Potato Disease Classification Using Deep Learning (CNN)

Overview

Potatoes are one of the most important food crops globally, yet farmers face significant **economic losses** each year due to various **leaf diseases** that affect plant health and yield. Two of the most common and destructive diseases are **Early Blight** and **Late Blight**, both of which can rapidly spread under favorable environmental conditions, severely impacting production quality and quantity.

- **Early Blight** is caused by a fungal pathogen (*Alternaria solani*), leading to brown lesions and defoliation of potato leaves.
- **Late Blight**, caused by *Phytophthora infestans*, is a microorganism-based infection responsible for the infamous Irish Potato Famine and continues to threaten crop yields today.

Problem Context

Most small-scale farmers **lack advanced disease detection tools**, relying solely on **manual observation**, which can be inaccurate and delayed. Early detection plays a **crucial role** in preventing disease spread and minimizing losses. Identifying whether a potato leaf is healthy, infected by early blight, or infected by late blight at the **right stage** allows for appropriate and timely treatment.

Business Scenario

AtliQ Agriculture, an AI-driven agritech company, has identified this problem and initiated the development of a **mobile application** capable of classifying potato plant health from an image.

The goal is to empower farmers with a **simple yet powerful AI-based solution** where they can:

1. Capture an image of a potato plant leaf using their smartphone.
2. Instantly receive feedback on whether the plant is **healthy**, **affected by early blight**, or **affected by late blight**.

This system leverages **Deep Learning** and **Convolutional Neural Networks (CNNs)** to analyze visual patterns and accurately detect diseases.

Core Objective

To **design, develop, and deploy an end-to-end deep learning system** that:

- Classifies potato plant leaves into **three categories** — *Healthy*, *Early Blight*, and *Late Blight*.
- Integrates seamlessly into a **mobile and web application** accessible to farmers.
- Utilizes **cloud deployment** and **model optimization** techniques for real-time performance and scalability.

Project Scope

This end-to-end system includes the following major components:

Stage	Description
1. Data Collection	Gathering images of potato leaves (healthy, early blight, late blight) from trusted datasets and online repositories.
2. Data Preprocessing & Augmentation	Cleaning, resizing, and augmenting the dataset using **TensorFlow Datasets (TFDS)** to improve model generalization.
3. Model Building (CNN)	Developing a **Convolutional Neural Network** for image classification using **TensorFlow/Keras**.
4. Model Serving (MLOps)	Using **TensorFlow Serving** with **FastAPI** for efficient model deployment and API creation.
5. Frontend Development	Building a **ReactJS** web app for testing and a **React Native** mobile app for farmers.
6. Model Optimization & Deployment	Converting the trained model into **TensorFlow Lite (TFLite)** format for mobile efficiency, and deploying it on **Google Cloud Platform (GCP)**.

Expected Impact

✅ **Economic Benefits:** Early detection minimizes disease spread and reduces yield loss.

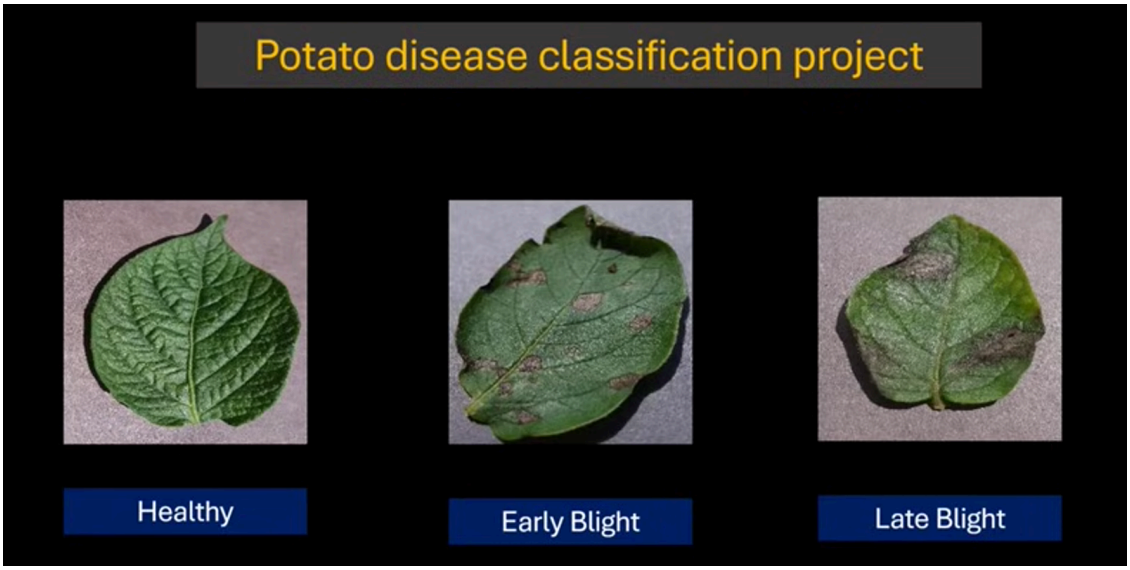
✅ **Accessibility:** Farmers can diagnose leaf conditions using only a smartphone camera.

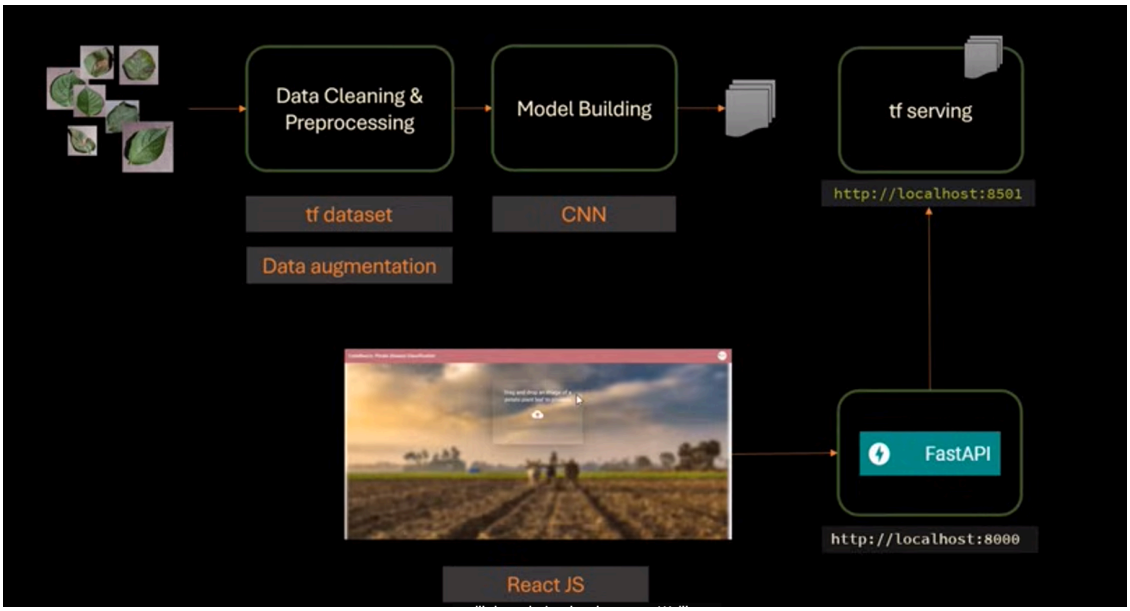
✅ **Scalability:** The system architecture supports expansion to other crops and diseases.

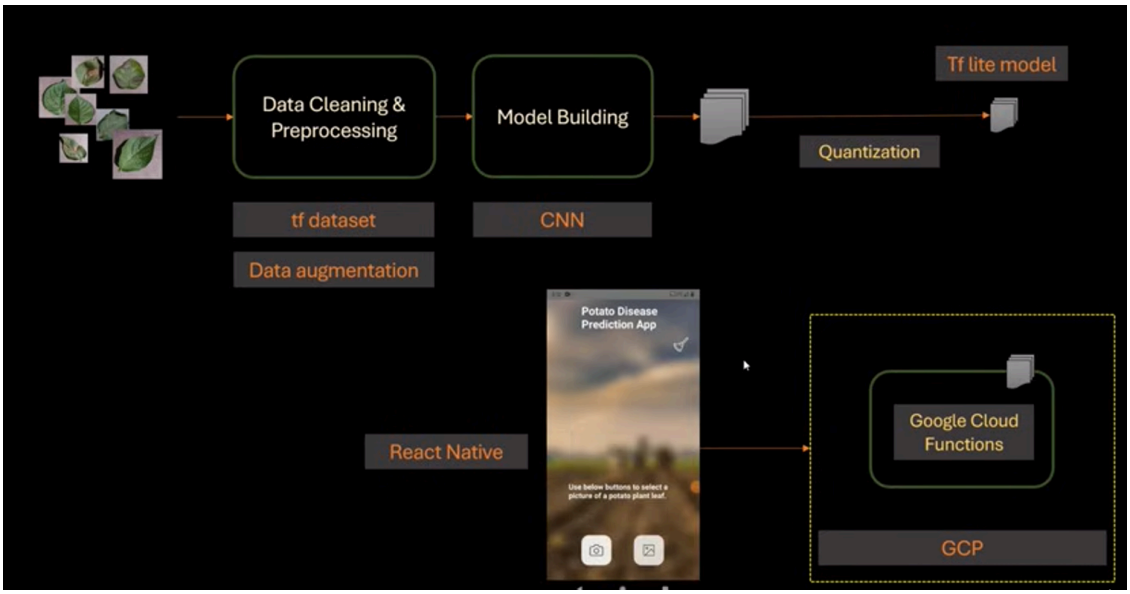
✅ **Innovation in Agriculture:** Demonstrates how AI and Deep Learning can drive practical solutions in the agri-tech domain.

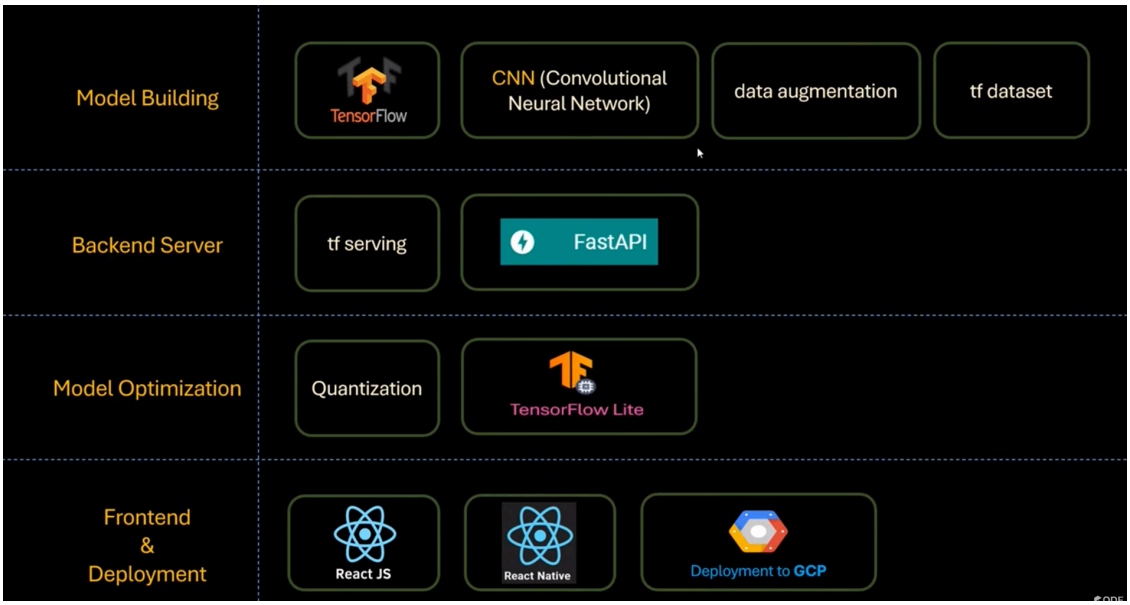
Summary

This project represents a **real-world application of AI in agriculture**, combining **Deep Learning, Cloud Computing, and Mobile Development** to deliver a **complete, production-ready solution**. Beyond the technical aspect, it highlights the transformative potential of machine learning in supporting sustainable farming and enhancing food security.




Data Cleaning & Preprocessing
Model Building
tf serving
http://localhost:8501
tf dataset
CNN
Data augmentation
FastAPI
http://localhost:8000
React JS


Data Cleaning & Preprocessing
Model Building
Tf lite model
Quantization
tf dataset
CNN
Data augmentation
Potato Disease Prediction App
Use below buttons to select a picture of a potato plant leaf.
React Native
Google Cloud Functions
GCP



🧩 2. Data Collection & Preprocessing

Core Theme

This stage lays the foundation of the entire deep learning pipeline. The quality, diversity, and organization of the dataset directly determine how well the model generalizes to unseen data. Here, we focus on **acquiring, preparing, and structuring image data** to train a robust **CNN-based potato disease classification model** using TensorFlow.

2.1 Data Acquisition

AtliQ Agriculture identified **three practical approaches** to collect leaf image data for model training:

Option	Description	Pros	Cons
1. Ready-made Datasets (Kaggle, etc.)	Use open-source datasets available online such as Kaggle's *PlantVillage* dataset.	Quick, cost-effective, well-structured, high-quality.	May contain unrelated categories (e.g., tomato, pepper).

Option	Description	Pros	Cons
2. Manual Data Collection & Annotation	Create an in-house dataset with help from farmers and domain experts who manually label images as “Healthy,” “Early Blight,” or “Late Blight.”	High relevance, realistic, domain-specific.	Time-consuming and expensive; requires annotation team and field work.
3. Web Scraping + Annotation Tools	Use web scraping scripts to collect images from multiple agricultural websites, then annotate them using tools like **Doccano**.	Flexible, scalable, custom dataset creation.	Risk of poor-quality images, annotation overhead.

👉 **In this project**, we used the **PlantVillage Kaggle dataset**, which includes images for multiple crops and diseases. Non-potato categories were removed manually, retaining only:

- **Potato___Healthy**
- **Potato___Early_Blight**
- **Potato___Late_Blight**

These images form the **core dataset** for our CNN model.

2.2 Dataset Preparation and Directory Setup

After downloading and extracting the Kaggle dataset:

1. Deleted all non-potato directories (e.g., tomato, pepper).
2. Moved the remaining three folders into a local project directory:

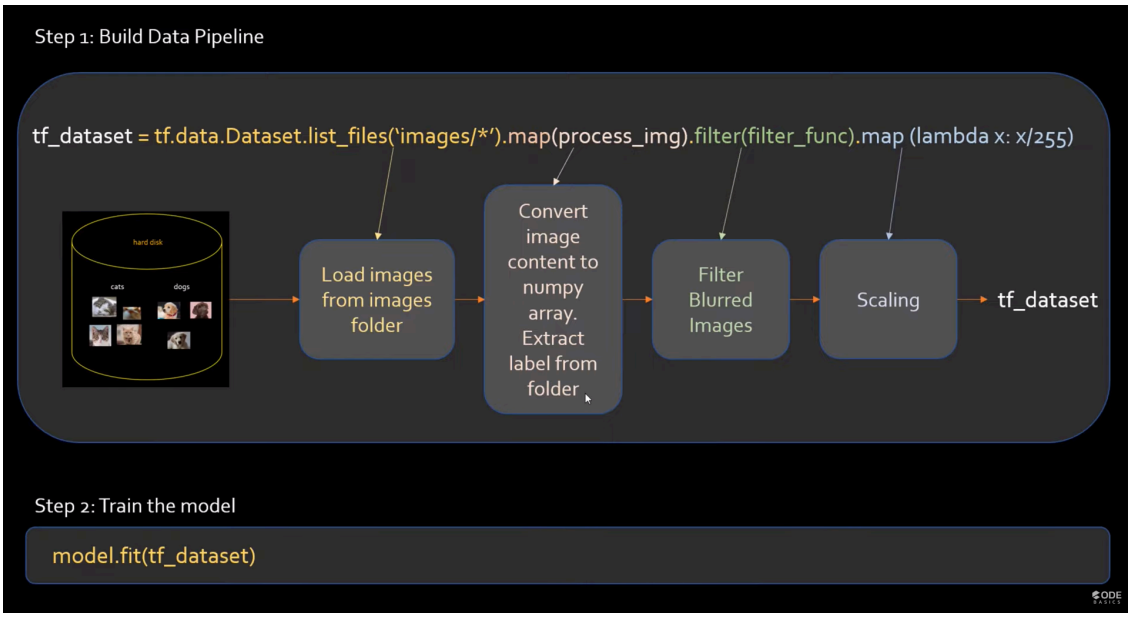
```
potato_disease/
    ├── Potato___Early_Blight/
    ├── Potato___Late_Blight/
    ├── Potato___Healthy/
```

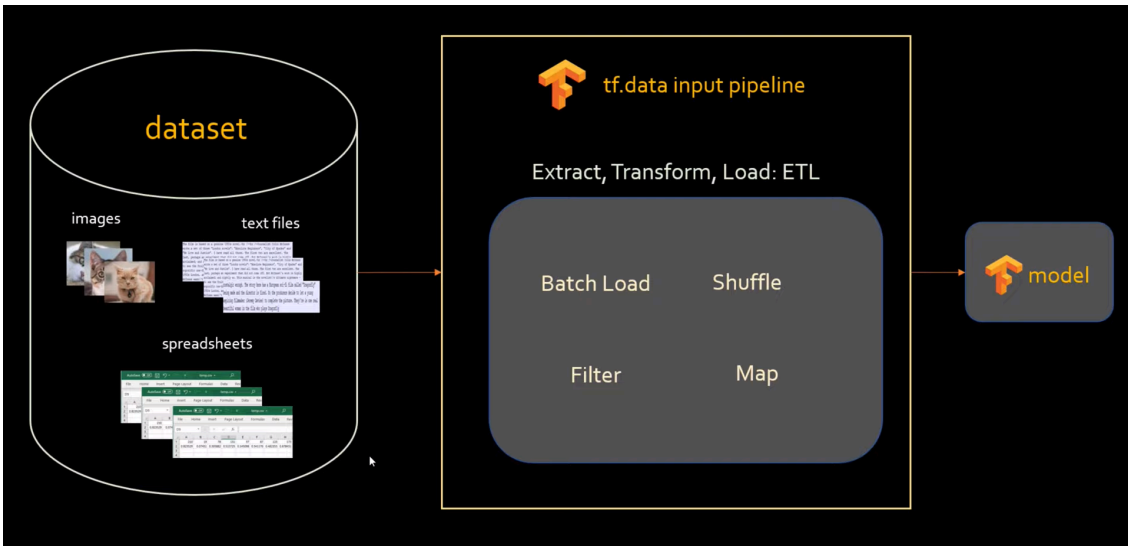
3. Created a **Jupyter Notebook** for experimentation and a **training** subfolder to store model-related files.

Each class folder contained:

- Around **1,000+ images per class** (RGB format, 256×256 pixels).
- File format: `.jpg`, `.png`.

2.3 Loading Dataset into TensorFlow


Step 1: Build Data Pipeline
tf_dataset = tf.data.Dataset.list_files('images/*').map(process_img).filter(filter_func).map (lambda x: x/255)
hard disk
cats
dogs
Load images from images folder
Convert image content to numpy array. Extract label from folder
Filter Blurred Images
Scaling
tf_dataset
Step 2: Train the model
model.fit(tf_dataset)


dataset
images
text files
spreadsheets
tf.data input pipeline
Extract, Transform, Load: ETL
Batch Load
Shuffle
Filter
Map
model

📌 Core Idea

The **TensorFlow input pipeline**, built using the `tf.data` API, is designed to efficiently load, preprocess, and stream large datasets during deep learning model training. It prevents memory issues and allows clean, scalable data transformations.

🎯 Why Do We Need an Input Pipeline?

✅ 1. Memory Efficiency

- Loading millions of images into RAM (e.g., 10M images on 8GB RAM) will crash your system.
- Pipelines stream data **in batches**, so only a small chunk is in memory at a time.

✅ 2. Data Transformation

`tf.data.Dataset` supports:

- `filter()` — remove unwanted samples
- `map()` — preprocessing, scaling, augmentation etc.
- `shuffle()` — randomize order

- `batch()` — create batches for training

✅ 3. Single-Line Data Pipeline

Multiple transformation steps can be chained:

```
dataset = files.map(load_img).filter(filter_blurry).map(scale).batch(32)
```

Clean, readable, modular.

✅ 4. Works With All Data Types

- Images
- Text
- CSV / Excel
- Cloud data (S3, GCS, Azure Blob Storage)

✅ 5. Direct Training Integration

The final dataset can be fed directly into:

```
model.fit(dataset)
```

✅ How tf.data.Dataset Works

1. Streaming Approach (Batch Loading)

Instead of loading everything at once:

- Load **batch 1**, train
- Load **batch 2**, train
- Continue...

This avoids memory overflow.

2. Special TensorFlow Data Structure

`tf.data.Dataset` acts like:

- A wrapper over your NumPy arrays / tensors
- A lazy loader that brings batches on demand

✅ Common Transformations

✅ Filtering

Remove unwanted samples (example: negative numbers, blurry images)

```
dataset = dataset.filter(lambda x: x > 0)
```

✅ Mapping

Apply a custom function to each element:

Example: scaling image pixels

```
dataset = dataset.map(lambda x: x / 255)
```

✅ Shuffling

Randomize order to remove bias:

```
dataset = dataset.shuffle(buffer_size=100)
```

📌 **Buffer size logic**

TF creates a sliding window of `buffer_size` items and randomly draws from it (StackOverflow analogy).

✅ Batching

```
dataset = dataset.batch(32)
```

Used for:

- Efficient GPU utilization
- Multi-GPU distributed training

✅ Chaining Transformations (Pipeline)

All steps can be chained:

```
dataset = (
    dataset
    .filter(lambda x: x > 0)
    .map(lambda x: x * 72)
    .shuffle(3)
    .batch(2)
)
```

This is the essence of a TensorFlow input pipeline.

✅ Working with Images

Step 1: Load Image Paths

```
dataset = tf.data.Dataset.list_files("images/*/*", shuffle=False)
```

Step 2: Shuffle

```
dataset = dataset.shuffle(200)
```

Step 3: Train-Test Split

```
train_ds = dataset.take(train_size)
test_ds  = dataset.skip(train_size)
```

Extract Label From File Path

Example:

`images/cat/img1.jpg` → label = "cat"

```
label = tf.strings.split(file_path, os.path.sep)[-2]
```

Process Each Image

```
def process_image(file_path):
    label = get_label(file_path)
    img = tf.io.read_file(file_path)
    img = tf.image.decode_jpeg(img)
    img = tf.image.resize(img, (128,128))
```

```
    return img, label
```

Apply:

```
train_ds = train_ds.map(process_image)
```

Scaling Images

```
def scale(img, label):
    return img / 255.0, label

train_ds = train_ds.map(scale)
```

✅ Final Image Pipeline Example

```
train_ds = (
    tf.data.Dataset.list_files("images/*/*")
    .shuffle(200)
    .map(process_image)
    .map(scale)
    .batch(32)
)
```

✅ ETL (Extract–Transform–Load) in tf.data

Extract

- Load file paths
- Read images/files
- Load text/CSV

Transform

- Clean
- Filter
- Resize
- Normalize
- Augment

Load

- Create batches
- Feed into `model.fit()`

To efficiently handle image batches, we leveraged **TensorFlow's data pipeline** using:

```
tf.keras.preprocessing.image_dataset_from_directory()
```

This API:

- Reads all images from the directory structure.
- Automatically assigns labels based on folder names.
- Converts them into a **tf.data.Dataset** object.
- Supports shuffling, batching, and resizing on the fly.

Configuration Used:

```
IMG_SIZE = (256, 256)
BATCH_SIZE = 32
```

```
dataset = tf.keras.preprocessing.image_dataset_from_directory(
    "PlantVillage",
    shuffle=True,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE
)
```

✅ Output: Loaded **2152 images** belonging to **3 classes**, batched into **68 batches of 32 images each**.

2.4 Data Visualization & Exploration

To verify data integrity, several samples from each class were visualized using **Matplotlib**:

```
plt.figure(figsize=(10,10))
for images, labels in dataset.take(1):
    for i in range(12):
        ax = plt.subplot(3, 4, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```

This step confirmed:

- Image clarity and resolution consistency.
- Proper class labeling.
- Visual variation among samples (e.g., color, lighting).

2.5 Dataset Splitting

To ensure fair model training and evaluation, the dataset was divided into:

- **80% → Training set**

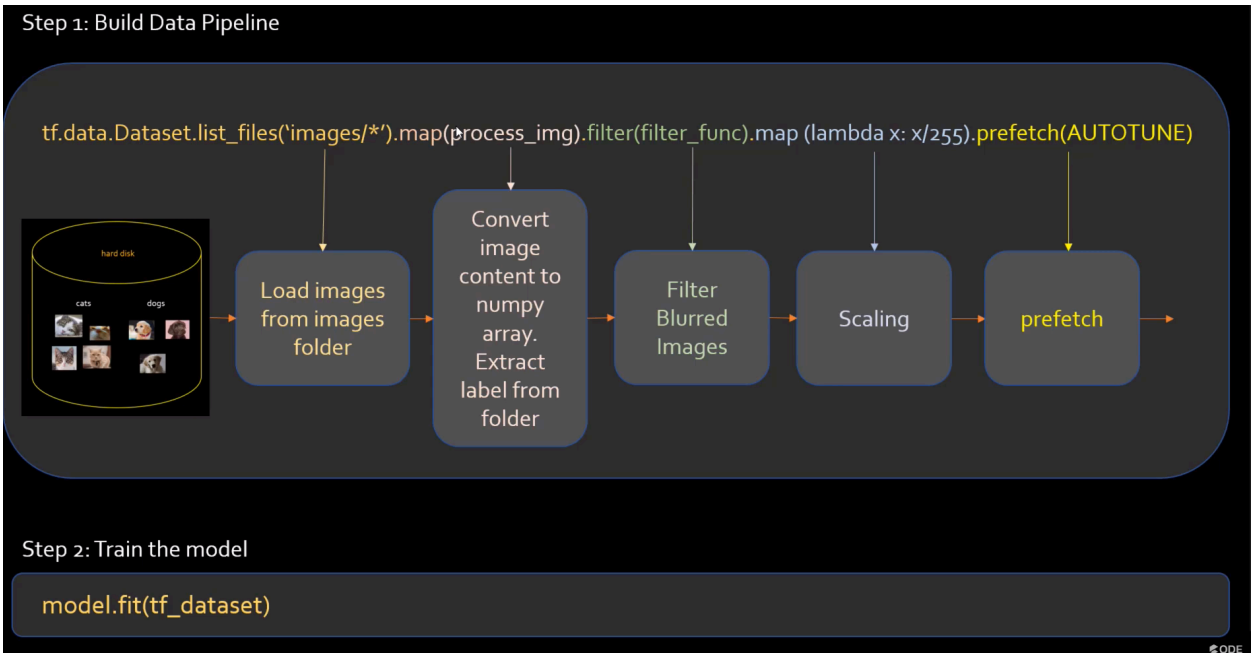
- **10% → Validation set**
- **10% → Test set**

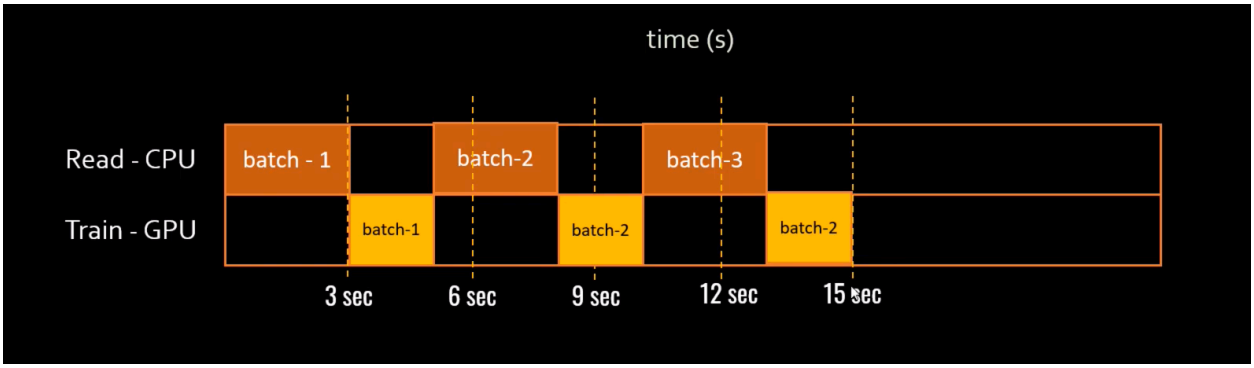
Unlike `sklearn`'s `train_test_split`, TensorFlow uses the dataset's internal APIs:

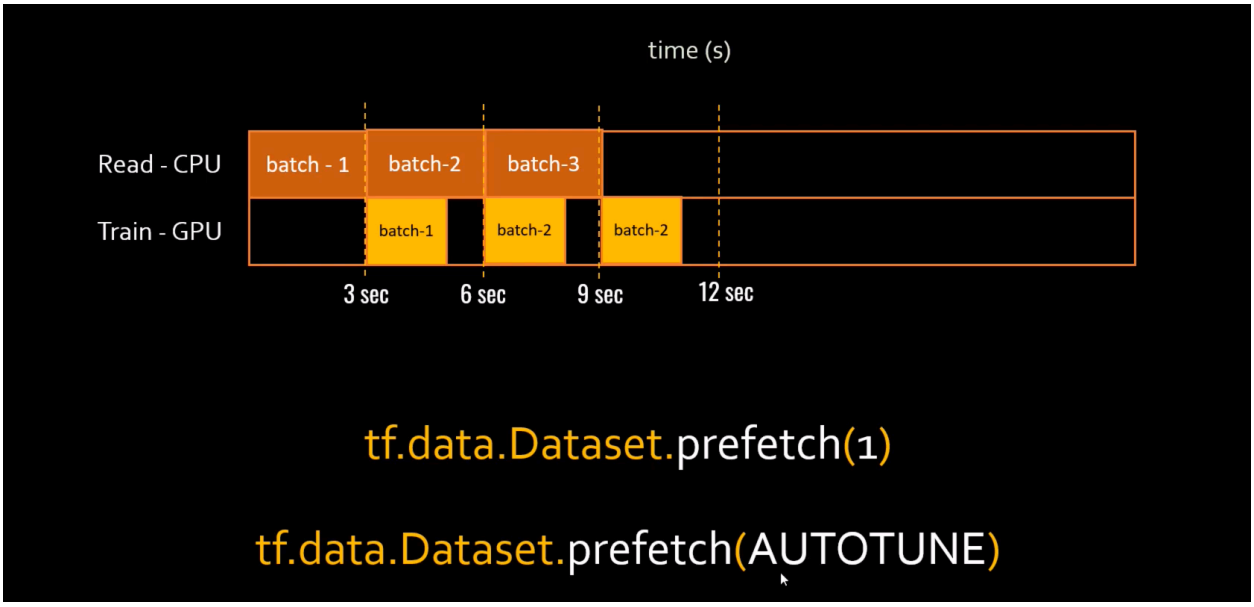
```
train_ds = dataset.take(54)
val_test_ds = dataset.skip(54)
val_ds = val_test_ds.take(7)
test_ds = val_test_ds.skip(7)
```

This maintains **shuffling integrity** and supports **batched processing**.

2.6 Data Pipeline Optimization


Step 1: Build Data Pipeline
tf.data.Dataset.list_files('images/*').map(process_img).filter(filter_func).map (lambda x: x/255).prefetch(AUTOTUNE)
hard disk
cats
dogs
Load images from images folder
Convert image content to numpy array. Extract label from folder
Filter Blurred Images
Scaling
prefetch
Step 2: Train the model
model.fit(tf_dataset)


time (s)
Read - CPU
Train - GPU
batch - 1
batch-2
batch-3
batch-1
batch-2
batch-2
3 sec
6 sec
9 sec
12 sec
15 sec


time (s)
Read - CPU
Train - GPU
batch - 1
batch-2
batch-3
batch-1
batch-2
batch-2
3 sec
6 sec
9 sec
12 sec
tf.data.Dataset.prefetch(1)
tf.data.Dataset.prefetch(AUTOTUNE)

Core Theme

Efficient deep learning training requires maximizing hardware utilization. TensorFlow's `tf.data` API provides two powerful tools—**prefetch** and **cache**—that significantly speed up training by improving how data flows from storage → CPU → GPU.

✅ Key Concepts

1. Prefetching

- **Goal:** Overlap **CPU data preparation** with **GPU model training**.
- Without prefetch:
  - GPU waits idle while CPU loads the next batch.
- With prefetch:
  - While GPU trains on batch *n*, CPU prepares batch *n+1*.
- This reduces GPU idle time and accelerates training.
- **Usage:**

```
dataset = dataset.prefetch(tf.data.AUTOTUNE)
```

- `AUTOTUNE` allows TensorFlow to choose the optimal prefetch buffer size automatically.

---

2. Caching

- **Goal:** Avoid recomputing expensive preprocessing across epochs.
- Problem without cache:
  - For every epoch: the dataset is read → parsed → transformed → mapped again.
  - This wastes time, especially for large static datasets.

- With cache:
  - After the **first epoch**, the preprocessed dataset is stored in memory (or a file).
  - Subsequent epochs reuse cached data instantly.
- **Usage:**

```
dataset = dataset.cache()
```

✅ How They Improve Performance

Prefetch Solves: CPU–GPU Parallelism

- Enables both CPU and GPU to work at the same time.
- Avoids GPU sitting idle waiting for data.
- Significant time reduction during training, especially for large batches.

Cache Solves: Redundant Computation Across Epochs

- During multi-epoch training, heavy preprocessing is done **only once**.
- All subsequent epochs skip reading/mapping overhead.
- HUGE gain when:
  - You have expensive map functions (image decoding, augmentation).
  - Dataset fits in memory.

✅ Deep Insight

Real performance gains in deep learning come not only from faster models, but also from **efficient data pipelines**.

- **Prefetch** maximizes throughput by overlapping stages.
- **Cache** removes redundant work in multi-epoch training.

- Combined, they dramatically reduce model training time and GPU idle time.

These techniques are essential when working with:

- Large datasets
- Performance-critical production pipelines
- Resource-constrained environments
- Iterative training loops

To enhance GPU training efficiency, **caching** and **prefetching** were applied:

```
AUTOTUNE = tf.data.AUTOTUNE
train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

- **Cache:** Stores previously read images in memory to avoid repeated disk reads.
- **Prefetch:** Loads upcoming batches while the GPU is training on the current batch.

These steps significantly reduce I/O bottlenecks and increase throughput during training.

2.7 Data Preprocessing

Each image undergoes basic normalization before feeding into the CNN:

- **Resizing:** Ensures every image is standardized to `(256, 256, 3)`.
- **Rescaling:** Converts pixel values from 0–255 to 0–1 range.

TensorFlow preprocessing layer:

```
resize_and_rescale = tf.keras.Sequential([
    tf.keras.layers.Resizing(256, 256),
```

```
    tf.keras.layers.Rescaling(1./255)
])
```

2.8 Data Augmentation

To make the model more **robust** against overfitting and environmental variations (like angle, brightness, or rotation), **data augmentation** techniques were applied using TensorFlow's built-in layers:

```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal_and_vertical"),
    tf.keras.layers.RandomRotation(0.2)
])
```

These transformations help the model generalize to real-world scenarios by synthetically increasing training diversity — e.g., flipped, rotated, or slightly altered versions of the same image.

2.9 Summary of the Data Pipeline

Step	Technique/Tool	Purpose
Data Source	Kaggle (PlantVillage)	Ready-made high-quality dataset
Loader	`tf.keras.preprocessing.image_dataset_from_directory()`	Efficient directory-based image loading
Splitting	`take()`, `skip()` methods	80/10/10 partition
Optimization	`cache()`, `prefetch()`	Faster data access & GPU utilization
Normalization	Rescaling 0–255 → 0–1	Standardize pixel intensity
Augmentation	Random Flip, Rotation	Enhance robustness, prevent overfitting

2.10 Key Insights

- Efficient use of **TensorFlow's** `tf.data.Dataset` API enables high-throughput image loading and real-time augmentation.
- Proper **data splitting** ensures fair model evaluation and generalization.
- **Performance optimization** (cache/prefetch) is essential when using GPUs for large datasets.
- Incorporating **data augmentation** significantly improves model robustness to unseen data conditions.

2.11 Practical Takeaway

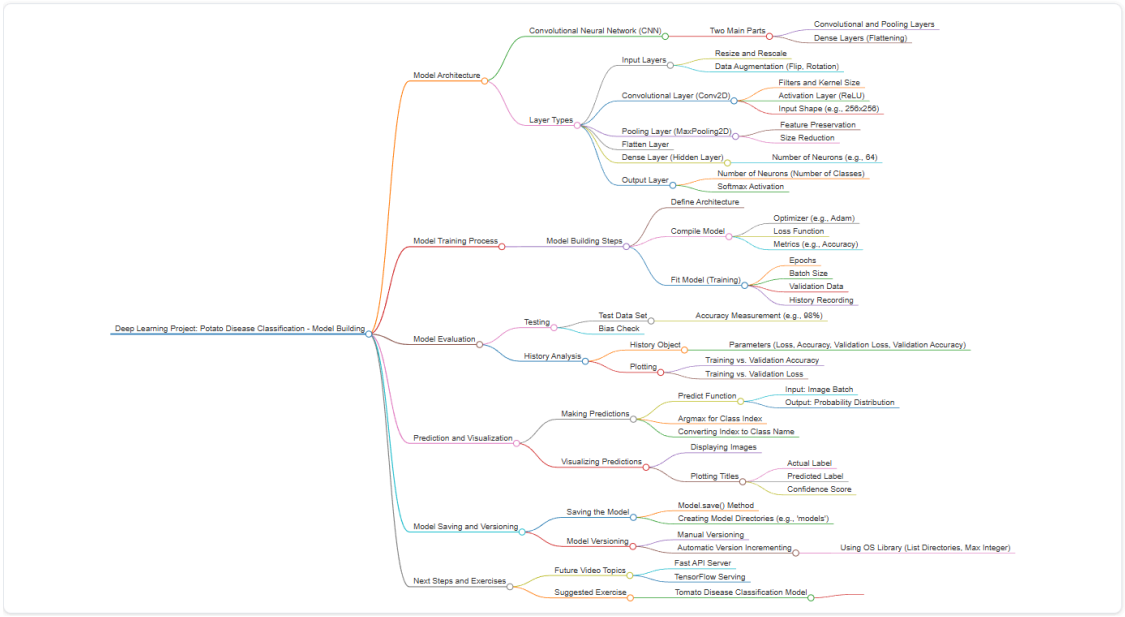
This phase demonstrates a **hands-on and scalable data preparation approach**, transforming raw Kaggle images into a **high-performance TensorFlow pipeline** — the foundation upon which the CNN model will be built and trained.

3. Model Building

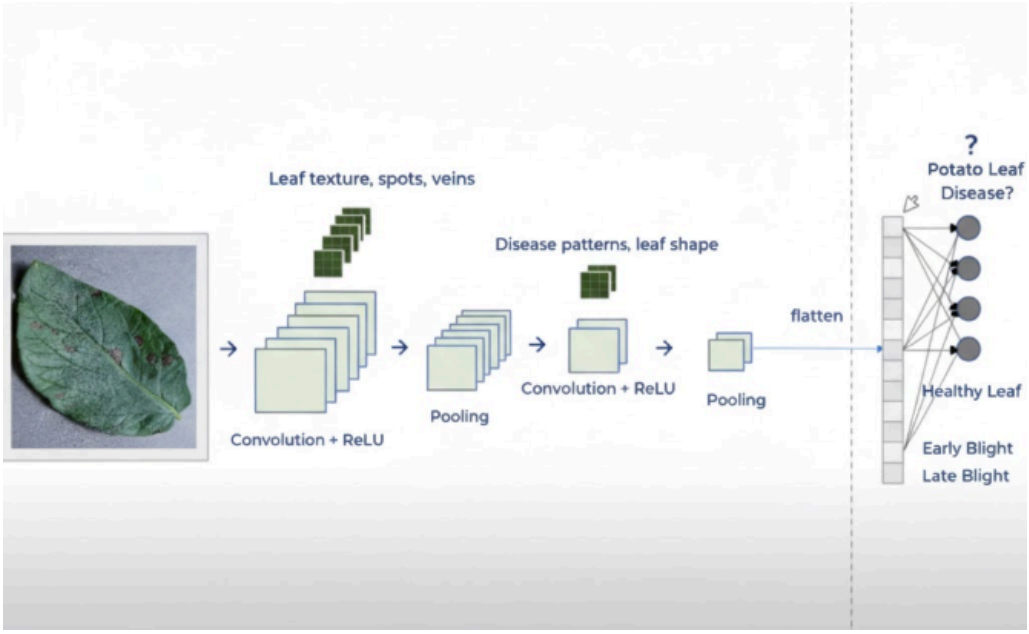
1. Overview of the Workflow

Complete **end-to-end deep learning pipeline**:

- Building a **Convolutional Neural Network (CNN)** architecture.
- Training the network on a **training dataset**.
- Evaluating performance using a **validation dataset** during training.
- Measuring final performance using an unseen **test dataset**.
- Visualizing training history (accuracy & loss curves).
- Running predictions on images and analyzing confidence scores.
- Saving the trained model to disk.
- Implementing **model versioning** (auto-increment model versions).
- Preparing the model for future use in a **FastAPI server** or **TensorFlow Serving**.



2. CNN Architecture Explained



A **Convolutional Neural Network** is extremely effective for image classification.

The video/text explains that a CNN typically has two major parts:

A) Feature Extraction Layers

These layers automatically learn to detect features like edges, textures, shapes, and object parts.

Includes:

✅ **Convolutional layers**

✅ **Activation functions (e.g., ReLU)**

✅ **Pooling layers (MaxPooling)**

✅ **Data augmentation** (for generalization)

✅ **Resizing & normalization** (pre-processing)

B) Classification Layers

After feature extraction, the output is flattened and fed into:

✅ **Dense (fully connected) layers**

✅ **Final output layer** with **softmax** for probability distribution across classes.

---

3. Preprocessing Steps

Before images reach the CNN:

✔ Resize

All input images are resized to a uniform size, e.g., **256 × 256 pixels**.

✔ Rescale

Pixel values (0–255) are scaled to (0–1) by dividing by 255.

This helps gradient descent converge faster.

✔ Data Augmentation

Common augmentations:

- Horizontal flip
- Vertical flip

- Random rotation
- Random zoom
- Random shift

Purpose:

- Artificially increase dataset size
- Reduce overfitting
- Improve generalization

---

4. Convolutional Layers Deep Explanation

A convolutional layer consists of several **filters (kernels)**.

For example:

- **32 filters**
- Kernel size **3 × 3**

Each filter learns a different pattern:

Filter	Learns to Detect
#1	vertical edges
#2	horizontal edges
#3	curves
#4	textures
...	eyes, noses, ears, patterns, shapes

Through **backpropagation**, the network automatically learns the best filters.

➡️ You don't manually create filters like "eye detector" or "nose detector".

---

5. Max Pooling Deep Explanation

After convolution:

✅ Reduces spatial size

✅ Preserves important features

✅ Reduces computation & overfitting

Example: MaxPool size = **2 × 2**

```
[3, 5]
[2, 9]
```

Max = 9 → kept

Other values → discarded

Effect:

- Reduces image dimension by 50%
- Keeps strongest spatial activations

---

6. Flattening and Dense Layers

After stacking many convolution + pooling layers, the output is a 3D tensor.

Example: `(8, 8, 64)`

Flatten converts this into:

`8 × 8 × 64 = 4096` neurons

Then:

- Dense layer with 64 neurons
- Dense output layer = number of classes (e.g., 3 for potatoes)

Final layer uses **softmax**, which outputs a probability distribution:

```
[0.88, 0.09, 0.03]
```

Meaning:

- Class 0: 88% chance

- Class 1: 9%
- Class 2: 3%

---

7. Compilation: Choosing Loss, Optimizer & Metrics

Loss Function

For multiclass classification:

✅ **categorical_crossentropy**

OR

✅ **sparse_categorical_crossentropy**

Optimizer

✅ **Adam** (popular, faster convergence)

Metric

✅ **Accuracy**

---

8. Training (model.fit)

Parameters include:

- **epochs** (e.g., 50)
- **batch size** (e.g., 32)
- **training dataset**
- **validation dataset**

During each epoch:

1. Model trains on batches
2. Computes training loss & accuracy
3. Tests on validation set
4. Validation accuracy guides tuning

---

9. Understanding Training Outputs

✅ Training Accuracy

Improves steadily as model sees more data.

✅ Validation Accuracy

Represents how well model generalizes.

Expected pattern:

- Starts lower than training accuracy.
- Increases over epochs.
- If it starts decreasing: overfitting.

Example progress:

- Epoch 1: Train 51%, Val 71%
- Epoch 30: Train 95%, Val 95%
- Epoch 50: Train 99%, Val 98%

This shows **excellent generalization**.

---

10. Testing on Unseen Dataset

After training, evaluate using:

```
model.evaluate(test_dataset)
```

If test accuracy ≈ validation accuracy → good model

In text: Test accuracy reaches **98%** → very high performance.

---

11. Using History Object for Visualization

The history object stores:

```
loss
accuracy
val_loss
val_accuracy
```

Each has 50 values (one per epoch).

You plot:

✅ Training vs Validation Accuracy

Shows how well the model improves.

✅ Training vs Validation Loss

Shows reduction in error.

---

12. Predictions on New Images

Process:

1. Extract 1 image from batch
2. Convert to numpy array
3. Feed through model
4. Get softmax probabilities
5. Use **argmax** to get predicted class index
6. Map index → class label
7. Display image + actual label + predicted label + confidence percentage

Example:

```
Actual: Early Blight
Predicted: Early Blight
```

```
Confidence: 100%
```

You also visualize **multiple images** using subplots.

---

13. Building Utility Function for Prediction

A function wraps:

- Image → array
- Array → batch
- batch → model.predict
- Softmax → predicted class
- Argmax → class index
- max(probabilities) → confidence score

This keeps code clean and reusable.

---

14. Model Saving

There are two type by which model can saved:

1. Save Architecture and Weights Separately

```
import os

# Create directory if it doesn't exist
os.makedirs("../saved_models/my_model", exist_ok=True)

# Save full model (architecture + weights)
model.save("../saved_models/my_model/model.keras")

# Save architecture separately (optional)
with open("../saved_models/my_model/architecture.json", "w") as f:
    f.write(model.to_json())
```

```
# Save only weights (optional)
model.save_weights("../saved_models/my_model/weights.weights.h5")
```

2. Save in TensorFlow SavedModel Format (.pb)

```
# Save model in TensorFlow SavedModel format
model.export("saved_models/my_model_pb")
```

15. Model Versioning System

To manage experiments:

✔ Create “saved_models” directory

✔ Automatically detect existing versions

✔ Increment version number

✔ Save new model as /models/2, /models/3, etc.

Uses:

```
os.listdir()
max()
int()
```

This creates a **production-grade version management system**, essential for MLOps.

✔ Create “saved_models” directory

```
import os
```

```
os.makedirs("saved_models", exist_ok=True)
```

✔ Automatically detect existing versions

```
versions = [int(v) for v in os.listdir("saved_models") if v.isdigit()]
```

✔ Increment version number

```
new_version = max(versions) + 1 if versions else 1
save_path = f"saved_models/{new_version}"
os.makedirs(save_path, exist_ok=True)
```

✔ Save new model

For **Keras format:**

```
model.save(f"{save_path}/model.keras")
model.save_weights(f"{save_path}/weights.weights.h5")
```

Or for **TensorFlow SavedModel (.pb) format:**

```
model.export(save_path)
```

Uses

```
os.listdir()
max()
```

```
int()
```

✅ This creates a **production-grade model version management system**, essential for **MLOps** and tracking different model iterations easily.

16. Future Steps (Preview of Next Video)

Next steps include:

✅ Building a **FastAPI server**

✅ Using **TensorFlow Serving** to load versioned models

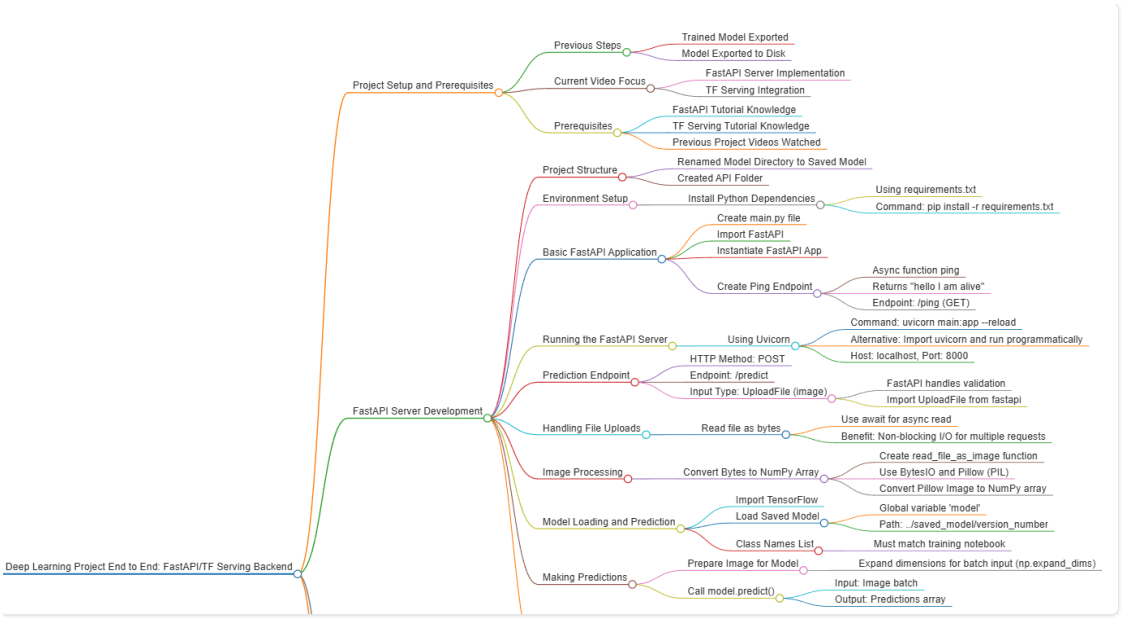
✅ Implementing real-time image prediction API endpoints

This is how deep learning models are deployed in real-world applications.

4. FastAPI/tf serving Backend

FastAPI + TensorFlow Serving + MLOps

The content is a detailed walkthrough of deploying a potato–disease detection ML model using **FastAPI**, first by loading the TensorFlow model directly and then by integrating with **TensorFlow Serving** for production-grade deployment. The tutorial covers practical MLOps principles and real-world deployment workflows.



Potato Disease Classification Project Setup

1. Open Project in PyCharm

1. Locate your project folder: `Potato-disease-classification`.

2. Right-click the folder → **Open with PyCharm** (or open PyCharm → File → Open → select folder).

---

2. Create Virtual Environment

1. Inside PyCharm or via CMD, create a virtual environment:

```
python -m venv venv
```

3. Activate the Virtual Environment

Windows

```
venv\\Scripts\\activate
```

4. Create `requirements.txt`

1. Open **Notepad**.
2. Write the following packages:

```
tensorflow
fastapi
uvicorn
python-multipart
pillow
tensorflow-serving-api
matplotlib
numpy
notebook
tensorflow-addons
tensorflow-model-optimization
```

3. Save the file as `requirements.txt` in your project folder.

5. Open Command Prompt in Project Folder

1. Navigate to your project folder:
   - Shift + Right-click on the folder → **Open command window here**

     or
   - Open CMD and run:

```
cd C:\Users\gs828\FINAL_PROJECTS\Potato-disease-classification
```

6. Install Requirements

```
pip install --upgrade pip
pip install -r requirements.txt
```

✅ This will install all necessary packages in your virtual environment.

7. Run `main.py`

1. Ensure the virtual environment is activated.
2. Run the main application:

```
python main.py
```

1. Your API/server should now be running and ready for testing.

Tip: You can clear CMD screen anytime using:

```
cls
```

✅ Key Takeaways

1. FastAPI Server Setup

- A simple FastAPI application is created with:
  - `/ping` endpoint → health check.
  - `/predict` endpoint → accepts an uploaded image file and returns model inference.
- Server is run using `uvicorn`, either via command-line or programmatically.

---

2. File Upload & Validation

- FastAPI's `UploadFile` type ensures that only valid uploaded file data (like images) can be sent.
- Demonstrated through both Swagger UI (`/docs`) and Postman.

---

3. Image Preprocessing

- Uploaded file is read asynchronously using `await file.read()`.
- Bytes → Pillow Image → NumPy array.
- An image batch is created via:

```
np.expand_dims(image, 0)
```

---

4. Direct Model Loading (Local TensorFlow SavedModel)

There are **two ways** to load models based on how they were saved:

1️⃣ Loading from Keras Model + Weights

If the model and weights were saved separately:

```
from keras import models, layers

resize_and_rescale = layers.Rescaling(1./255)  # or your custom layer
input_shape = (256, 256, 3)
n_classes = 3

model = models.Sequential([
    resize_and_rescale,
    layers.Conv2D(32, kernel_size = (3,3), activation='relu', input_shape=input_
shape),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (3,3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (3,3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(n_classes, activation='softmax'),
])

Load the trained model
model.load_weights("../saved_models/my_model/weights.weights.h5")
```

2️⃣ Loading from TensorFlow SavedModel (.pb)

If the full model was saved in **SavedModel** format:

```
from keras.models import load_model
```

```
model = load_model("saved_models/my_model_pb")
```

- In both cases, the prediction output is a **1×3 array** (for 3 classes).
- Use `np.argmax()` to get the **predicted class index**.
- Define class names for mapping:

```
CLASS_NAMES = ["Early Blight", "Late Blight", "Healthy"]
```

5. Asynchronous Architecture

- The tutorial explains why `async/await` is crucial:
  - Allows other requests to be processed while large files are being read.
  - Solves concurrency bottlenecks in production APIs.

---

6. MLOps Angle

- The tutorial gently introduces:
  - Model versioning.
  - Prediction endpoints.
  - Serving architecture.
  - Difference between direct loading vs. TensorFlow Serving.
- TensorFlow Serving is positioned as the real industrial solution for scalable and version-managed deployment.

✅ Deep Insights

- Demonstrates the transition from "notebook ML" → "real deployable ML".
- Shows the foundational MLOps pattern:

```
Client (Mobile/Web) → FastAPI → TF Serving → Model
```

- Highlights the value of:
  - Asynchronous APIs
  - Model lifecycle management
  - Separation of concerns (API vs. model serving)
  - Predictable, versioned deployments.

#️⃣ 1. FastAPI App Using Two Types of Model Loading (Keras Weights & TensorFlow SavedModel)

```
# ================================================
# ✅ FASTAPI APP FOR POTATO DISEASE CLASSIFICATION
# Supports two model loading methods:
#   1️⃣ Keras Model (Architecture + Weights)  ← Default Active
#   2️⃣ TensorFlow SavedModel (.pb format)     ← Alternate Option
# ================================================

from fastapi import FastAPI, File, UploadFile
from fastapi.middleware.cors import CORSMiddleware
import uvicorn
import numpy as np
from io import BytesIO
from PIL import Image
from keras.models import load_model
from keras import layers, models
import tensorflow as tf

# ------------------------------------------------
# 🚀 Initialize FastAPI app
# ------------------------------------------------
```

```
app = FastAPI()

# ---------------------------------------------
# 🌐 Allow CORS (for frontend access, e.g. React)
# ---------------------------------------------
origins = ["http://localhost", "http://localhost:3000"]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# ================================================================
# 🧩 1️⃣ LOAD MODEL (Keras Architecture + Weights) ← ACTIVE
# ================================================================

USE_KERAS_MODEL = True  # 🔁 Toggle between Keras (True) and SavedModel (False)

if USE_KERAS_MODEL:
    print(" 🔹 Loading Keras model (architecture + weights)...")

    resize_and_rescale = layers.Rescaling(1. / 255)
    input_shape = (256, 256, 3)
    n_classes = 3

    model = models.Sequential([
        resize_and_rescale,
        layers.Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape),
        layers.MaxPooling2D((2, 2)),
```

```
        layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(n_classes, activation='softmax'),
    ])

    # Load the trained weights only
    model.load_weights("../saved_models/my_model/weights.weights.h5")

else:
    # ================================================================
    # 🧠 2️⃣ LOAD MODEL (TensorFlow SavedModel .pb format)
    # ================================================================
    print("🔹 Loading TensorFlow SavedModel (.pb format)...")

    model = tf.saved_model.load("../saved_models/my_model_pb")
    infer = model.signatures["serving_default"]  # ✅ Serving signature for inference

# ------------------------------------------------
# 🏷️ Define class labels
# ------------------------------------------------
CLASS_NAMES = ["Early Blight", "Late Blight", "Healthy"]

# ------------------------------------------------
```

```python
# 🔍 Health check route
# ------------------------------------------------
@app.get("/ping")
async def ping():
    return {"message": "Hello, I am alive"}

# ------------------------------------------------
# 🖼️ Helper function — convert uploaded image to NumPy array
# ------------------------------------------------
def read_file_as_image(data) -> np.ndarray:
    image = Image.open(BytesIO(data)).convert("RGB")
    image = image.resize((256, 256))
    image = np.array(image) / 255.0
    return image

# ------------------------------------------------
# 🤖 Prediction endpoint
# ------------------------------------------------
@app.post("/predict")
async def predict(file: UploadFile = File(...)):
    image = read_file_as_image(await file.read())
    img_batch = np.expand_dims(image, axis=0)

    if USE_KERAS_MODEL:
        predictions = model.predict(img_batch)
    else:
        output = infer(tf.constant(img_batch, dtype=tf.float32))
        output_key = list(output.keys())[0]
        predictions = output[output_key].numpy()

    predicted_class = CLASS_NAMES[np.argmax(predictions[0])]
    confidence = float(np.max(predictions[0]))

    return {
        "class": predicted_class,
        "confidence": confidence
```

```
    }

# ---------------------------------------------
# ▶️ Run FastAPI app
# ---------------------------------------------
if __name__ == "__main__":
    uvicorn.run(app, host="localhost", port=8000)
```

🧠 FastAPI (main.py) — Breakdown explanation

Supports two model loading methods:

✅ **Keras Model (Architecture + Weights)** *(Default Active)*

⚙️ **TensorFlow SavedModel (.pb format)** *(Alternate Option)*

```
from fastapi import FastAPI, File, UploadFile
from fastapi.middleware.cors import CORSMiddleware
import uvicorn
import numpy as np
from io import BytesIO
from PIL import Image
from keras.models import load_model
from keras import layers, models
import tensorflow as tf
```

✅ What this section does

- **FastAPI** → creates and manages the backend API server.
- **UploadFile, File** → handle uploaded images sent from clients.
- **CORS Middleware** → allows frontend apps (like React) to make API calls.
- **uvicorn** → runs the FastAPI application.

- **NumPy** → performs array and mathematical operations for preprocessing.
- **Pillow (PIL)** → handles image opening, resizing, and format conversion.
- **TensorFlow / Keras** → load and execute trained ML models for predictions.

```
app = FastAPI()
```

✅ Creates the FastAPI App Instance

This initializes the web server that handles all API endpoints.

```
origins = ["http://localhost", "http://localhost:3000"]
```

✅ Allowed Frontend Origins

Allows frontend applications (like React) running locally to access this API.

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

✅ CORS Middleware Setup

Prevents browser CORS (Cross-Origin Resource Sharing) errors.

Enables communication between backend and frontend apps hosted on different ports.

🧩 MODEL LOADING METHODS

1️⃣ Load Keras Model (Architecture + Weights) — *Default Active*

```
USE_KERAS_MODEL = True  # 🔁 Toggle between Keras (True) and SavedMo
del (False)

if USE_KERAS_MODEL:
    print(" 🔹 Loading Keras model (architecture + weights)...")

    resize_and_rescale = layers.Rescaling(1. / 255)
    input_shape = (256, 256, 3)
    n_classes = 3

    model = models.Sequential([
        resize_and_rescale,
        layers.Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=inpu
t_shape),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(n_classes, activation='softmax'),
    ])

    # Load trained weights only
```

```
    model.load_weights("../saved_models/my_model/weights.weights.h5")
```

🟢 **Use this when:**

- The model was saved using `model.save_weights()`.
- Architecture is recreated manually before loading the `.weights.h5` file.
- You want flexibility to modify or inspect the model structure.

2️⃣ Load TensorFlow SavedModel (.pb format) — *Alternate Option*

```
else:
    print("🔹 Loading TensorFlow SavedModel (.pb format)...")
    model = tf.saved_model.load("../saved_models/my_model_pb")
    infer = model.signatures["serving_default"]
```

🟢 **Use this when:**

- The model was exported or saved in **SavedModel** format.
- All components (architecture, weights, signatures) are bundled together.
- No need to redefine the architecture in code.

```
CLASS_NAMES = ["Early Blight", "Late Blight", "Healthy"]
```

✅ Human-Readable Labels

Maps model output indices to disease category names.

```
@app.get("/ping")
async def ping():
```

```
    return {"message": "Hello, I am alive"}
```

✅ Health Check Endpoint

Used to verify the FastAPI server is running correctly.

```
def read_file_as_image(data) -> np.ndarray:
    image = Image.open(BytesIO(data)).convert("RGB")
    image = image.resize((256, 256))
    image = np.array(image) / 255.0
    return image
```

✅ Convert Uploaded Image → NumPy Array

1. Converts bytes data to an RGB image.
2. Resizes to `256×256` (model input size).
3. Normalizes pixel values to range `[0, 1]`.

```
@app.post("/predict")
async def predict(file: UploadFile = File(...)):
```

✅ Prediction Endpoint

Handles image uploads from clients and returns disease predictions.

```
    image = read_file_as_image(await file.read())
    img_batch = np.expand_dims(image, axis=0)
```

✅ Preprocessing Before Prediction

- Reads image asynchronously.
- Expands dimensions to simulate a batch input shape `(1, 256, 256, 3)`.

```
if USE_KERAS_MODEL:
    predictions = model.predict(img_batch)
else:
    output = infer(tf.constant(img_batch, dtype=tf.float32))
    output_key = list(output.keys())[0]
    predictions = output[output_key].numpy()
```

✅ Model Inference

- **Keras model:** Uses `.predict()` for inference.
- **SavedModel:** Uses TensorFlow's `infer()` signature to generate predictions.

```
predicted_class = CLASS_NAMES[np.argmax(predictions[0])]
confidence = float(np.max(predictions[0]))
```

✅ Convert Predictions → Labels

- `np.argmax()` → identifies highest probability class.
- Maps it to readable class name via `CLASS_NAMES`.
- Extracts confidence score.

```
return {
    "class": predicted_class,
    "confidence": confidence
}
```

✅ API Response Format

Sends prediction as JSON:

```
{
  "class": "Early Blight",
  "confidence": 0.92
}
```

```
if __name__ == "__main__":
    uvicorn.run(app, host="localhost", port=8000)
```

✅ Run FastAPI Server

Runs the API on your local machine at:

👉 **http://localhost:8000**

⚙️ Summary

Model Type	Save Method	Load Method	Architecture Needed?
Keras Weights	`model.save_weights()`	Recreate model + `load_weights()`	✅ Yes
TensorFlow SavedModel	`model.export()` or `model.save()`	`load_model("path")`	❌ No

✅ JSON response sent to frontend

#️⃣ 2. FastAPI App Using TensorFlow Serving (Recommended for Production)

This version **does not load the model directly**.

Instead, it sends the image to **TensorFlow Serving**.

```
# ================================================
# 🚀 Potato Disease Classification using FastAPI + TensorFlow Serving
# ================================================

# Import required libraries
from fastapi import FastAPI, File, UploadFile          # For building REST API an
d file upload
from fastapi.middleware.cors import CORSMiddleware      # To allow cross-ori
gin requests (frontend-backend connection)
import uvicorn                                  # For running FastAPI server
import numpy as np                                  # For numerical operations
from io import BytesIO                              # For reading image bytes
from PIL import Image                               # For image processing
import requests                                 # For sending HTTP requests to Tens
orFlow Serving

# Initialize FastAPI app with a title
app = FastAPI(title="Potato Disease Classification API (TF Serving)")

# ================================================
# 🌐 Configure CORS (Cross-Origin Resource Sharing)
# ================================================
# Allow requests from your frontend (like React app running on localhost:300
0)
origins = ["http://localhost", "http://localhost:3000"]
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,          # Allowed origins
    allow_credentials=True,
    allow_methods=["*"],            # Allow all HTTP methods (GET, POST, etc.)
    allow_headers=["*"],            # Allow all headers
)
```

```
# ================================================
# 🔗 TensorFlow Serving endpoint (Docker container)
# ================================================
# This URL corresponds to the TF Serving container running your model
ENDPOINT = "http://localhost:8501/v1/models/potatoes_model:predict"

# ================================================
# 🏷️ Class labels corresponding to your model output
# ================================================
CLASS_NAMES = ["Early Blight", "Late Blight", "Healthy"]

# ================================================
# ✅ Health check route — confirms that API is alive
# ================================================
@app.get("/ping")
async def ping():
    return {"message": "Hello, I am alive!"}

# ================================================
# 🖼️ Function to preprocess uploaded image
# ================================================
def read_file_as_image(data) -> np.ndarray:
    # Read the image from uploaded bytes and convert to RGB format
    image = Image.open(BytesIO(data)).convert("RGB")
    # Resize to model input size
    image = image.resize((256, 256))
    # Convert image to numpy array and normalize (0-1 range)
    image = np.array(image) / 255.0
    return image

# ================================================
# 🔮 Prediction route — sends image to TF Serving and returns result
# ================================================
@app.post("/predict")
async def predict(file: UploadFile = File(...)):
```

```
    # Step 1️⃣: Read and preprocess image
    image = read_file_as_image(await file.read())
    # Add batch dimension — model expects input shape (1, 256, 256, 3)
    img_batch = np.expand_dims(image, axis=0)

    # Step 2️⃣: Prepare JSON data and send request to TF Serving
    json_data = {"instances": img_batch.tolist()}
    response = requests.post(ENDPOINT, json=json_data)

    # Step 3️⃣: Handle TensorFlow Serving errors (if any)
    if response.status_code != 200:
        return {"error": f"TensorFlow Serving error: {response.text}"}

    # Step 4️⃣: Extract predictions from TF Serving response
    predictions = np.array(response.json()["predictions"][0])
    predicted_class = CLASS_NAMES[np.argmax(predictions)]
    confidence = float(np.max(predictions))

    # Step 5️⃣: Return final result to frontend / user
    return {"class": predicted_class, "confidence": confidence}

# ================================================
# ▶️ Run the FastAPI server (localhost:8000)
# ================================================
if __name__ == "__main__":
    uvicorn.run(app, host="localhost", port=8000)
```

🧠 Goal

You'll run both:

- **TensorFlow Serving (Docker)** → serves the model
- **FastAPI app** → calls TF Serving and returns predictions

⚙️ Step 1. Make sure Docker is running

✅ Start Docker Desktop first.

Wait until it says **"Docker engine is running."**

🧩 Step 2. Run TensorFlow Serving container

Use this **exact command** (since your model path must not change):

```
docker rm -f potatoes_model_serving
docker run -p 8501:8501 --name=potatoes_model_serving `
  --mount type=bind,source="C:\Users\gs828\FINAL_PROJECTS\Potato-disease-classification\saved_models\my_model_pb",target=/models/potatoes_model/1 `
  -e MODEL_NAME=potatoes_model -t tensorflow/serving
```

💡 What this does

- Mounts your **local model** folder into the Docker container
- Exposes port **8501** for predictions
- Model available at:

  👉 `http://localhost:8501/v1/models/potatoes_model:predict`

✅ Check if it loaded successfully

After running, you should see:

```
SavedModel load for tags { serve }; Status: success: OK.
```

and no errors repeating.

🧠 Step 3. Activate your Python virtual environment

In PowerShell, inside your project folder:

```
cd C:\Users\gs828\FINAL_PROJECTS\Potato-disease-classification
.\.venv\Scripts\activate
cd api
```

Now you'll see `(.venv)` before the path → means venv is active.

🚀 Step 4. Run the FastAPI app

Run:

```
python main-tf-serving.py
```

You should see:

```
Uvicorn running on http://localhost:8000
```

🌐 Step 5. Test in browser or Postman

- Open docs UI: 👉 http://localhost:8000/docs
- Use `/ping` to check connection
- Use `/predict` → upload an image (potato leaf)

✅ Step 6. Stop when done

- To stop FastAPI → Press `CTRL + C`
- To stop Docker container → run:

  ```
  docker stop potatoes_model_serving
  ```

#️⃣ Summary: Direct Model Loading vs TensorFlow Serving

Feature	Direct Loading	TensorFlow Serving
Where model runs	Inside FastAPI	In separate TF Serving container
Performance	Slower for production	Much faster & scalable
Model Reloading	Requires restart	Hot-reload supported
Versioning	Manual	Built-in versioning
Best for	Small projects, local testing	Production, scaling, MLOps

BREAKDOWN EXPLANATION

1️⃣ Import Required Libraries

```
from fastapi import FastAPI, File, UploadFile
from fastapi.middleware.cors import CORSMiddleware
import uvicorn
import numpy as np
from io import BytesIO
from PIL import Image
import requests
```

Explanation:

- **FastAPI** → To create REST APIs.
- **UploadFile & File** → Handle image file uploads.
- **CORSMiddleware** → Enable communication between frontend (React) and backend.
- **uvicorn** → ASGI server to run FastAPI.
- **numpy** → Numerical operations, array manipulations.
- **BytesIO & PIL.Image** → Read and process images.
- **requests** → Send HTTP POST requests to TensorFlow Serving.

2️⃣ Initialize FastAPI App

```
app = FastAPI(title="Potato Disease Classification API (TF Serving)")
```

Explanation:

- Creates a FastAPI instance with a custom title.
- All API routes will be defined using this instance.

3️⃣ Configure CORS (Cross-Origin Requests)

```
origins = ["http://localhost", "http://localhost:3000"]
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Explanation:

- Allows frontend applications (like React on localhost:3000) to access the backend.
- Enables all HTTP methods and headers.

4️⃣ Define TensorFlow Serving Endpoint

```
ENDPOINT = "http://localhost:8501/v1/models/potatoes_model:predict"
```

Explanation:

- URL of the TensorFlow Serving Docker container where the model is served.
- `/v1/models/<model_name>:predict` is the standard TF Serving REST API format.

5️⃣ Define Class Labels

```
CLASS_NAMES = ["Early Blight", "Late Blight", "Healthy"]
```

Explanation:

- Labels corresponding to the output of your model.
- Will be used to interpret predictions.

6️⃣ Health Check Route

```
@app.get("/ping")
async def ping():
    return {"message": "Hello, I am alive!"}
```

Explanation:

- Simple GET route to check if the API is running.
- Can be tested in browser or Postman: `http://localhost:8000/ping`.

7️⃣ Preprocess Uploaded Image

```
def read_file_as_image(data) -> np.ndarray:
    image = Image.open(BytesIO(data)).convert("RGB")
    image = image.resize((256, 256))
    image = np.array(image) / 255.0
    return image
```

Explanation:

- Converts uploaded image bytes into RGB format.
- Resizes image to model input size (256×256).
- Normalizes pixel values to range `[0, 1]`.
- Returns **numpy array** ready for model prediction.

8️⃣ Prediction Route

```
@app.post("/predict")
async def predict(file: UploadFile = File(...)):
    image = read_file_as_image(await file.read())
    img_batch = np.expand_dims(image, axis=0)
    json_data = {"instances": img_batch.tolist()}
    response = requests.post(ENDPOINT, json=json_data)

    if response.status_code != 200:
        return {"error": f"TensorFlow Serving error: {response.text}"}

    predictions = np.array(response.json()["predictions"][0])
    predicted_class = CLASS_NAMES[np.argmax(predictions)]
    confidence = float(np.max(predictions))
    return {"class": predicted_class, "confidence": confidence}
```

Step-by-step Explanation:

1. **Upload and preprocess image** → Converts image into normalized array.
2. **Add batch dimension** → Model expects shape `(1, 256, 256, 3)`.
3. **Send request to TF Serving** → Convert image array to JSON and POST to the endpoint.
4. **Error handling** → If TF Serving fails, return error message.
5. **Extract prediction** → Find the class with highest probability.
6. **Return result** → Send JSON response with predicted class and confidence.

9️⃣ Run FastAPI Server

```
if __name__ == "__main__":
    uvicorn.run(app, host="localhost", port=8000)
```

Explanation:

- Starts FastAPI server locally at `http://localhost:8000`.
- Can now send requests to `/ping` and `/predict` routes.

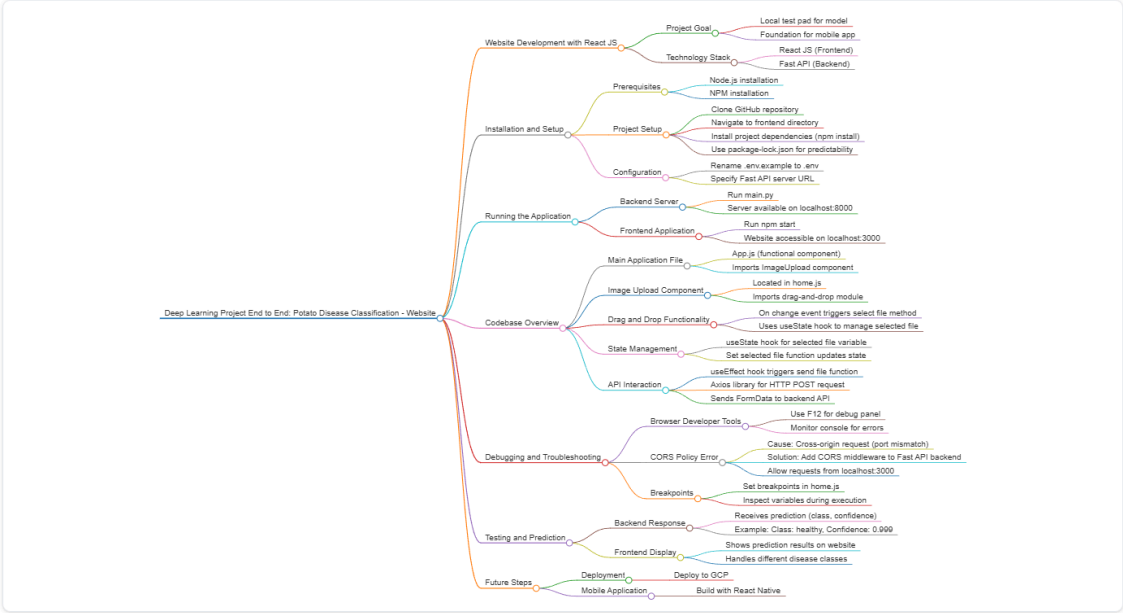
✅ Summary of Workflow

1. **Frontend uploads image** → POST request to `/predict`.
2. **Backend preprocesses image** → Resize, normalize.
3. **Backend sends image to TF Serving** → Gets prediction.
4. **Backend returns class label and confidence** → Displayed on frontend

5. Website (In React JS)

Building a React.js Frontend for Potato Disease Detection

This documentation explains how a React.js web application was built as a **local test interface** for a deep learning model. The website allows a user to **drag and drop an image** of a potato leaf, sends the file to a **FastAPI backend**, and displays the disease prediction returned by the model.



✅ Core Theme

The project demonstrates how to integrate:

- A **React.js UI**
- A **FastAPI backend**
- A **TensorFlow-based deep learning model**

The purpose is to provide a **user-friendly testing environment** before deploying the solution as a **mobile app for farmers**.

✅ Website Functionality (High-Level)

The React app allows users to:

1. **Drag and drop** an image of a potato leaf.
2. The frontend sends the image to the FastAPI server endpoint:

   ```
   POST http://localhost:8000/predict
   ```

3. The FastAPI backend preprocesses the image and performs inference using:

- A locally loaded TensorFlow model, or
- A remote TensorFlow Serving endpoint.

4. The frontend receives the prediction and displays:
   - Predicted class (Early Blight, Late Blight, Healthy)
   - Confidence score
5. This gives an **instant visualization** of how the trained model behaves with real images.

This “test pad” acts as a safe, local playground before building the **React Native mobile app** for deployment.

✅ Development Environment Setup

The tutorial walks through preparing the front-end environment:

⚙️ 1. Install **Node.js**

Node.js includes **npm**, so installing Node covers both.

🪟 On Windows:

1. Go to 👉 https://nodejs.org
2. Download the **LTS version** (Recommended for most users).
3. Run the installer:
   - Click “Next” on each step.
   - Check **“Add to PATH”** during installation.
4. Once done, verify installation:

```
node -v
npm -v
```

   You should see version numbers (e.g., `v20.x.x` and `10.x.x`).

🧩 2. Install Dependencies

Navigate to your **frontend** folder (where your React app code exists):

```
cd frontend
```

Then install all dependencies listed in your lock file:

```
npm install --from-lock-json
```

> ⚠️ Note: If you get an error with --from-lock-json, use npm ci instead (it's the modern replacement).

```
npm ci
```

or (if lock file not available)

```
npm install
```

Then fix any security vulnerabilities:

```
npm audit fix
```

If you still see high/critical ones and you want to force update (be cautious):

```
npm audit fix --force
```

🧾 3. Setup Environment File

Copy the example environment file and rename it:

```
cp .env.example .env
```

(If you're on Windows PowerShell:)

```
copy .env.example .env
```

🌐 4. Change API URL in `.env`

Open `.env` in any editor (like VS Code):

```
code .env
```

Then find the line:

```
REACT_APP_API_URL=http://0.0.0.0:8000/predict
```

Change it to your backend API address, e.g.:

```
REACT_APP_API_URL=http://localhost:8000
```

or

```
REACT_APP_API_URL=https://your-deployed-api.com
```

🚀 Final setup summary

Component	Runs on	URL
React app	Port 3000	http://localhost:3000
FastAPI app	Port 8000	http://localhost:8000

Component	Runs on	URL
Prediction endpoint	POST	http://localhost:8000/predict
`.env` **variable**		`REACT_APP_API_URL=http://localhost:8000`

🚀 5. Run React App

Finally, start your frontend:

```
npm run start
```

Your React app will run on 👉 http://localhost:3000

✅ Understanding the React.js Architecture

home.js

```
import React, { useState, useEffect, useCallback } from "react";
import {
  makeStyles,
  withStyles,
} from "@material-ui/core/styles";
import {
  AppBar,
  Toolbar,
  Typography,
  Avatar,
  Container,
  Card,
  CardContent,
  Paper,
  CardActionArea,
  CardMedia,
  Grid,
  TableContainer,
```

```
  Table,
  TableBody,
  TableHead,
  TableRow,
  TableCell,
  Button,
  CircularProgress,
} from "@material-ui/core";
import { common } from "@material-ui/core/colors";
import Clear from "@material-ui/icons/Clear";
import { DropzoneArea } from "material-ui-dropzone";
import axios from "axios";
import potatologo from "./potatologo.png";
import bgImage from "./bg.png";

// ----------------------------------------------------
// 🎨 Custom Button
// ----------------------------------------------------
const ColorButton = withStyles((theme) => ({
  root: {
    color: theme.palette.getContrastText(common.white),
    backgroundColor: common.white,
    "&:hover": {
      backgroundColor: "#ffffff7a",
    },
  },
}))(Button);

// ----------------------------------------------------
// 🧩 Styles
// ----------------------------------------------------
const useStyles = makeStyles((theme) => ({
  grow: { flexGrow: 1 },
  clearButton: {
    width: "-webkit-fill-available",
    borderRadius: "15px",
```

```
    padding: "15px 22px",
    color: "#000000a6",
    fontSize: "20px",
    fontWeight: 900,
  },
  media: { height: 400 },
  gridContainer: {
    justifyContent: "center",
    padding: "4em 1em 0 1em",
  },
  mainContainer: {
    backgroundImage: `url(${bgImage})`,
    backgroundRepeat: "no-repeat",
    backgroundPosition: "center",
    backgroundSize: "cover",
    height: "93vh",
    marginTop: "8px",
  },
  imageCard: {
    margin: "auto",
    maxWidth: 400,
    height: 500,
    backgroundColor: "transparent",
    boxShadow: "0px 9px 70px 0px rgb(0 0 0 / 30%) !important",
    borderRadius: "15px",
  },
  appbar: {
    background: "#be6a77",
    boxShadow: "none",
    color: "white",
  },
  loader: {
    color: "#be6a77 !important",
  },
  tableCell: {
    fontSize: "22px",
```

```
    backgroundColor: "transparent !important",
    borderColor: "transparent !important",
    color: "#000000a6 !important",
    fontWeight: "bolder",
    padding: "1px 24px 1px 16px",
  },
  tableCell1: {
    fontSize: "14px",
    backgroundColor: "transparent !important",
    borderColor: "transparent !important",
    color: "#000000a6 !important",
    fontWeight: "bolder",
    padding: "1px 24px 1px 16px",
  },
  detail: {
    backgroundColor: "white",
    display: "flex",
    justifyContent: "center",
    flexDirection: "column",
    alignItems: "center",
  },
}));

// ----------------------------------------------------
// 🚀 Main Component
// ----------------------------------------------------
export const ImageUpload = () => {
  const classes = useStyles();
  const [selectedFile, setSelectedFile] = useState(null);
  const [preview, setPreview] = useState(null);
  const [data, setData] = useState(null);
  const [hasImage, setHasImage] = useState(false);
  const [isLoading, setIsLoading] = useState(false);

  // ✅ API Base URL (use .env or fallback)
  const API_BASE = process.env.REACT_APP_API_URL || "http://localhost:800
```

```
0";

  // ✅ Send File to FastAPI backend
  const sendFile = useCallback(async () => {
    if (hasImage && selectedFile) {
      try {
        setIsLoading(true);
        const formData = new FormData();
        formData.append("file", selectedFile);

        const res = await axios.post(`${API_BASE}/predict`, formData, {
          headers: { "Content-Type": "multipart/form-data" },
        });

        if (res.status === 200) {
          setData(res.data);
        }
      } catch (err) {
        console.error("❌ Error uploading file:", err);
        alert("Prediction failed! Please check backend connection.");
      } finally {
        setIsLoading(false);
      }
    }
  }, [hasImage, selectedFile, API_BASE]); // dependencies

  // ✅ Clear all data
  const clearData = () => {
    setData(null);
    setHasImage(false);
    setSelectedFile(null);
    setPreview(null);
  };

  // ✅ Generate preview for uploaded image
  useEffect(() => {
```

```
  if (!selectedFile) {
    setPreview(undefined);
    return;
  }
  const objectUrl = URL.createObjectURL(selectedFile);
  setPreview(objectUrl);
  return () => URL.revokeObjectURL(objectUrl);
}, [selectedFile]);

// ✅ Trigger prediction automatically once preview is ready
useEffect(() => {
  if (preview) sendFile();
}, [preview, sendFile]);

// ✅ Handle file selection from dropzone
const onSelectFile = (files) => {
  if (!files || files.length === 0) {
    setSelectedFile(undefined);
    setHasImage(false);
    setData(undefined);
    return;
  }
  setSelectedFile(files[0]);
  setData(undefined);
  setHasImage(true);
};

const confidence = data
  ? (parseFloat(data.confidence) * 100).toFixed(2)
  : 0;

return (
  <>
    {/* Navbar */}
    <AppBar position="static" className={classes.appbar}>
      <Toolbar>
```

```
          <Typography variant="h6" noWrap>
            Potato Disease Classifier
          </Typography>
          <div className={classes.grow} />
          <Avatar src={potatologo} />
        </Toolbar>
      </AppBar>

      {/* Main Container */}
      <Container
        maxWidth={false}
        className={classes.mainContainer}
        disableGutters={true}
      >
        <Grid
          className={classes.gridContainer}
          container
          direction="row"
          justifyContent="center"
          alignItems="center"
          spacing={2}
        >
          <Grid item xs={12}>
            <Card className={classes.imageCard}>
              {/* Image Preview */}
              {hasImage && (
                <CardActionArea>
                  <CardMedia
                    className={classes.media}
                    image={preview}
                    component="img"
                    title="Potato Leaf"
                  />
                </CardActionArea>
              )}
```

```
          {/* Upload Area */}
          {!hasImage && (
            <CardContent>
              <DropzoneArea
                acceptedFiles={["image/*"]}
                dropzoneText={
                  "Drag and drop an image of a potato leaf to classify"
                }
                onChange={onSelectFile}
              />
            </CardContent>
          )}

          {/* Prediction Result */}
          {data && (
            <CardContent className={classes.detail}>
              <TableContainer component={Paper}>
                <Table size="small">
                  <TableHead>
                    <TableRow>
                      <TableCell className={classes.tableCell1}>
                        Label:
                      </TableCell>
                      <TableCell
                        align="right"
                        className={classes.tableCell1}
                      >
                        Confidence:
                      </TableCell>
                    </TableRow>
                  </TableHead>
                  <TableBody>
                    <TableRow>
                      <TableCell
                        component="th"
                        scope="row"
```

```
                  className={classes.tableCell}
                >
                  {data.class}
                </TableCell>
                <TableCell
                  align="right"
                  className={classes.tableCell}
                >
                  {confidence}%
                </TableCell>
              </TableRow>
            </TableBody>
          </Table>
        </TableContainer>
      </CardContent>
    )}

    {/* Loader */}
    {isLoading && (
      <CardContent className={classes.detail}>
        <CircularProgress color="secondary" />
        <Typography variant="h6" noWrap>
          Processing...
        </Typography>
      </CardContent>
    )}
  </Card>
</Grid>

{/* Clear Button */}
{data && (
  <Grid item className={classes.buttonGrid}>
    <ColorButton
      variant="contained"
      className={classes.clearButton}
      onClick={clearData}
```

```
            startIcon={<Clear fontSize="large" />}
          >
            Clear
          </ColorButton>
        </Grid>
      )}
    </Grid>
  </Container>
 </>
 );
};
```

🧭 Step-by-Step Explanation of **frontend (React)** and **backend (FastAPI)**

Step 1: Project Structure Overview

Your folder contains two main parts:

```
POTATO-DISEASE-CLASSIFICATION/
│
├── api/                → FastAPI backend
│   ├── main.py
│   ├── main-aug.py
│   └── main-tf-serving.py
│
├── frontend/           → React frontend
│   ├── public/
│   └── src/
│       ├── App.js
│       └── home.js
```

- **Backend (FastAPI)** handles **image prediction** using the trained ML model.
- **Frontend (React)** provides a **web interface** for uploading images and displaying results.

Step 2: App.js – Root Component

`App.js` is the entry point of your React app.

It simply loads the main component (`ImageUpload`) from `home.js`:

```
import { ImageUpload } from "./home";

function App() {
  return <ImageUpload />;
}

export default App;
```

So when the React app runs, it displays the entire upload and prediction interface from `home.js`.

Step 3: home.js – Main Functional Component

`home.js` defines the main UI and logic for uploading an image, sending it to the backend, and showing predictions.

Inside it, React hooks are used:

- **useState** – to store file, prediction data, and loader states.
- **useEffect** – to perform actions automatically when certain data changes.

Step 4: Selecting and Previewing an Image

When the user uploads or drags a file:

```
const onSelectFile = (files) => {
  setSelectedFile(files[0]);
```

```
  setHasImage(true);
};
```

React immediately generates a preview of the image using:

```
const objectUrl = URL.createObjectURL(selectedFile);
setPreview(objectUrl);
```

This shows the uploaded leaf image on the screen.

Step 5: Preparing Image for Backend

When the preview is ready, React automatically sends the image to the backend using `Axios`:

```
const formData = new FormData();
formData.append("file", selectedFile);
```

Step 6: Sending the Request to FastAPI

Axios sends a POST request to the backend:

```
axios.post(`${API_BASE}/predict`, formData, {
  headers: { "Content-Type": "multipart/form-data" },
});
```

Here:

- `API_BASE` = `"http://localhost:8000"`
- `/predict` = backend endpoint in FastAPI
- **FormData** contains the uploaded image file

Step 7: FastAPI Backend Receives the Image

FastAPI's `/predict` route receives the image:

```
@app.post("/predict")
async def predict(file: UploadFile = File(...)):
    contents = await file.read()
    image = Image.open(io.BytesIO(contents))
```

Then:

1. The image is converted to a NumPy array
2. It's passed to the trained ML model
3. The model returns a **class name** (e.g. "Late Blight") and **confidence** value

Example response:

```
{
  "class": "Healthy",
  "confidence": 0.9876
}
```

Step 8: FastAPI Response Sent to React

FastAPI sends this JSON back to the frontend:

```
return {"class": "Healthy", "confidence": 0.9876}
```

Step 9: React Displays the Prediction

React receives the response and updates the UI:

```
setData(res.data);
```

Then it shows the result inside a styled Material-UI card:

- **Class Name:** Healthy
- **Confidence:** 98.76%

Step 10: Showing Loader During Processing

While waiting for prediction, a circular loader appears:

```
{isLoading && (
  <CircularProgress color="secondary" />
)}
```

Once prediction is complete, the loader disappears and the result is shown.

Step 11: Clear Button Functionality

The “Clear” button removes the old image and prediction:

```
const clearData = () => {
  setData(null);
  setHasImage(false);
  setSelectedFile(null);
  setPreview(null);
};
```

This resets the UI for a new upload.

Step 12: Handling CORS Between Frontend and Backend

Since:

- Frontend runs on **localhost:3000**
- Backend runs on **localhost:8000**

Browsers block direct communication unless CORS is allowed.

So in FastAPI:

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

This allows React (port 3000) to connect safely to FastAPI (port 8000).

Step 13: Environment Variables

You can store the API URL in a `.env` file inside the frontend:

```
REACT_APP_API_URL=http://localhost:8000
```

Then use:

```
const API_BASE = process.env.REACT_APP_API_URL || "http://localhost:8000";
```

This helps when deploying or running on different servers.

Step 14: Optional – Serve Frontend from FastAPI

In production, you can serve the built React app from FastAPI itself:

1. Build the frontend:

```
cd frontend
npm run build
```

2. Mount the build folder in FastAPI:

```
from fastapi.staticfiles import StaticFiles
app.mount("/", StaticFiles(directory="../frontend/build", html=True), name
="frontend")
```

Now both frontend and backend run on the same URL (e.g., `http://localhost:8000`).

Step 15: End-to-End Data Flow Summary

```
User → React (home.js) → Axios POST /predict → FastAPI → Model → Predicti
on → React (Display)
```

Explanation:

1. User uploads image
2. React sends image → FastAPI
3. FastAPI sends image → Model
4. Model predicts class + confidence
5. FastAPI sends JSON → React
6. React shows result instantly

Step 16: Technologies Working Together

Layer	Technology	Purpose
Frontend	React.js + Material UI	User interface
Backend	FastAPI	REST API to serve predictions

Layer	Technology	Purpose
Model Serving	TensorFlow / TensorFlow Serving	Runs trained model
Communication	Axios (Frontend) + JSON (Backend)	File & data transfer
Deployment	Uvicorn	Run backend server

Step 17: Real-World Concept

This setup demonstrates an **end-to-end ML deployment**:

> Frontend (UI) → API Layer (FastAPI) → ML Model (TensorFlow) → Response

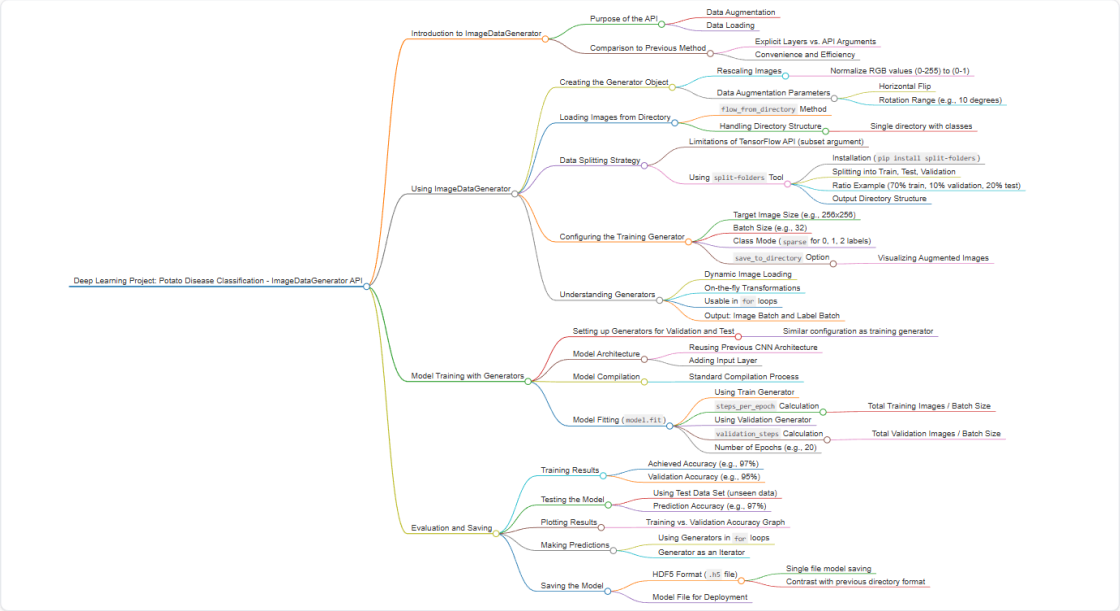
6. ImageDataGenerator API

Understanding TensorFlow's ImageDataGenerator API

Before deploying the model to Google Cloud, the video walks through a foundational topic:

using the ImageDataGenerator API in TensorFlow/Keras to handle **data augmentation**, **image loading**, and **batch generation** during training.

This explanation covers the full workflow of preparing a dataset with ImageDataGenerator, training a Convolutional Neural Network (CNN), evaluating it properly, and finally saving the model in a deployment-friendly format.



✅ Why This Session Exists

In earlier parts of the series, the model used **Keras augmentation layers** like:

```
tf.keras.layers.Rescaling(...)
tf.keras.layers.RandomRotation(...)
tf.keras.layers.RandomFlip(...)
```

Many viewers requested an alternative approach using the **ImageDataGenerator API**, an older but very convenient method.

Thus, this session **repeats the same CNN training**, but using ImageDataGenerator — which combines **augmentation + data loading** in a single interface.

✅ What Is ImageDataGenerator?

`ImageDataGenerator` is a high-level API that:

✅ Loads images directly from folders

✅ Applies augmentation (rotation, flipping, brightness, etc.)

✅ Feeds data to the model in real-time batches

✅ Automatically rescales pixel values

✅ Works very well for directory-based datasets

It dramatically simplifies preprocessing by packing augmentation into one object.

Example:

```
datagen = ImageDataGenerator(
    rescale=1/255,
    horizontal_flip=True,
    rotation_range=10
)
```

✅ Limitation: No Built-in Train/Val/Test Split

`flow_from_directory()` supports **only training & validation split**, like:

```
ImageDataGenerator(validation_split=0.1)
```

But there is **no built-in option** for a **test** split.

Thus, the video uses an external tool:

```
pip install split-folders
```

This tool can split a dataset into:

- Train
- Validation
- Test

Example usage:

```
splitfolders --ratio 0.7 0.1 0.2 --output dataset plant_village
```

Resulting folder structure:

```
dataset/
    train/
        early_blight/
        late_blight/
        healthy/
    val/
    test/
```

✅ Creating the Data Generators

Once the dataset is split, all three loaders are created:

Training Generator

```
train_gen = datagen.flow_from_directory(
    'dataset/train',
    target_size=(256, 256),
    batch_size=32,
    class_mode='sparse'   # labels: 0,1,2 instead of one-hot
    save_to_dir="AugmentedImages" #and then in cmd in training folder write c
ode: mkdir AugmentedImages
)
```

✅ Returns batches of (32 images, 32 labels)

✅ Applies augmentation on-the-fly

✅ Rescales pixel values (0–1)

✅ Dynamically loads from disk

Validation Generator

Same code with directory changed.

Test Generator

Same code with directory changed.

✅ How Generators Work (Important Concept)

Generators yield data **batch-by-batch** using Python's iterator protocol.

```
for images, labels in train_gen:
    ...
    break
```

- `images.shape → (32, 256, 256, 3)`
- `labels.shape → (32,)`

Each image is augmented differently every epoch — which increases dataset diversity.

If `save_to_dir` is used, the augmented images are physically saved to disk (for demonstration).

✅ Building & Training the CNN Model

The CNN architecture remains the same as previous videos:

- Convolution layers
- MaxPooling
- Flatten

- Dense layers

Compiled with:

```
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)
```

✅ Using Model.fit() With Generators

Unlike training with arrays, here we specify:

- `steps_per_epoch`
- `validation_steps`

Example:

```
steps_per_epoch = num_train_images // 32
validation_steps = num_val_images // 32

model.fit(
    train_gen,
    steps_per_epoch=steps_per_epoch,
    validation_data=val_gen,
    validation_steps=validation_steps,
    epochs=20
)
```

This ensures each epoch processes the full dataset.

✅ Results

On training:

✅ ~97% training accuracy

✅ ~95% validation accuracy

Evaluation on unseen test data:

✅ ~97% test accuracy

The high test score confirms the model generalizes well.

✅ Making Predictions on Images

Generates a batch from the test generator, and predictions are made on those images:

```
for img_batch, label_batch in test_gen:
    pred = model.predict(img_batch)
    break
```

Predictions are compared to true labels.

Save model in TensorFlow SavedModel format

```
model.export("../saved_models/my_aug_model_pb")
```

Similar same steps followed for FastAPI for main-aug.py

```
                main-aug.py

# ============================================
# ✅ FASTAPI APP FOR POTATO DISEASE CLASSIFICATION
# Uses TensorFlow SavedModel (.pb format)
```

```
# ================================================

from fastapi import FastAPI, File, UploadFile
from fastapi.middleware.cors import CORSMiddleware
import uvicorn
import numpy as np
from io import BytesIO
from PIL import Image
import tensorflow as tf

# ------------------------------------------------
# 🚀 Initialize FastAPI app
# ------------------------------------------------
app = FastAPI()

# ------------------------------------------------
# 🌐 Allow CORS (for frontend access, e.g. React)
# ------------------------------------------------
origins = ["http://localhost", "http://localhost:3000"]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# ================================================================
# 🧠 LOAD MODEL (TensorFlow SavedModel .pb format)
# ================================================================
model = tf.saved_model.load("../saved_models/my_aug_model_pb")

# ------------------------------------------------
```

```python
# 🏷️ Define class labels
# ------------------------------------------------
CLASS_NAMES = ["Early Blight", "Late Blight", "Healthy"]

# ------------------------------------------------
# 🔍 Health check route
# ------------------------------------------------
@app.get("/ping")
async def ping():
    return {"message": "Hello, I am alive"}

# ------------------------------------------------
# 🖼️ Helper function — convert uploaded image to NumPy array
# ------------------------------------------------
def read_file_as_image(data) -> np.ndarray:
    """
    Convert uploaded image bytes into a normalized NumPy array.
    """
    image = Image.open(BytesIO(data)).convert("RGB")
    image = image.resize((256, 256))    # Resize to match model input
    image = np.array(image) / 255.0     # Normalize pixel values (0–1)
    return image

# ------------------------------------------------
# 🤖 Prediction endpoint
# ------------------------------------------------
@app.post("/predict")
async def predict(file: UploadFile = File(...)):
    # Read and preprocess image
    image = read_file_as_image(await file.read())
    img_batch = np.expand_dims(image, axis=0)  # Add batch dimension

    # Perform prediction using the SavedModel signature
    infer = model.signatures["serving_default"]
    prediction = infer(tf.constant(img_batch, dtype=tf.float32))
    predictions = list(prediction.values())[0].numpy()
```

```
    predicted_class = CLASS_NAMES[np.argmax(predictions[0])]
    confidence = float(np.max(predictions[0]))

    # Return response
    return {
        "class": predicted_class,
        "confidence": confidence
    }

# ------------------------------------------------
# ▶️ Run FastAPI app
# ------------------------------------------------
if __name__ == "__main__":
    uvicorn.run(app, host="localhost", port=8001)
```

✅ Deep Insights (From Your Summary)

- ImageDataGenerator **reduces boilerplate**, making augmentation configuration easier.
- Its lack of native train/val/test split highlights a **gap** in TensorFlow's API.
- Hands-on practice is essential — this is emphasized heavily in the content.

FastAPI backend (main.py / main-aug.py) — and where exactly you need to edit things like API URLs or endpoints (`/predict` , `/ping` , etc.).

🧠 1️⃣ BACKEND SIDE (FastAPI)

You already have two backend files:

- `main.py` → base model (maybe normal dataset)
- `main-aug.py` → augmented dataset version (or another model version)

Both of them expose **the same FastAPI endpoint structure**, like:

```
GET /ping
POST /predict
```

You can run **only one backend at a time** (on different ports if you like).

Example:

```
# Run normal model
uvicorn main:app --host localhost --port 8000

# Run augmented model
uvicorn main_aug:app --host localhost --port 8001
```

👉 Each one will have a different URL:

- Normal → `http://localhost:8000/predict`
- Augmented → `http://localhost:8001/predict`

💻2️⃣ FRONTEND SIDE (React)

Your frontend (React) doesn't directly "take" the model —

it just **sends an image** to your backend **via API** and **gets prediction results** (class + confidence).

You'll usually find this logic inside:

```
frontend/src/pages/Home.js
```

or sometimes:

```
frontend/src/components/Upload.js
```

✅ Typical Code Snippet (inside `Home.js` or similar)

Find the function that uploads the image — it'll look like this:

```
const [selectedImage, setSelectedImage] = useState(null);
const [prediction, setPrediction] = useState(null);

const handleUpload = async () => {
  const formData = new FormData();
  formData.append("file", selectedImage);

  const response = await fetch("http://localhost:8000/predict", {
    method: "POST",
    body: formData,
  });

  const data = await response.json();
  setPrediction(data);
};
```

🟢 Edit the URL here —

that's where you switch between your `main.py` and `main-aug.py` backend models.

✨ Example: Switching Between Models

If you want your frontend to use the **augmented model**, change this line:

```
const response = await fetch("http://localhost:8001/predict", {  // use port 800
1
```

If you switch back to the **original model**, use:

```
const response = await fetch("http://localhost:8000/predict", {  // use port 80
00
```

🧩 3️⃣ Optional: Dynamic Switching Between Models

You can even add a **dropdown button** in your frontend to let users choose which model to use:

```
const [modelType, setModelType] = useState("main");

const baseURL = modelType === "main"
  ? "http://localhost:8000/predict"
  : "http://localhost:8001/predict";
```

Then use:

```
const response = await fetch(baseURL, {
  method: "POST",
  body: formData,
});
```

Now users can switch between models directly from the frontend.

🧾 4️⃣ Frontend Folder Path Summary

File Name	Purpose	What to Edit
Home.js or **Upload.js**	Where image is uploaded and API is called	Change the fetch URL to correct FastAPI port
App.js	Routing/navigation	Usually no change needed

File Name	Purpose	What to Edit
backend/main.py	Base model (architecture + weights)	Runs on 8000
backend/main-aug.py	Augmented model version	Runs on 8001

🧠 In Simple Words

> 🔷 Frontend never loads the model directly.
>
> It only sends the image to FastAPI.
>
> FastAPI loads the model, predicts, and sends results back as JSON.

🔥 IMPORTANT: Backend must stay running WHILE frontend runs

You need **two terminals open**:

🟦 Terminal 1 → Backend (keep it open)

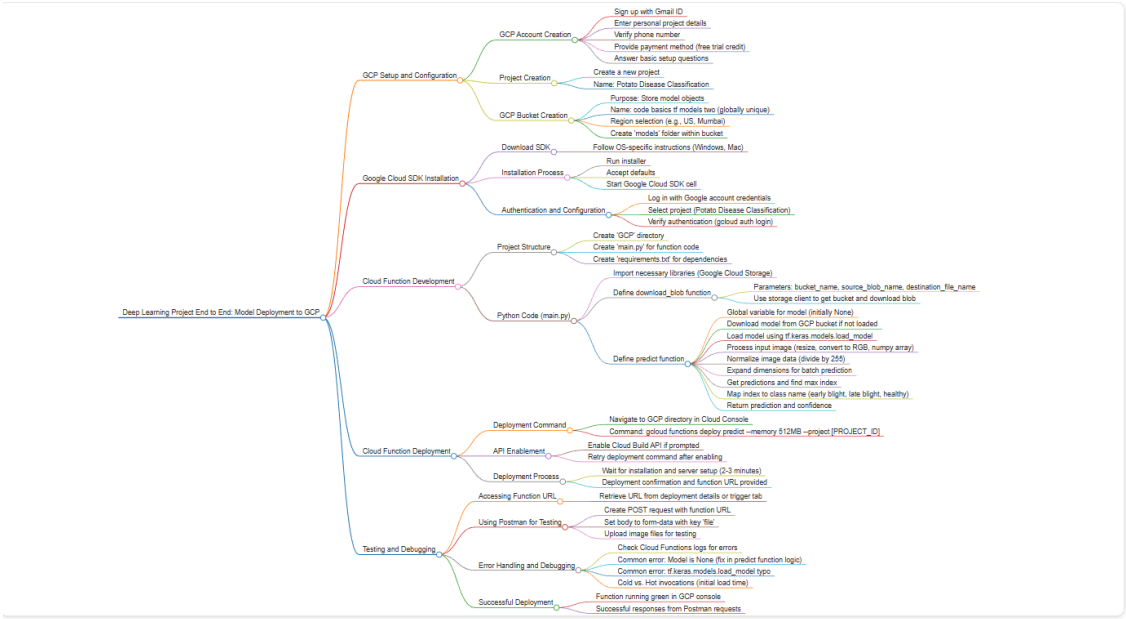
```
python main-aug.py
```

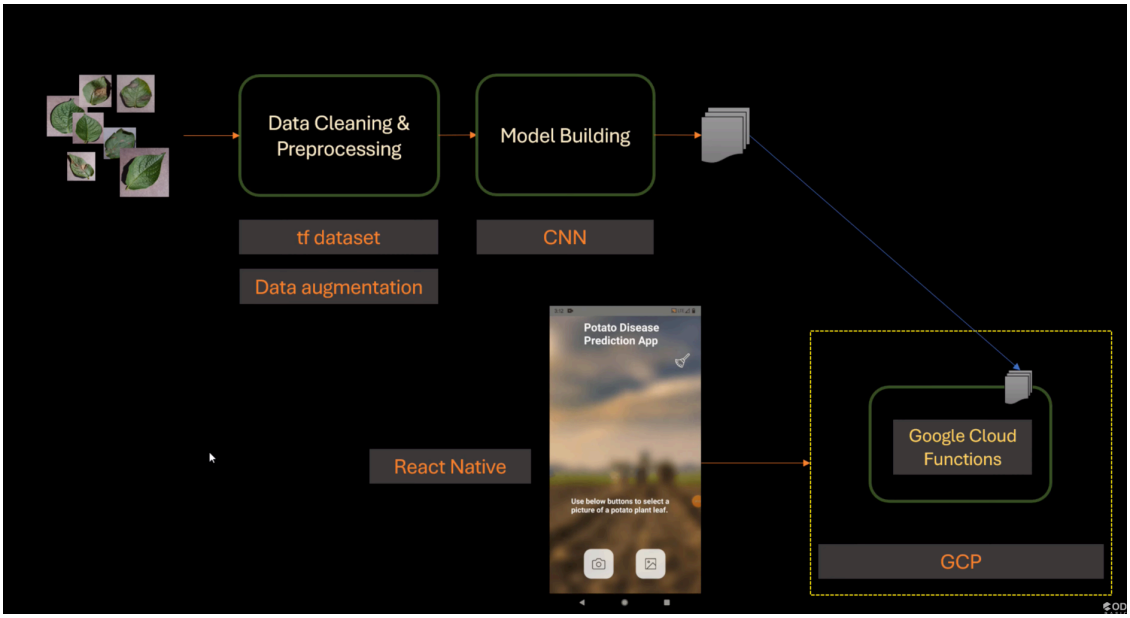
🟩 Terminal 2 → Frontend

```
npm start
```

If you run frontend in the same terminal → backend stops → React cannot connect → ERR_CONNECTION_REFUSED.

7. Model Deployment To GCP





⭐ STEP 1 — Create & Set Up GCP Account

1. Go to Google Cloud Console.
2. Create a project → **note the Project ID**.
3. Enable **Cloud Functions API** and **Cloud Storage API**.

⭐ STEP 2 — Create a Bucket

1. Go to *Cloud Storage → Buckets*
2. Click **Create Bucket**
3. Name it (example):

```
potato-disease-bucket
```

⭐ STEP 3 — Upload Your SavedModel Folder

You need to upload the **entire folder**, not a single file.

Upload:

```
saved_models/my_aug_model_pb/
```

Bucket structure becomes:

```
gs://potato-disease-bucket/saved_models/my_aug_model_pb/saved_model.pb
```

⭐ STEP 4 — Install Google Cloud SDK

Download: https://cloud.google.com/sdk

After install:

```
gcloud auth login
gcloud config set project YOUR_PROJECT_ID
```

⭐ STEP 5 — Update Your Prediction Code

In your Cloud Function, replace the model path with:

```
MODEL_PATH = "gs://potato-disease-bucket/saved_models/my_aug_model_pb"
model = tf.saved_model.load(MODEL_PATH)
```

⭐ STEP 6 — Deploy Cloud Function

Inside your **gcp/** folder (where main.py + requirements.txt exist):

```
cd gcp
gcloud functions deploy predict \
  --runtime python38 \
  --trigger-http \
  --memory 1024MB \
  --entry-point predict \
  --project YOUR_PROJECT_ID
```

⚠️ You may need **1024 MB** for SavedModel.

⭐ STEP 7 — Get the Trigger URL

After deployment, GCP will show URL like:

```
https://REGION-PROJECTID.cloudfunctions.net/predict
```

⭐ STEP 8 — Test in Postman

Set:

- Method: **POST**
- Body: **form-data**
  - key: `file` → upload image
- Hit **Send**

You'll get JSON prediction results.

🎉 Done!

Your TF SavedModel (`my_aug_model_pb`) is now deployed on GCP.

This Cloud Function implements an HTTP endpoint `predict(request)` that:

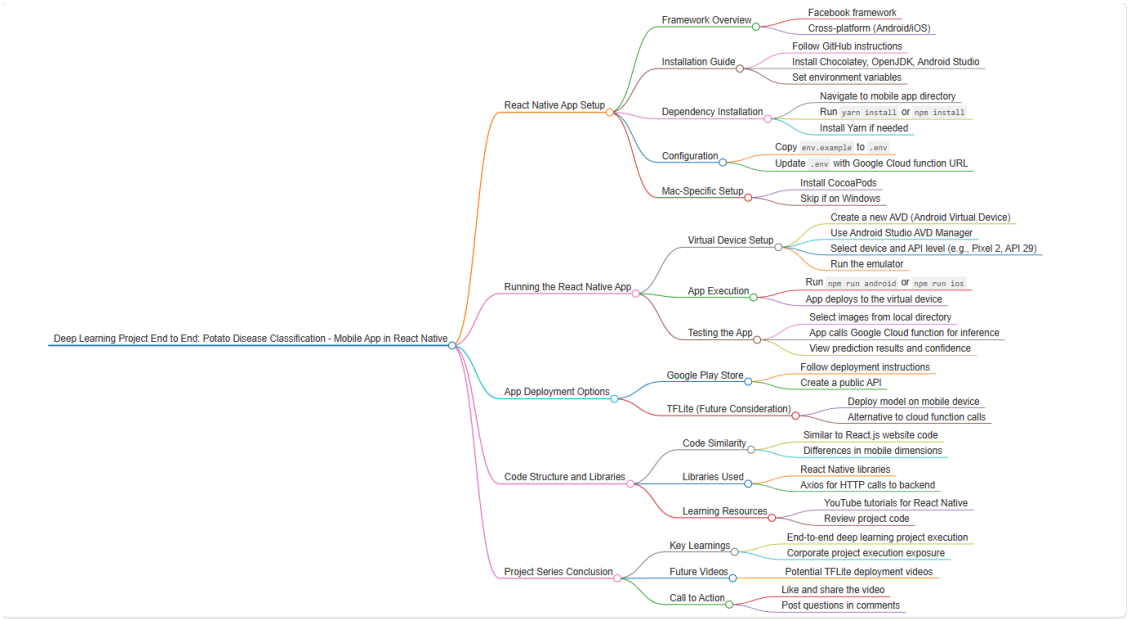
- on first invocation downloads a Keras `.h5` model from a GCS bucket into the function's ephemeral filesystem (`/tmp`),
- loads it with `tf.keras.models.load_model`,
- accepts an uploaded image via form-data (`file` key),
- preprocesses the image (RGB, resize 256×256, normalize),
- runs `model.predict(...)`,
- finds the predicted class and confidence, and returns a JSON-like response.

Top-level globals

```
model = None
interpreter = None
input_index = None
output_index = None
class_names = ["Early Blight", "Late Blight", "Healthy"]
BUCKET_NAME = "codebasics-tf-models"
```

- `model` is a global cache for the loaded Keras model so the code downloads & loads it only once per cold start.
- `interpreter`, `input_index`, `output_index` appear intended for TF Lite usage but are unused in this `.h5` Keras setup. They can be removed to simplify the code.
- `class_names` maps model output indices → human-readable labels.
- `BUCKET_NAME` should be set to your GCS bucket name; don't hardcode secrets — this is fine for a public project but consider using environment variables for flexibility.

8. Mobile App in React Native



This content explains how to build and run a React Native mobile application that sends images to a Google Cloud Function for deep learning predictions. It completes the end-to-end workflow—from model development to deployment inside a fully functional mobile app.

Key Points

1. React Native Setup

Users are guided through configuring the React Native environment, including:

- Installing **Chocolatey**, **OpenJDK**, **Android Studio**, and **Yarn** (Windows example).
- Selecting the non-Expo React Native setup instructions.
- Carefully following each step—missing any step can cause errors.

2. Project Initialization

After cloning the repository:

- Navigate to the **mobile-app** folder.
- Run **yarn install** to install all dependencies (similar to pip installing Python libraries).
- Copy `env.example` to `.env` and replace the API URL with your own Google Cloud Function endpoint from the previous video.

3. Setting Up Android Virtual Device (AVD)

Before running the app:

- Open **Android Studio → Tools → AVD Manager**.
- Create a new virtual device (e.g., Pixel 2, API Level 29).
- Launch the emulator, which acts like a real Android phone for testing the app.

4. Running the Mobile App

- Use **npm run android** (or **npm run ios** on Mac).

- The emulator starts, and the React Native app is deployed into it.
- Inside the app, selecting an image triggers:
  - Uploading the image to the Google Cloud Function.
  - Running model inference on the backend.
  - Displaying prediction results (e.g., 76.68% confidence).

5. Deployment

The documentation includes a section on:

- Publishing the app to the **Google Play Store**.
- Setting up a public API endpoint for production usage.

6. Code Insight

- The mobile app is built with **React Native**, while the earlier web app was in **React.js**.
- Much of the logic is similar.
- Uses **Axios** to send HTTP requests to the Cloud Function.
- UI adjustments are specific to mobile layouts and dimensions.

7. Future Enhancements

The series may later include tutorials on deploying **TensorFlow Lite (TFLite)** directly on the device for offline inference, removing the need for cloud calls.

Deep Insights

This tutorial bridges the gap between AI model development and real-world app deployment, offering:

- Full-stack project exposure.
- A practical demonstration of integrating cloud-based ML with mobile apps.
- A realistic corporate-style end-to-end deep learning project experience.

Major Challenges Faced in the Potato Disease Classification Project

1️⃣ Library Compatibility

Initially, there were compatibility issues between TensorFlow, Keras, and other libraries.

Some functions were deprecated in the newer versions, causing model loading and preprocessing errors.

I solved this by creating a clean virtual environment and fixing all library versions to ensure smooth training and deployment.

2️⃣ Image Processing Errors

The dataset contained images of different sizes, lighting conditions, and color formats, leading to shape mismatch and poor training results.

I resolved this by resizing all images to a uniform dimension (256×256), normalizing pixel values, and applying proper preprocessing and augmentation.

3️⃣ Model Loading and Architecture Issues

At first, the model could not be loaded properly — only weights were getting restored, not the full architecture.

I fixed this by saving the model in TensorFlow's **SavedModel (.pb)** format, which includes both the structure and weights, ensuring consistent predictions during inference.

4️⃣ Hyperparameter Tuning and Model Training

Finding the right learning rate, batch size, and number of epochs was challenging.

The model also faced overfitting due to limited data.

I handled this by using regularization, dropout, early stopping, and systematic hyperparameter tuning to improve performance.

5️⃣ Dataset Quality and Imbalance

The dataset had uneven samples across classes and some noisy images.

I performed data cleaning and applied augmentation techniques (rotation, flipping, zoom) to balance the dataset, which improved model generalization and accuracy.

6️⃣ Model Deployment and Performance

Deployment brought new challenges — large model size and version conflicts between training and serving environments.

I optimized the model, reduced its size, and integrated it successfully using Flask API and TensorFlow Serving for real-time predictions.

✅ Summary for Interview (Short Version)

> "The main challenges I faced were — library compatibility, image preprocessing, model loading, hyperparameter tuning, data imbalance, and deployment optimization.
>
> I systematically solved them by version control, preprocessing pipelines, proper model saving/loading, data augmentation, and optimization during deployment."