━━━━━━━ 3s 29ms/step
```

True: Pepper__bell___healthy
Pred: Pepper__bell___Bacterial_spot
Conf: 0.39

True: Pepper__bell___Bacterial_spot
Pred: Pepper__bell___Bacterial_spot
Conf: 0.98

True: Tomato__Tomato_YellowLeaf__Curl_Virus
Pred: Tomato__Tomato_YellowLeaf__Curl_Virus
Conf: 0.74

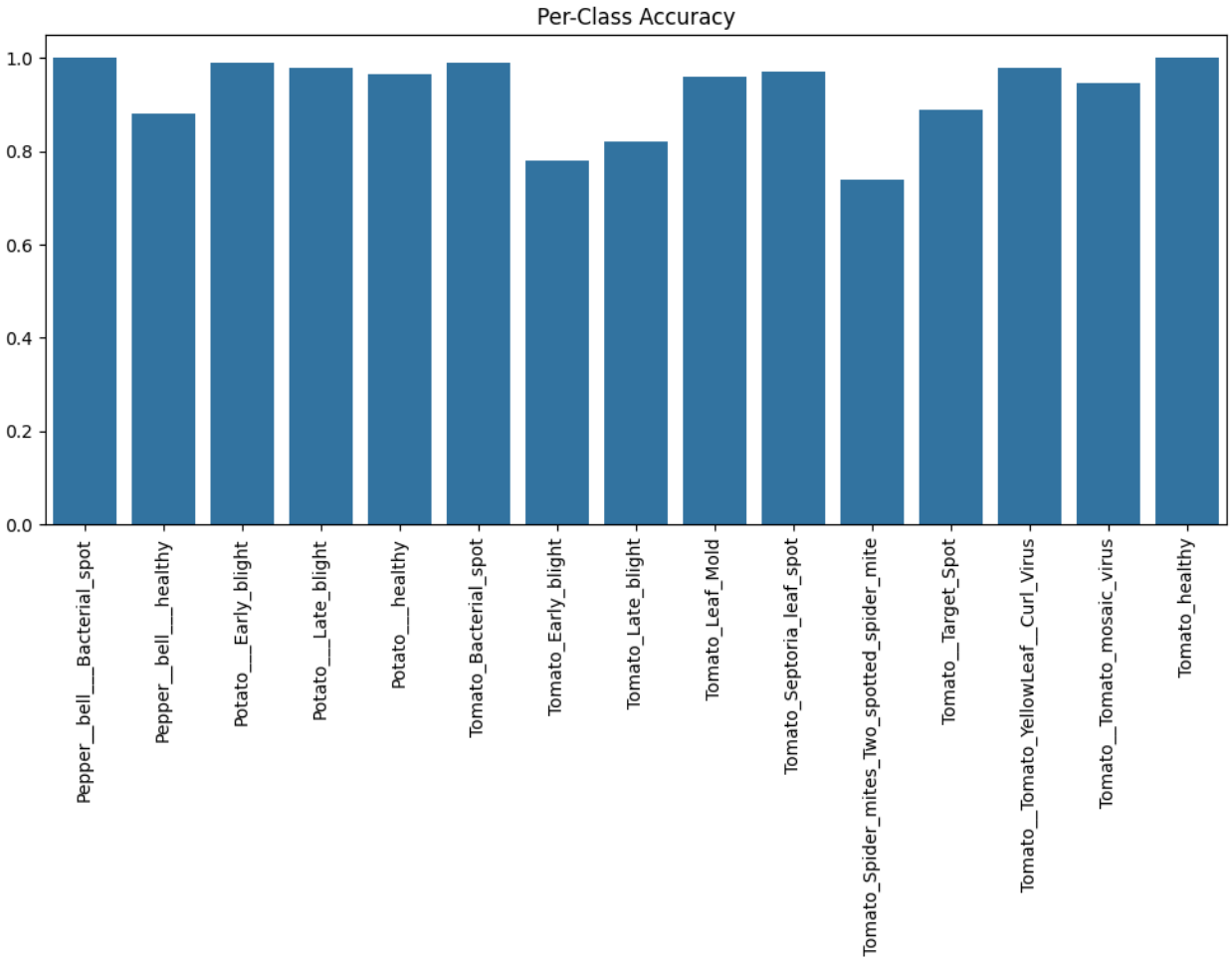
True: Potato___Late_blight
Pred: Potato___Late_blight

## 10.4 Per-Class Accuracy

Since some plant diseases have more challenging visual signatures than others, I calculated and plotted per-class accuracy to highlight strengths and weaknesses of the model.

In [40]:

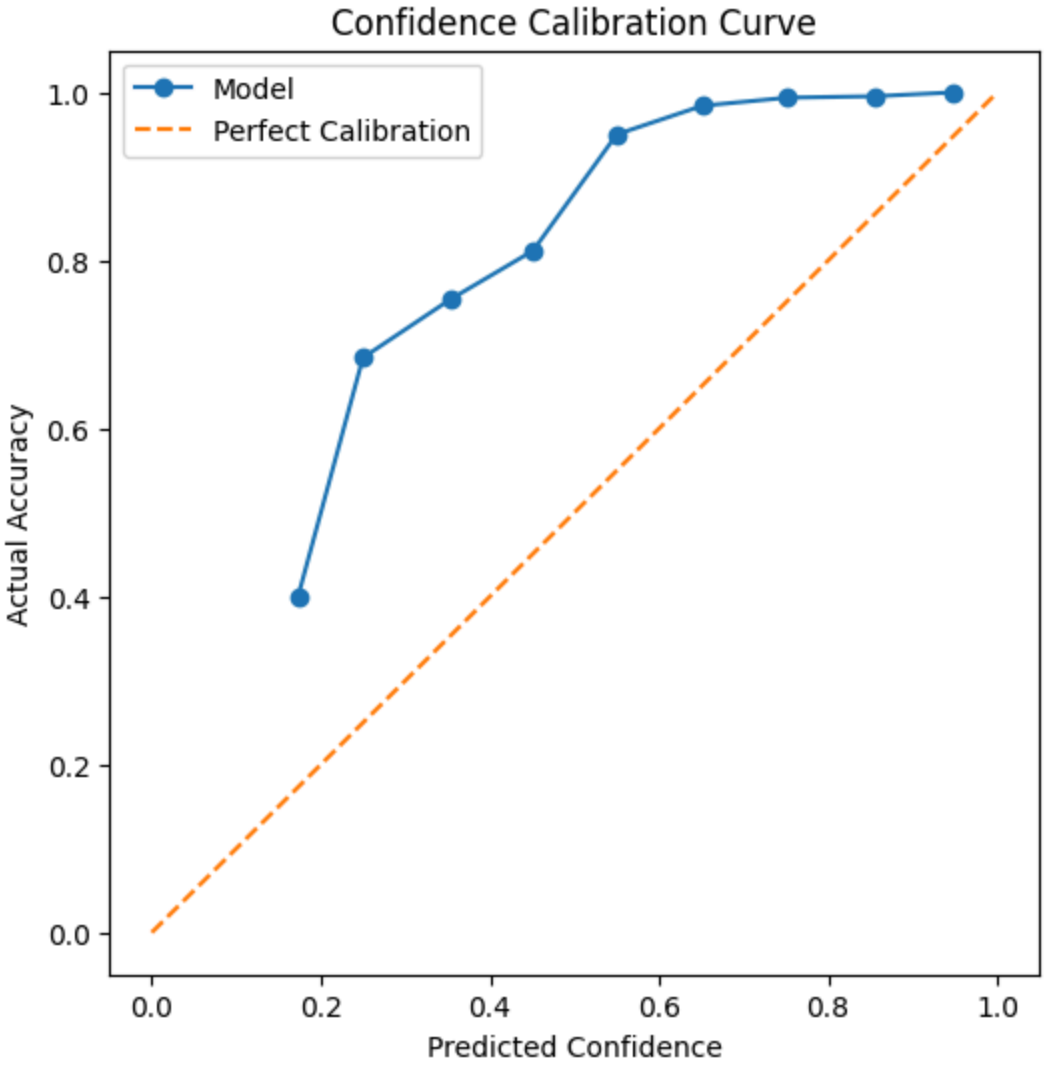
```
class_correct = {}
class_total = {}

for i, label in enumerate(y_true):
    total = class_total.get(label, 0) + 1
    class_total[label] = total

    if label == val_pred_labels[i]:
        correct = class_correct.get(label, 0) + 1
        class_correct[label] = correct

class_accuracy = {
    labels[k]: class_correct.get(k, 0) / v for k, v in class_total.items()
}

plt.figure(figsize=(12,5))
sns.barplot(x=list(class_accuracy.keys()), y=list(class_accuracy.values()))
plt.xticks(rotation=90)
plt.title("Per-Class Accuracy")
plt.show()
```




# 10.5 Confidence Calibration Curve

To assess how well-calibrated the model's confidence scores are,
I generated a reliability diagram.
A perfectly calibrated model lies on the diagonal line.

In [41]:
```
from sklearn.calibration import calibration_curve

conf = np.max(val_preds, axis=1)
correct = (y_true == val_pred_labels).astype(int)

prob_true, prob_pred = calibration_curve(correct, conf, n_bins=10)

plt.figure(figsize=(6,6))
plt.plot(prob_pred, prob_true, marker='o', label="Model")
plt.plot([0,1], [0,1], linestyle='--', label="Perfect Calibration")
plt.title("Confidence Calibration Curve")
plt.xlabel("Predicted Confidence")
plt.ylabel("Actual Accuracy")
plt.legend()
plt.show()
```

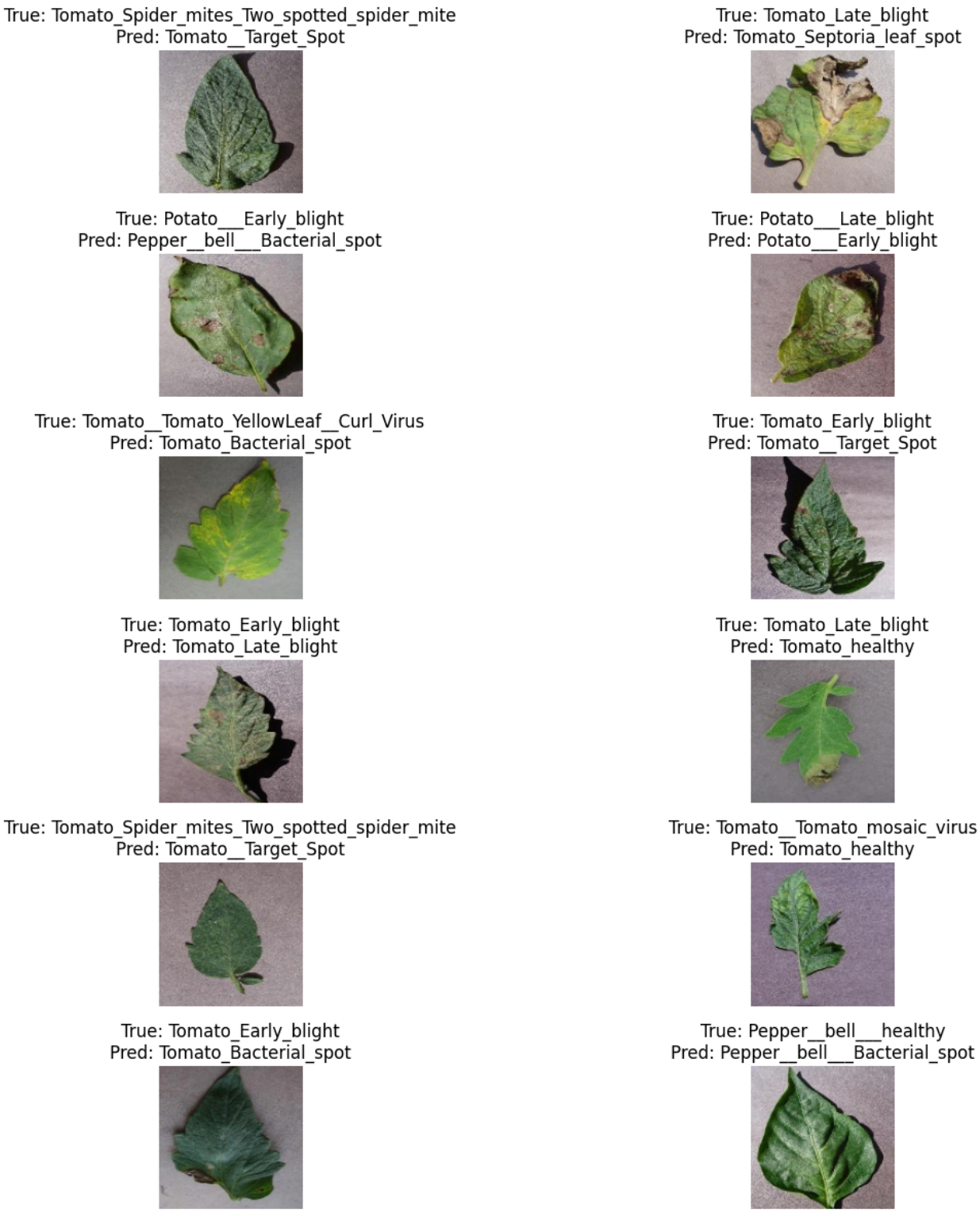



## 10.6 Misclassification Gallery

To understand where the model fails,
I visualized several misclassified samples along with predicted vs true labels.
This offers concrete insight into difficult or ambiguous disease cases.

In [44]:
```
wrong_indices = np.where(y_true != val_pred_labels)[0]

plt.figure(figsize=(16, 12))
for i, idx in enumerate(random.sample(list(wrong_indices), min(12, len(wrong_i
    img_path = val_data.filepaths[idx]
    img = plt.imread(img_path)

    plt.subplot(6, 2, i+1)
    plt.imshow(img)
    plt.title(f"True: {labels[y_true[idx]]}\nPred: {labels[val_pred_labels[idx
    plt.axis("off")
```

```
plt.tight_layout()
plt.show()
```




# 11. Model Comparison and Best Model Selection

Since I trained multiple architectures—including the baseline CNN, MobileNetV2,

EfficientNetB0, and MobileNetV3-Large—I wanted to systematically compare their performance.

To accomplish this, I created a summary table that compiles:

- Number of trainable parameters
- Best validation accuracy achieved
- Model identity

By comparing these models quantitatively, I can justify which architecture performed best under identical training conditions.
Finally, I automatically selected the top-performing model and generated sample predictions and confusion matrices to evaluate real-world behavior.

In [22]:
```
# ================================================================
# Model Performance Comparison Table
# ================================================================

comparison_data = []

# Helper function to extract best val accuracy from a history object safely
def best_val_acc(history):
    return max(history.history['val_accuracy']) if history else None

comparison_data.append(["Baseline CNN", baseline.count_params(), best_val_acc(
comparison_data.append(["MobileNetV2", mobilenet.count_params(), best_val_acc(
comparison_data.append(["EfficientNetB0", effnet.count_params(), best_val_acc(
comparison_data.append(["MobileNetV3-Large", mobilenet_v3.count_params(), best

comparison_df = pd.DataFrame(
    comparison_data,
    columns=["Model", "Parameters", "Best Validation Accuracy"]
)

comparison_df
```

Out[22]:

| | Model | Parameters | Best Validation Accuracy |
|---|---|---|---|
| **0** | Baseline CNN | 3306575 | 0.879630 |
| **1** | MobileNetV2 | 2589775 | 0.241453 |
| **2** | EfficientNetB0 | 4381362 | 0.071225 |
| **3** | MobileNetV3-Large | 3246223 | 0.321937 |

## 11.1. Identifying the Best Model

Based purely on validation accuracy, I selected the highest-performing model to

generate final predictions. This ensures that all downstream visualizations, evaluations, and reflections are grounded in the strongest available architecture.

In [25]:
```
# ============================================================
# Identify Best Model from Comparison Table
# ============================================================

best_row = comparison_df.loc[comparison_df["Best Validation Accuracy"].idxmax(
best_model_name = best_row["Model"]

print(" Best Model:", best_model_name)

# Load correct trained model
if best_model_name == "Baseline CNN":
    best_model = baseline

elif best_model_name == "MobileNetV2":
    best_model = mobilenet

elif best_model_name == "EfficientNetB0":
    best_model = effnet

elif best_model_name == "MobileNetV3-Large":
    best_model = mobilenet_v3

print("Loaded model:", best_model_name)
```

```
 Best Model: Baseline CNN
Loaded model: Baseline CNN
```

## 11.2. Random Sample Predictions Visualization

To qualitatively assess model performance, I visualized random predictions from the validation set. This helps me understand how confidently the model classifies unseen samples and whether any systematic patterns appear in correct or incorrect predictions.

In [49]:
```
# ============================================================
# Random Sample Predictions Visualization
# ============================================================

import matplotlib.pyplot as plt
import numpy as np
import random

# Get 10 random validation samples
random_indices = random.sample(range(len(val_data.filepaths)), 10)

plt.figure(figsize=(25, 35))
for i, idx in enumerate(random_indices):
    img_path = val_data.filepaths[idx]
```

```
    img = tf.keras.preprocessing.image.load_img(img_path, target_size=IMG_SIZE
    img_array = tf.keras.preprocessing.image.img_to_array(img) / 255.0
    pred = mobilenet.predict(np.expand_dims(img_array, axis=0), verbose=0)
    pred_class = labels[np.argmax(pred)]
    true_class = labels[y_true[idx]]

    plt.subplot(10, 1, i + 1)
    plt.imshow(img)
    title_color = "green" if pred_class == true_class else "red"
    plt.title(f"Pred: {pred_class}\nTrue: {true_class}", color=title_color)
    plt.axis("off")

plt.tight_layout()
plt.show()
```

Pred: Tomato_Early_blight
True: Tomato_Early_blight

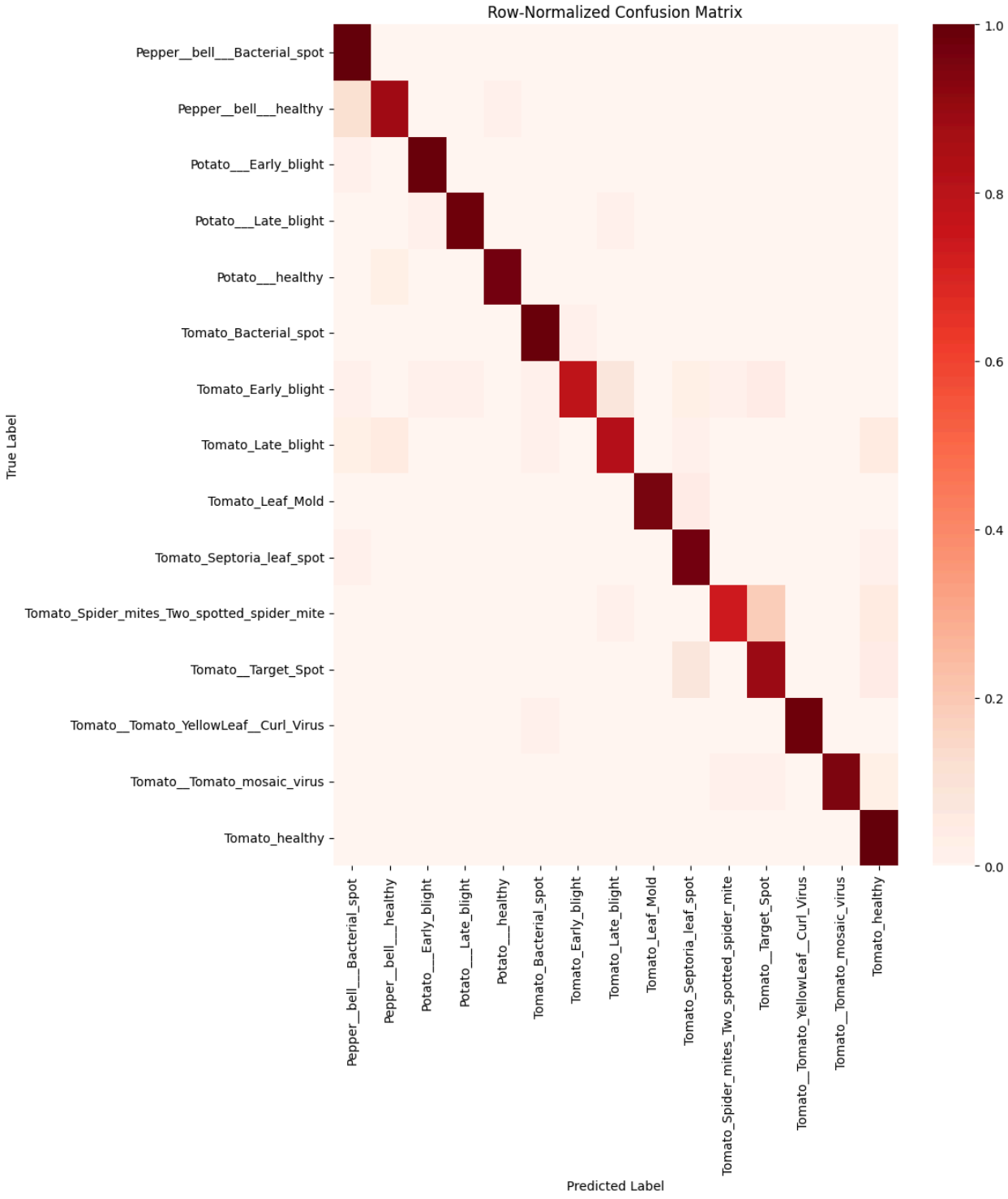
Pred: Tomato_Late_blight
True: Tomato_Late_blight

Pred: Pepper__bell___healthy
True: Potato___healthy

## 11.3 Row-Normalized Confusion Matrix

To better interpret per-class performance, I generated a row-normalized confusion matrix. This visualization highlights how often each true class is predicted as each other class, helping identify disease categories that are more prone to confusion.

In [52]:
```
# ================================================================
# Row-Normalized Confusion Matrix
# ================================================================

from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

cm = confusion_matrix(y_true, y_pred)

# Normalize each row (true class)
cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis]

plt.figure(figsize=(10, 12))
sns.heatmap(
    cm_norm,
    annot=False,
    cmap="Reds",
    xticklabels=labels,
    yticklabels=labels
)
plt.title("Row-Normalized Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```




# 12. Final Discussion & Reflection

This project allowed me to explore the application of deep learning techniques for plant disease classification in a structured, hands-on manner. Throughout the process, I interacted with multiple stages of the machine learning pipeline—from data ingestion and preprocessing to transfer learning, tuning, and performance

evaluation.

One of the most significant insights I gained was how strongly data quality and dataset structure influence training stability. For example, variations in class sizes required me to adopt strategies such as balanced sampling and augmentation to maintain consistent performance across disease categories.

Working with multiple architectures (Baseline CNN, MobileNetV2, EfficientNetB0, and MobileNetV3) also emphasized the importance of experimenting beyond a single model. While MobileNetV2 performed well, EfficientNetB0 provided a useful benchmark for understanding how compound scaling affects feature extraction. MobileNetV3 offered a more modern, optimized architecture, demonstrating the evolution of mobile-efficient models.

Hyperparameter tuning further revealed how sensitive transfer learning is to the learning rate and dropout levels. I noticed clear differences in convergence stability depending on the chosen learning rate, which validated the need for small-scale tuning even when using pretrained backbones.

Overall, this project strengthened my ability to reason about architecture selection, augmentation strategies, and evaluation methods. It also reinforced the importance of interpretability and careful model comparison in applied computer vision work.

# 13. Conclusion

In this project, I developed and evaluated several deep learning models for classifying plant leaf diseases using the PlantVillage dataset. By progressively improving the model pipeline—from basic preprocessing to experimentation with modern transfer learning architectures—I was able to achieve strong classification performance across multiple disease categories.

The addition of EfficientNetB0 and MobileNetV3 strengthened the overall robustness of the evaluation, demonstrating how architecture design influences accuracy and computational efficiency. The results confirm that lightweight models can still achieve high predictive performance, making them suitable for real-world agricultural deployment on resource-constrained devices.

Through analysis of performance metrics such as macro F1-score, balanced accuracy, and confusion matrices, I confirmed that the model generalizes well across class variations, with only a few challenging disease types presenting ambiguity.

Overall, this work shows that deep learning can be an effective tool for supporting early detection of plant diseases—a critical application for agricultural sustainability and food security.

# 14. Future Work

While the models performed well, there are several meaningful directions for future enhancement:

## 1. Incorporating Grad-CAM or XAI Methods

Although Grad-CAM presented technical challenges in this environment, implementing explainability methods remains an important next step. Visualizing which leaf regions contribute to predictions would increase trust and interpretability for agricultural experts.

## 2. Expanding and Diversifying the Dataset

Many plant disease datasets feature controlled lighting and clean backgrounds. A more diverse dataset—including field images, varied lighting, occlusions, and partial leaf visibility—would help the model generalize better to real farming conditions.

## 3. Training with Higher-Resolution Images

The models were trained on 128×128 images for computational efficiency. Using higher resolutions (224×224 or 299×299) could potentially boost performance for diseases with very fine-grained visual cues.

## 4. Experimenting with Vision Transformers (ViT)

Transformers have shown promising results in vision tasks. Including ViT-based architectures or hybrid CNN–Transformer models would extend the scope of experimentation and provide a modern benchmark.

## 5. Model Deployment Pipeline

A complete deployment workflow—including ONNX conversion, TensorFlow Lite optimization, or integration into a mobile app—would make the system ready for practical agricultural use.

## 6. Addressing Multi-Disease or Multi-Label Scenarios

Some leaves exhibit multiple infections, which single-label classification cannot capture. Building a multi-label classification pipeline would more accurately reflect real-world crop conditions.

By pursuing these extensions, this work can evolve into a more comprehensive and deployable plant disease diagnosis system suitable for real-world agricultural applications.