Problem Statement:

Ninjacart, India's largest fresh produce supply chain company, aims to enhance its automation process by developing a robust classifier capable of distinguishing between different types of vegetables (onion, potato, tomato) and noise (Indian market scenes) in images. The classifier will be utilized to identify vegetables accurately for efficient supply chain management.

Objective:

The primary objective is to develop a multiclass classifier using computer vision techniques to accurately identify vegetables (onion, potato, tomato) and differentiate them from noise (Indian market scenes) in images. The classifier should be capable of accurately classifying images to improve efficiency in sourcing, sorting, and distribution processes.

Key Components:

1. **Dataset Preparation:** Collect and preprocess a dataset consisting of images of vegetables and Indian market scenes obtained from online sources.
2. **Model Development:** Design and train Convolutional Neural Network (CNN) models using TensorFlow and Keras libraries to classify images into vegetable categories and noise.
3. **Model Evaluation:** Evaluate the performance of trained models using metrics such as accuracy, precision, recall, F1-score, and confusion matrix analysis.
4. **Hyperparameter Tuning:** Optimize model hyperparameters to improve classification accuracy and generalization.
5. **Deployment:** Deploy the trained model to a suitable platform for integration into Ninjacart's automation system, enabling real-time classification of vegetable images.

Constraints and Requirements:

1. **Accuracy Threshold:** Achieve a minimum accuracy threshold specified by Ninjacart for reliable classification performance.
2. **Computational Resources:** Develop models considering computational resource constraints to ensure feasibility for deployment in production environments.
3. **Model Interpretability:** Ensure the developed classifier is interpretable, allowing users to understand and trust its predictions in real-world scenarios.

By addressing these components and meeting the defined objectives, Ninjacart aims to enhance its supply chain automation process by leveraging innovative computer vision technology for vegetable classification.

In [1]:
```
# Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import os
import cv2
from glob import glob
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
```

1. Download & Unzip the Dataset

- We will use the gdown library to download the dataset from the provided Google Drive link.
- After downloading, we will unzip the compressed file to access the train and test folders.

Download the dataset

```
import gdown

    url = "https://drive.google.com/uc?id=1clZX-lV_MLxKHSyeyTheX5OCQtNCUcqT"

    output = "ninjacart_data.zip" gdown.download(url, output, quiet=False)
```

Unzip the dataset

```
import zipfile

    with zipfile.ZipFile("ninjacart_data.zip", "r") as z:

    z.extractall()
```

- This will create a dataset folder containing the train and test sub-folders.

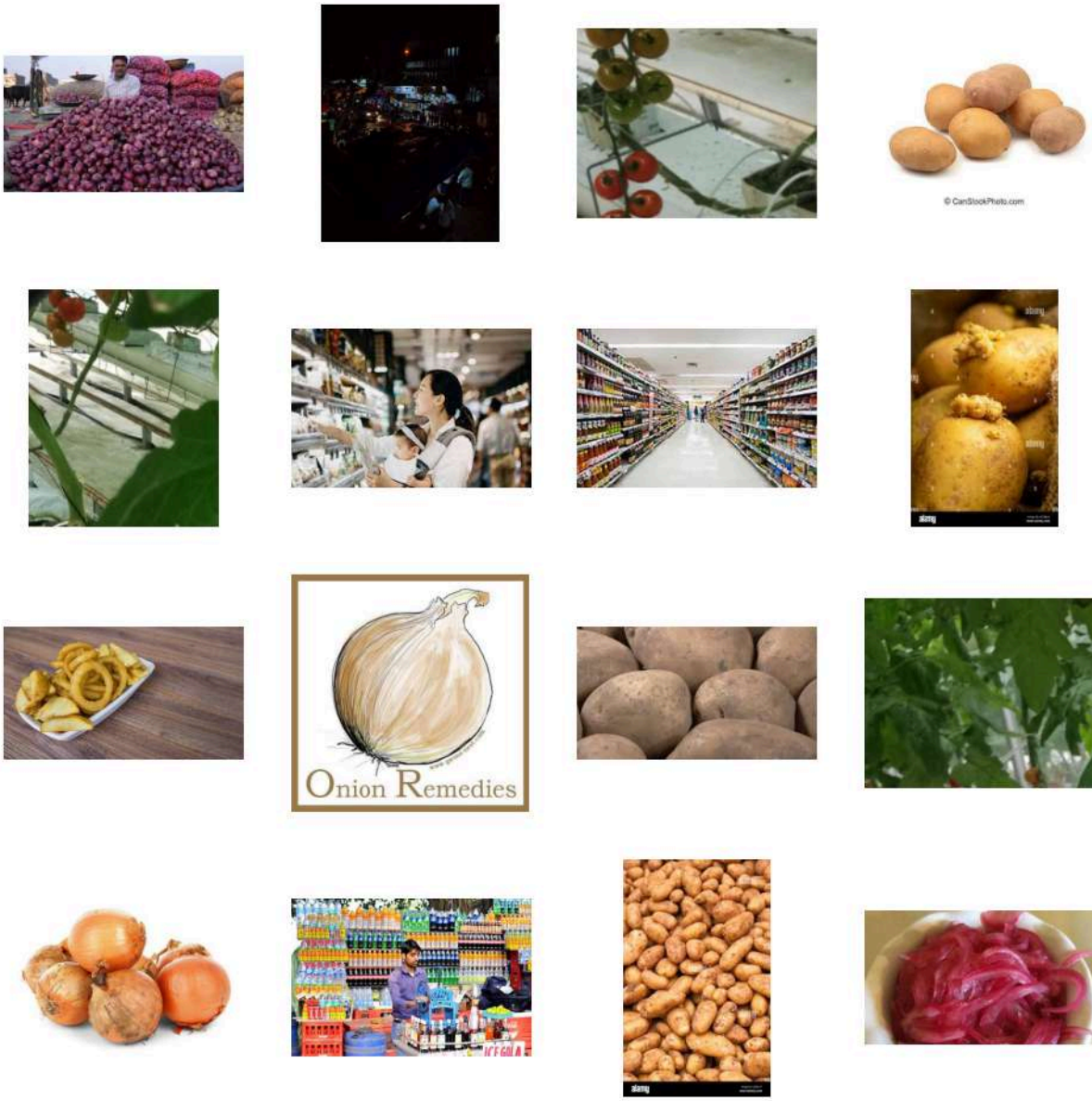
In [2]:
```
# Combine file paths for all three formats
train_paths = glob("ninjacart_data/train/*/*.jpeg") + \
                  glob("ninjacart_data/train/*/*.jpg") + \
                  glob("ninjacart_data/train/*/*.png")

test_paths = glob("ninjacart_data/test/*/*.jpeg") + \
                  glob("ninjacart_data/test/*/*.jpg") + \
                  glob("ninjacart_data/test/*/*.png")

print(f"Total training images: {len(train_paths)}")
print(f"Total test images: {len(test_paths)}")
```

```
Total training images: 3135
Total test images: 351
```

2. Data Visualization and Exploration

- Visualize the data to understand its distribution and characteristics.
- Plot sample images from each class to get an idea of the data.

In [3]:
```
# Plot a grid of sample images
fig, ax = plt.subplots(nrows=4, ncols=4, figsize=(12, 12))
for i, image_path in enumerate(np.random.choice(train_paths, 16, replace=False)):
    img = cv2.imread(image_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    row, col = i // 4, i % 4
    ax[row, col].imshow(img)
    ax[row, col].axis('off')

plt.show()
```



In [4]:

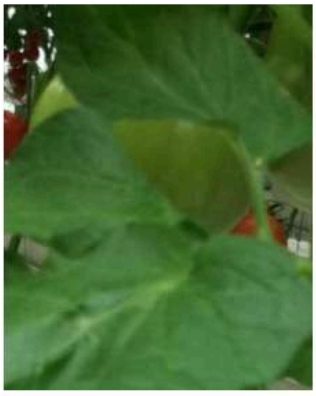
```
# List of class names
class_names = ['indian market', 'onion', 'potato', 'tomato']

# Plot a few images from each class
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(12, 8))
for i, class_name in enumerate(class_names):
    # Construct the directory path for the current class
    class_dir = os.path.join('ninjacart_data', 'train', class_name)

    # Get a list of image files in the directory (assuming all files are images)
    class_files = [os.path.join(class_dir, file) for file in os.listdir(class_dir)]

    # Choose a random image path from the class
    img_path = np.random.choice(class_files)

    # Read and display the image
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    row, col = i // 2, i % 2
    ax[row, col].imshow(img)
    ax[row, col].set_title(class_name)
    ax[row, col].axis('off')

plt.show()
```

indian market

onion

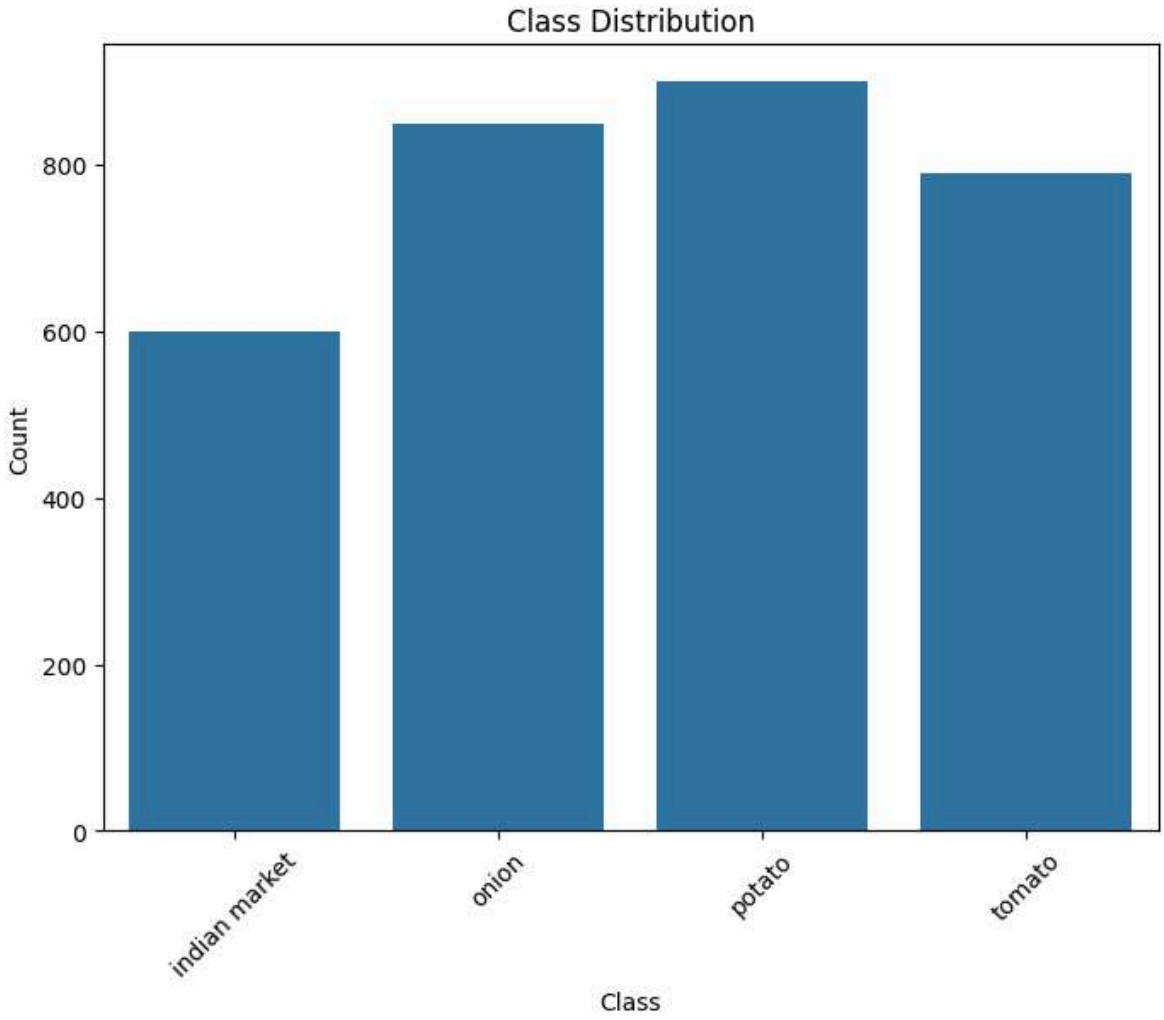
potato

tomato



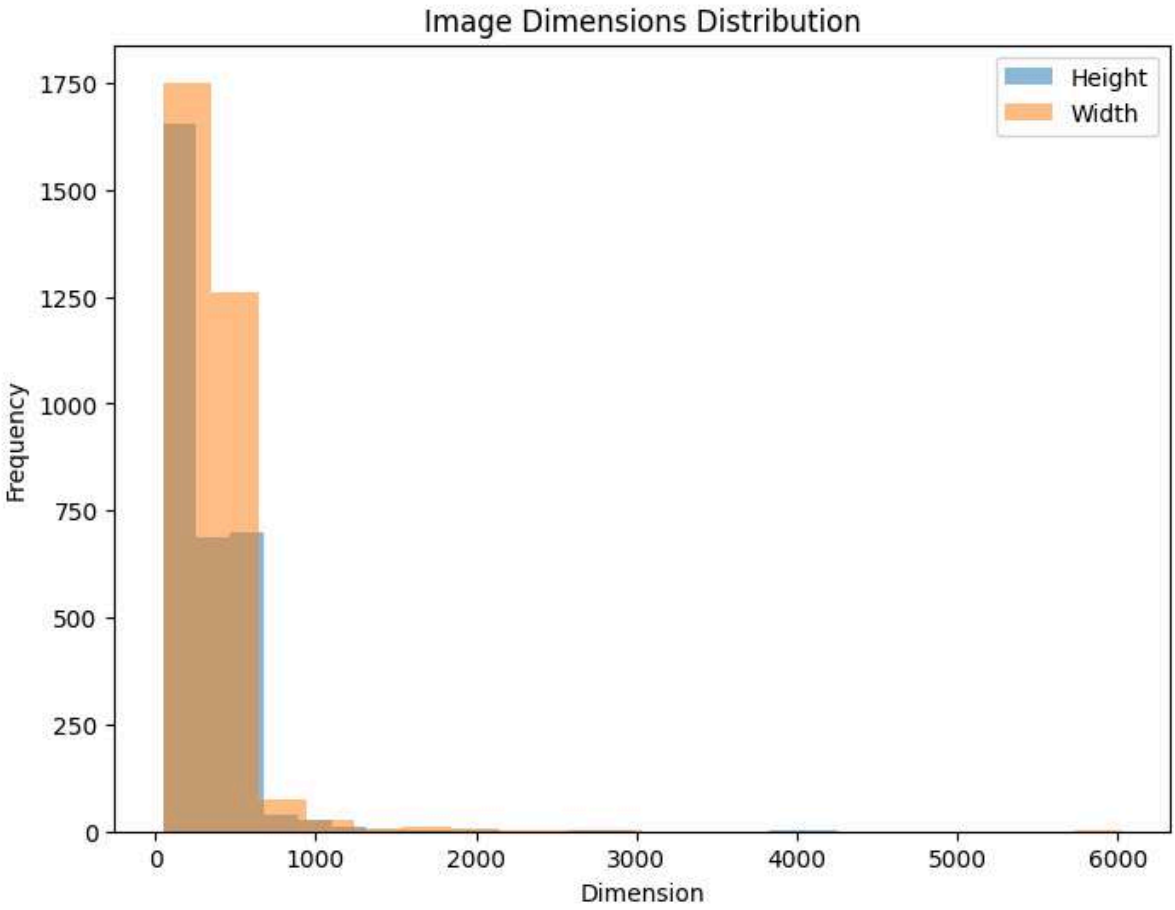
i. Class Distribution

In [5]:
```
# Count the number of images in each class
class_counts = {class_name: len(glob(os.path.join('ninjacart_data', 'train', class_name, '*')))
                for class_name in class_names}

# Plot class distribution
plt.figure(figsize=(8, 6))
sns.barplot(x=list(class_counts.keys()), y=list(class_counts.values()))
plt.title('Class Distribution')
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()
```



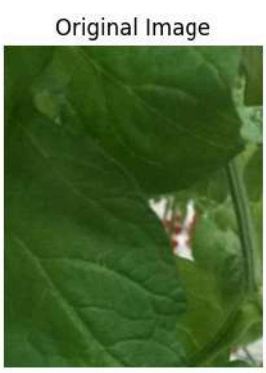
ii. Image Dimensions

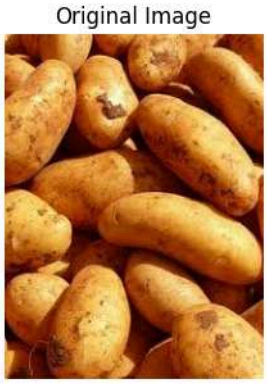
In [6]:
```
# Get dimensions of all images
image_dimensions = [(cv2.imread(image_path).shape[0], cv2.imread(image_path).shape[1])
                    for image_path in train_paths]

# Plot histogram of image dimensions
plt.figure(figsize=(8, 6))
plt.hist([dim[0] for dim in image_dimensions], bins=20, alpha=0.5, label='Height')
plt.hist([dim[1] for dim in image_dimensions], bins=20, alpha=0.5, label='Width')
plt.title('Image Dimensions Distribution')
plt.xlabel('Dimension')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

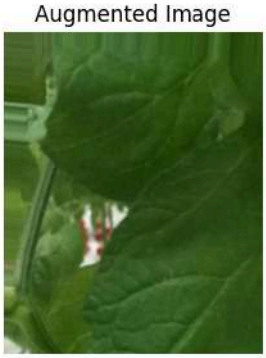


iii. Image Augmentation

In [7]:
```
# Define image augmentation parameters
datagen = ImageDataGenerator(rotation_range=20, width_shift_range=0.1, height_shift_range=0.1,
                             shear_range=0.2, zoom_range=0.2, horizontal_flip=True, vertical_flip

# Plot augmented images
plt.figure(figsize=(12, 12))
for i, image_path in enumerate(np.random.choice(train_paths, 4, replace=False)):
    img = cv2.imread(image_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    ax = plt.subplot(2, 4, i + 1)  # Adjusted subplot grid to accommodate 4 images
    ax.set_title('Original Image')
    ax.imshow(img)
    ax.axis('off')
    ax = plt.subplot(2, 4, i + 5)  # Adjusted indexing
    ax.set_title('Augmented Image')
    ax.imshow(datagen.random_transform(img))
    ax.axis('off')
plt.show()
```

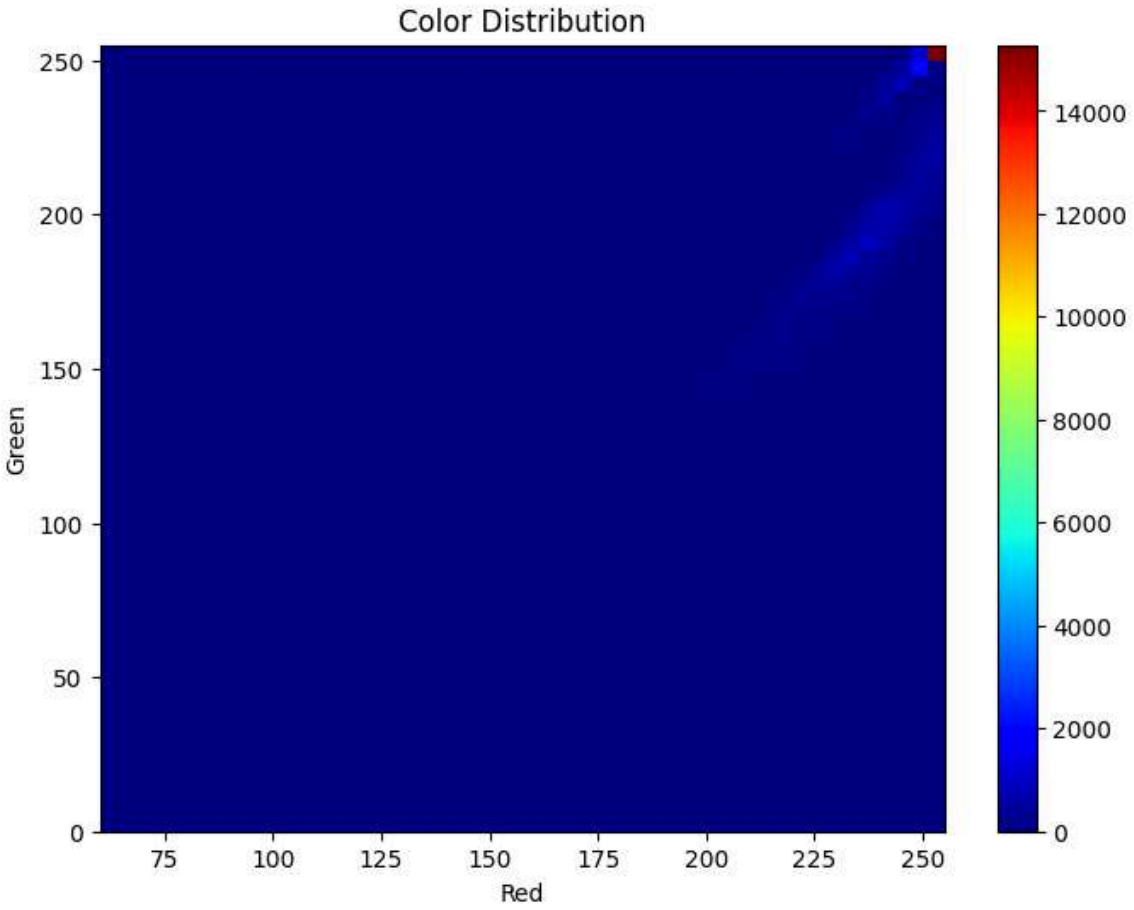






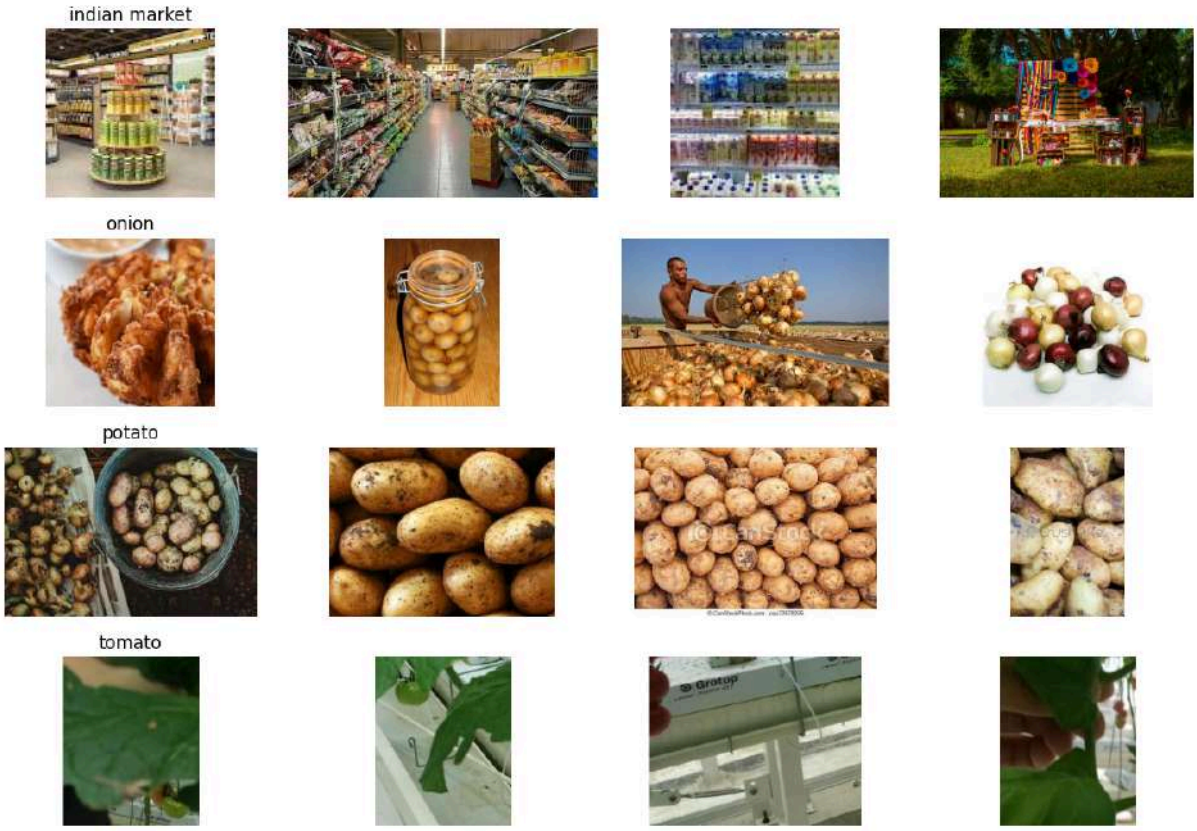
iv. Color Distribution

In [8]:
```python
# Function to plot color distribution
def plot_color_distribution(image_path):
    img = cv2.imread(image_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    colors = img.reshape(-1, 3)
    plt.figure(figsize=(8, 6))
    plt.hist2d(colors[:, 0], colors[:, 1], bins=(50, 50), cmap=plt.cm.jet)
    plt.title('Color Distribution')
    plt.xlabel('Red')
    plt.ylabel('Green')
    plt.colorbar()
    plt.show()

# Plot color distribution for a sample image
plot_color_distribution(np.random.choice(train_paths))
```



v. Error Analysis

In [9]:
```
# Plot a few random images from each class for inspection
plt.figure(figsize=(12, 8))
for i, class_name in enumerate(class_names):
    # Construct the directory path for the current class
    class_dir = os.path.join('ninjacart_data', 'train', class_name)

    # Get a list of image files in the directory (assuming all files are images)
    class_files = [os.path.join(class_dir, file) for file in os.listdir(class_dir)]

    # Choose a random sample of images from the class
    sample_images = np.random.choice(class_files, min(len(class_files), 4), replace=False)

    # Plot the sample images
    for j, img_path in enumerate(sample_images):
        img = cv2.imread(img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        ax = plt.subplot(len(class_names), 4, i*4 + j + 1)
        ax.set_title(class_name if j == 0 else '')  # Only show class name for the first image in
        ax.imshow(img)
        ax.axis('off')

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

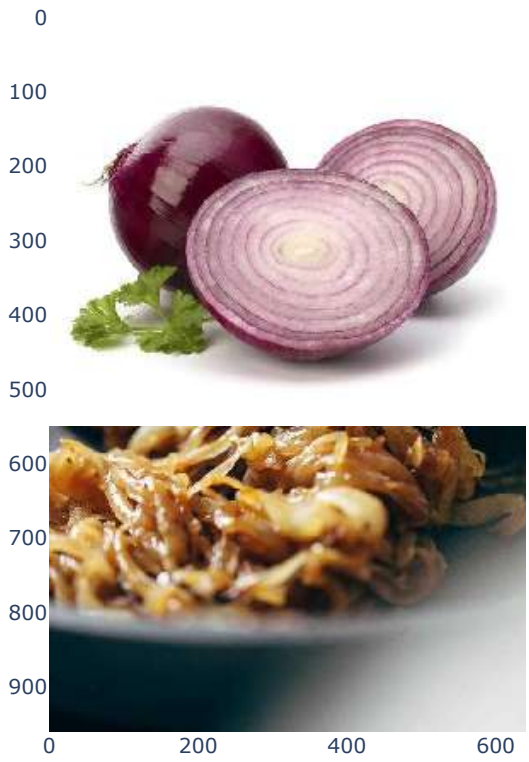


vi. Interactive Visualization

In [10]:

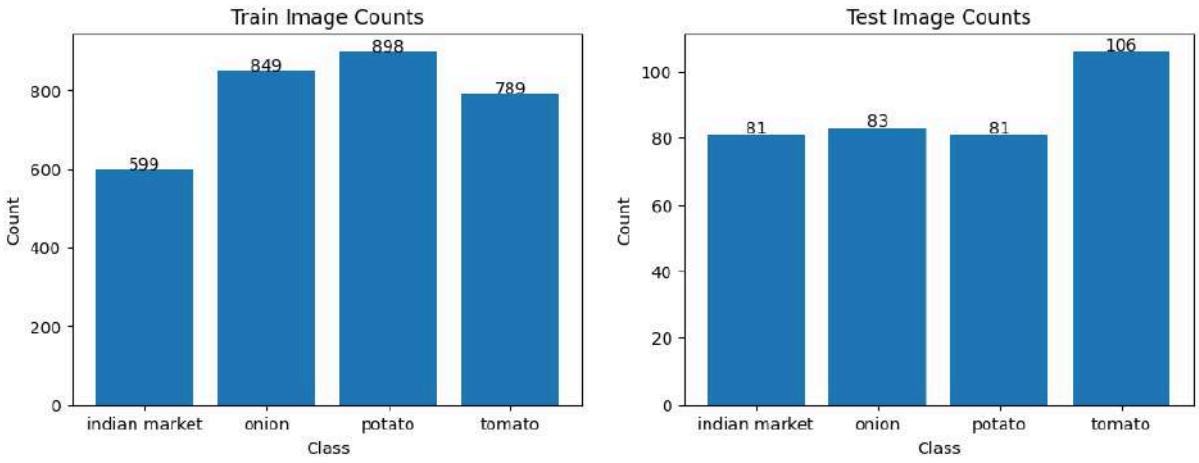
```
import plotly.graph_objs as go

# Define a function to plot interactive images
def plot_interactive_images(image_paths):
    fig = go.Figure()

    # Add images to the figure
    for i, image_path in enumerate(image_paths):
        img = cv2.imread(image_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        fig.add_trace(go.Image(z=img))

    # Configure layout
    fig.update_layout(title='Interactive Image Visualization',
                      showlegend=False,
                      width=800,
                      height=600)

    # Show plot
    fig.show()

# Choose a random sample of images for interactive visualization
sample_image_paths = np.random.choice(train_paths, 9, replace=False)

# Plot interactive images
plot_interactive_images(sample_image_paths)
```

Interactive Image Visualization



vii. Verify Image Counts

In [11]:
```
# Count images in train and test folders
train_counts = {class_name: sum([1 for file in train_paths if class_name in file])
                for class_name in class_names}
test_counts = {class_name: sum([1 for file in test_paths if class_name in file])
               for class_name in class_names}

# Calculate total images
total_train_images = sum(train_counts.values())
total_test_images = sum(test_counts.values())

# Plot histograms
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

for ax, counts, title in zip([ax1, ax2], [train_counts, test_counts],
                             ['Train Image Counts', 'Test Image Counts']):
    ax.bar(counts.keys(), counts.values())
    ax.set_title(title)
    ax.set_xlabel('Class')
    ax.set_ylabel('Count')

    # Adding text annotations for individual class totals
    for i, class_name in enumerate(class_names):
        ax.text(i, counts[class_name] + 0.5, str(counts[class_name]), ha='center')

plt.show()
```



3. Data Preprocessing:

- Data preprocessing is a crucial step in building a machine learning model. Here are some common preprocessing steps:

i. Image Resizing

We may want to resize all images to a consistent size to ensure that they have the same dimensions. This step is important for feeding images into a neural network, as most models expect inputs of fixed size.

ii. Data Augmentation

Data augmentation techniques such as rotation, flipping, and zooming can help increase the diversity of the training dataset, leading to better generalization and improved model performance.

iii. Normalization

Normalize pixel values to a range between 0 and 1 or -1 and 1. Normalization helps in stabilizing and accelerating the training process.

iv. Batch Processing

Batch processing involves dividing the dataset into small batches during training to improve computational efficiency and convergence speed.

v. Handling Class Imbalance

If there's a significant class imbalance in the dataset (i.e., some classes have many more samples than others), we may need to address it using techniques such as oversampling, undersampling, or class weighting.

In [12]:
```
# Define image dimensions
img_height = 224
img_width = 224
batch_size = 32

# Data preprocessing and augmentation
train_datagen = ImageDataGenerator(
    rescale=1./255,
 rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

test_datagen = ImageDataGenerator(rescale=1./255)
```

```
train_generator = train_datagen.flow_from_directory(
    'ninjacart_data/train',
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical'
)

validation_generator = test_datagen.flow_from_directory(
    'ninjacart_data/test',
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical'
)
```

```
Found 3135 images belonging to 4 classes.
Found 351 images belonging to 4 classes.
```

Insights:

- We define the image dimensions (`img_height` and `img_width`) to which all images will be resized.
- We use `ImageDataGenerator` to perform data augmentation on the training set while also rescaling pixel values to the range [0, 1].
- We create separate generators for the training and validation sets using `flow_from_directory` .
- Finally, we specify the directory paths, target size, batch size, and class mode for each generator.

4. Dataset Splitting

- Split the dataset into training and validation sets (80-20 split) for model training and hyperparameter tuning

In [13]:
```
# Import Library
from sklearn.model_selection import train_test_split

# Define the path to the dataset directory
dataset_dir = 'ninjacart_data/train_split'

# Get the list of class names (subdirectories)
class_names = os.listdir(dataset_dir)

# Initialize lists to store file paths and labels
file_paths = []
labels = []

# Iterate over each class and collect file paths and labels
for class_name in class_names:
    class_dir = os.path.join(dataset_dir, class_name)
    class_files = [os.path.join(class_dir, file) for file in os.listdir(class_dir)]
    file_paths.extend(class_files)
    labels.extend([class_name] * len(class_files))

# Split the dataset into training and validation sets (80-20 split)
train_files, val_files, train_labels, val_labels = train_test_split(
    file_paths, labels, test_size=0.2, random_state=42, stratify=labels
)
```

In [14]:
```
# Create directories to organize the data
train_dir = r'ninjacart_data/train_split'
val_dir = r'ninjacart_data/val_split'

for dir_path in [train_dir, val_dir]:
    for class_name in class_names:
        os.makedirs(os.path.join(dir_path, class_name), exist_ok=True)

# Move training images to the train directory
for file, label in zip(train_files, train_labels):
    dst_dir = os.path.join(train_dir, label)
    os.replace(file, os.path.join(dst_dir, os.path.basename(file)))
```

```
# Move validation images to the validation directory
for file, label in zip(val_files, val_labels):
    dst_dir = os.path.join(val_dir, label)
    os.replace(file, os.path.join(dst_dir, os.path.basename(file)))

print("Dataset split completed successfully.")
```

```
Dataset split completed successfully.
```

Insights:

- We first define the path to the dataset directory (`dataset_dir`).
- We collect file paths and corresponding labels for all images in the dataset.
- We use `train_test_split` from the `sklearn.model_selection` module to split the dataset into training and validation sets, with an 80-20 ratio.
- We create separate directories (`train_dir` and `val_dir`) to organize the split data.
- Finally, we move images to their respective directories based on the split.

After executing this code, we will have separate directories (`train_split` and `val_split`) containing training and validation images, respectively, organized by class.

5. Model Selection and Training

a. Defining the CNN Classifier Model from Scratch

- We can define a basic CNN classifier model using TensorFlow/Keras

In [15]:
```
#Import Libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# Define input shape
input_shape = (img_height, img_width, 3)

# Initialize the model
model = Sequential()

# Add convolutional layers
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten the feature maps
model.add(Flatten())

# Add fully connected layers
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(len(class_names), activation='softmax'))  # Output layer

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Print model summary
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 54, 54, 64)	0
flatten (Flatten)	(None, 186624)	0
dense (Dense)	(None, 128)	23,888,000
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 4)	516

Total params: 23,907,908 (91.20 MB)

Trainable params: 23,907,908 (91.20 MB)

Non-trainable params: 0 (0.00 B)

Insights:

- We define the input shape based on the dimensions of the images.
- We initialize a Sequential model.
- We add convolutional layers followed by max-pooling layers to extract features from the images.
- We flatten the feature maps and add fully connected layers for classification.
- We compile the model with an optimizer, loss function, and evaluation metrics.

b. Improving Baseline CNN to Reduce Overfitting

- To reduce overfitting, we can add dropout layers and apply regularization.

Here's how we can improve the baseline CNN model:

In [16]:
```
#Import Library
from tensorflow.keras import regularizers

# Initialize the model
model = Sequential()

# Add convolutional layers with dropout and regularization
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model.add(Dropout(0.5))
model.add(Dense(len(class_names), activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Print model summary
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_3 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 54, 54, 64)	0
flatten_1 (Flatten)	(None, 186624)	0
dense_2 (Dense)	(None, 128)	23,888,000
dropout_1 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 4)	516

Total params: 23,907,908 (91.20 MB)

Trainable params: 23,907,908 (91.20 MB)

Non-trainable params: 0 (0.00 B)

Insights:

- We add dropout layers and apply L2 regularization to the convolutional and fully connected layers.

c. Implementing Callbacks while Training the Model

- Callbacks are used to perform actions at various stages of the training process, such as saving model checkpoints, early stopping, or adjusting learning rates.

Here's an example of implementing callbacks:

In [17]:
```
# Import Library
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

# Define callbacks
checkpoint = ModelCheckpoint("best_model.keras", monitor='val_accuracy', verbose=1, save_best_onl
early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1, restore_best_weights=Tr

# Define the number of epochs
epochs = 5  # set the number of epochs to 5

# Train the model with callbacks
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // batch_size,
    callbacks=[checkpoint, early_stopping]
)
```

```
Epoch 1/5
97/97 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.4394 - loss: 4.0412
Epoch 1: val_accuracy improved from -inf to 0.71875, saving model to best_model.keras
97/97 ━━━━━━━━━━━━━━━━━━━━ 110s 1s/step - accuracy: 0.4408 - loss: 4.0255 - val_accuracy: 0.7188 -
val_loss: 1.2803
Epoch 2/5
 1/97 ━━━━━━━━━━━━━━━━━━━━ 1:23 869ms/step - accuracy: 0.8438 - loss: 1.1503
Epoch 2: val_accuracy improved from 0.71875 to 0.77419, saving model to best_model.keras
97/97 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.8438 - loss: 0.5811 - val_accuracy: 0.7742 -
val_loss: 0.5556
Epoch 3/5
97/97 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.7558 - loss: 1.1905
Epoch 3: val_accuracy did not improve from 0.77419
97/97 ━━━━━━━━━━━━━━━━━━━━ 123s 1s/step - accuracy: 0.7557 - loss: 1.1902 - val_accuracy: 0.7688 -
val_loss: 1.0932
Epoch 4/5
 1/97 ━━━━━━━━━━━━━━━━━━━━ 1:22 854ms/step - accuracy: 0.6875 - loss: 1.3156
Epoch 4: val_accuracy did not improve from 0.77419
97/97 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6875 - loss: 0.6646 - val_accuracy: 0.7742 -
val_loss: 0.5132
Epoch 5/5
97/97 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.7750 - loss: 1.0448
Epoch 5: val_accuracy did not improve from 0.77419
97/97 ━━━━━━━━━━━━━━━━━━━━ 108s 1s/step - accuracy: 0.7749 - loss: 1.0447 - val_accuracy: 0.7250 -
val_loss: 0.9661
Restoring model weights from the end of the best epoch: 4.
```

Insights:

- We define callbacks for model checkpointing and early stopping.
- We pass the callbacks to the `fit` method when training the model.

d. Finetune Pretrained Models

- We can also fine-tune pretrained models such as VGG, ResNet, or MobileNet on your dataset.

Here's an example of how to fine-tune a pretrained VGG16 model:

In [18]:
```python
# Import Libraries
from tensorflow.keras.applications import VGG16
from tensorflow.keras.layers import GlobalAveragePooling2D

# Load pretrained VGG16 model (without top layers)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=input_shape)

# Add custom top layers
model = Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(len(class_names), activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Print model summary
model.summary()
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
vgg16 (Functional)	?	14,714,688
global_average_pooling2d (GlobalAveragePooling2D)	?	0 (unbuilt)
dense_4 (Dense)	?	0 (unbuilt)
dropout_2 (Dropout)	?	0
dense_5 (Dense)	?	0 (unbuilt)

Total params: 14,714,688 (56.13 MB)

Trainable params: 14,714,688 (56.13 MB)

Non-trainable params: 0 (0.00 B)

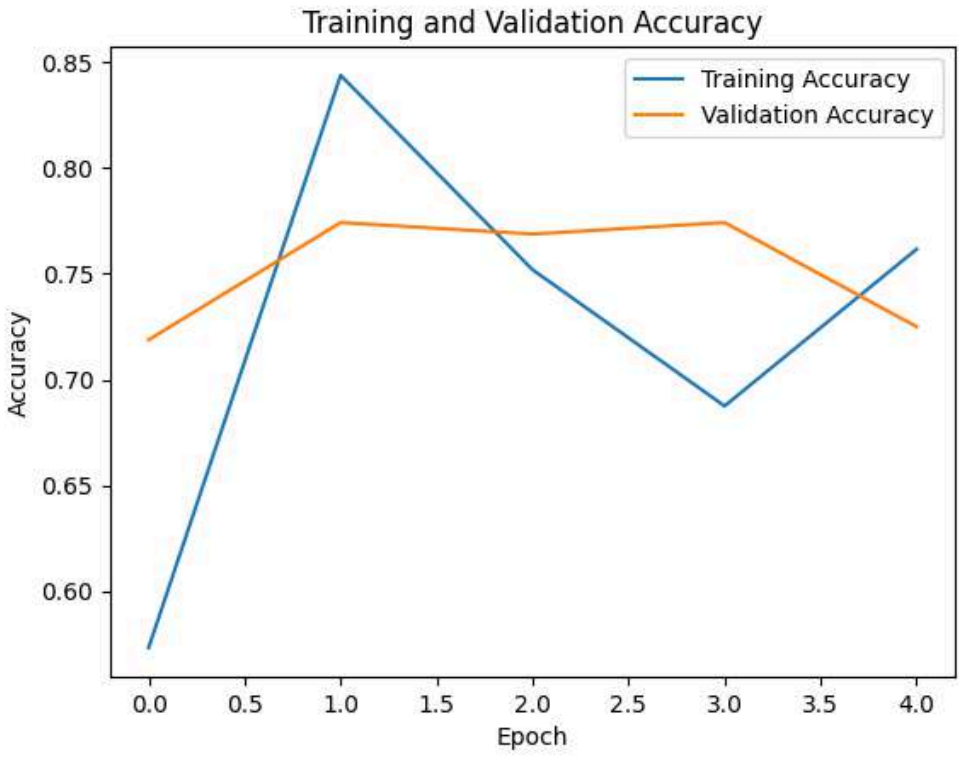
Insights:

- We load the pretrained VGG16 model without the top layers.
- We add custom top layers for classification.
- We compile the model and print the summary.

e. Plotting Model Training Metrics

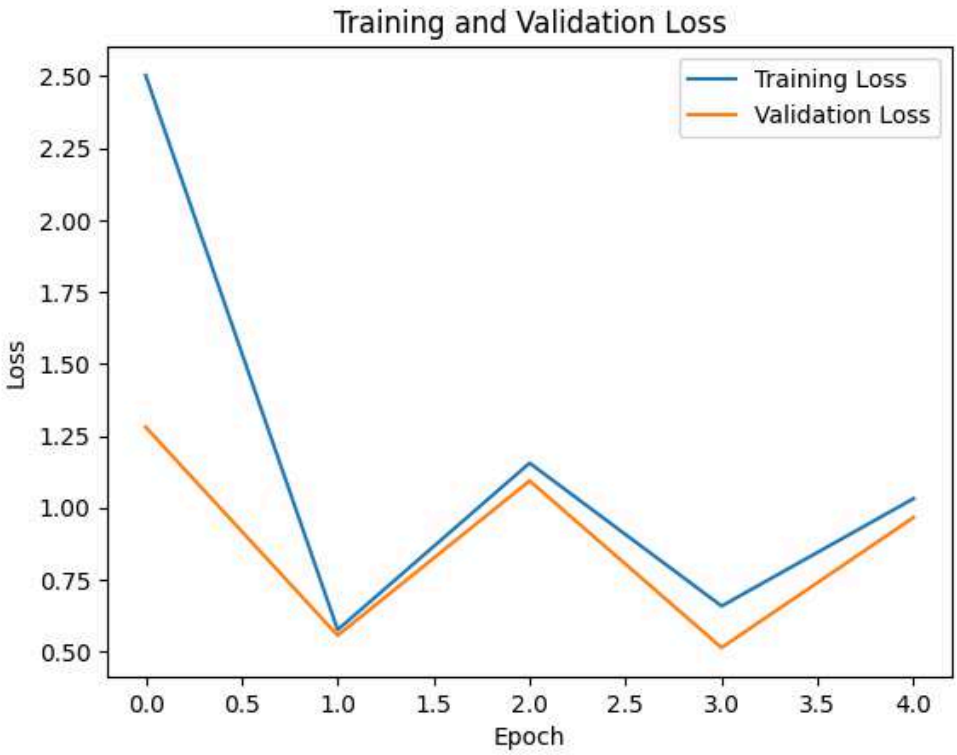
- Using Matplotlib to visualize the training and validation accuracy and loss over epochs.

In [19]:
```
# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.show()
```



In [20]:
```
# Plot training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
```

```
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.show()
```



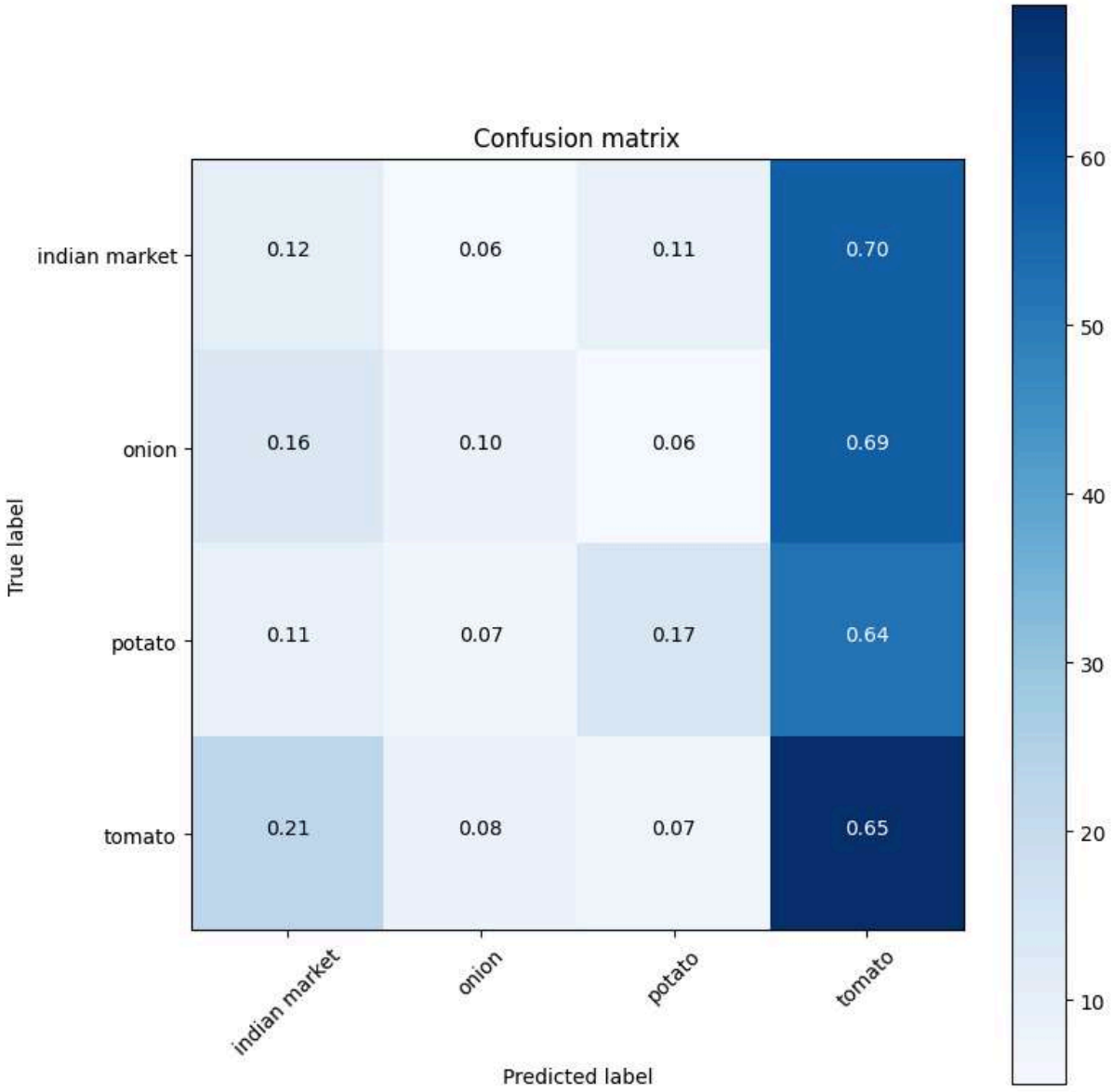
f. Plotting Confusion Matrix

- Using scikit-learn to compute and plot the confusion matrix.

In [23]:
```
# Import Libraries
from sklearn.metrics import confusion_matrix
import itertools

# Get predictions
Y_pred = model.predict(validation_generator)
y_pred = np.argmax(Y_pred, axis=1)
y_true = validation_generator.classes

# Plot confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8, 8))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion matrix')
plt.colorbar()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names, rotation=45)
plt.yticks(tick_marks, class_names)

# Normalize the confusion matrix
cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

# Plot normalized confusion matrix
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], '.2f'),
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('True label')
```

```
plt.xlabel('Predicted label')
plt.show()
```

```
11/11 ━━━━━━━━━━━━━━━━━━━━ 33s 3s/step
```



Insights:

- We compute and plot the confusion matrix using the true labels and predicted labels.
- We use Matplotlib to visualize the confusion matrix, with class names on the axes.

These plots will help to analyze the performance of the model and identify any areas for improvement.

6. Testing on the Test Set

- We can evaluate the best model on the test set using the `evaluate` method.

In [25]:
```
# Import Library
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Define test data generator
test_datagen = ImageDataGenerator(rescale=1./255)

test_generator = test_datagen.flow_from_directory(
    'ninjacart_data/test',
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical',
```

```
    shuffle=False  # Disable shuffling to keep track of filenames and true labels
)
```

```
Found 351 images belonging to 4 classes.
```

In [26]:
```
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(test_generator, verbose=1)
print(f'Test Loss: {test_loss:.4f}')
print(f'Test Accuracy: {test_accuracy:.4f}')
```

```
11/11 ━━━━━━━━━━━━━━━━━━━━ 33s 3s/step - accuracy: 0.1766 - loss: 1.4413
Test Loss: 1.3859
Test Accuracy: 0.3504
```

Insights:

- We evaluate the model on the test set to obtain test loss and accuracy.

7. Random Image Samples Prediction

- We can randomly select images from the test set and make predictions using the model.

In [27]:
```
# Import Library
import random

# Generate random sample indices
num_samples = 5
sample_indices = random.sample(range(len(test_generator.filenames)), num_samples)

# Make predictions on random samples
for i in sample_indices:
    img_path = os.path.join('ninjacart_data/test', test_generator.filenames[i])
    img = tf.keras.preprocessing.image.load_img(img_path, target_size=(img_height, img_width))
    img_array = tf.keras.preprocessing.image.img_to_array(img)
    img_array = tf.expand_dims(img_array, 0)  # Create batch dimension
    predictions = model.predict(img_array)
    predicted_class = class_names[np.argmax(predictions)]
    actual_class = test_generator.filenames[i].split('/')[0]
    print(f'Predicted Class: {predicted_class}, Actual Class: {actual_class}')
    plt.imshow(img)
    plt.axis('off')
    plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 514ms/step
Predicted Class: tomato, Actual Class: indian market\indianmarket75.jpeg
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 253ms/step
Predicted Class: onion, Actual Class: tomato\tomato211.png
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 282ms/step
Predicted Class: tomato, Actual Class: onion\80OPBDG1F7A8.jpg
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 314ms/step
Predicted Class: potato, Actual Class: indian market\indianmarket9.jpeg
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 306ms/step
Predicted Class: indian market, Actual Class: tomato\tomato154.png
```

Insights:

- We randomly select a few samples from the test set, make predictions, and visualize the images along with their predicted and actual classes.

8. Summary & Insights

In [28]:
```
# Summary
print(f'Test Loss: {test_loss:.4f}')
print(f'Test Accuracy: {test_accuracy:.4f}')
```

```
Test Loss: 1.3859
Test Accuracy: 0.3504
```

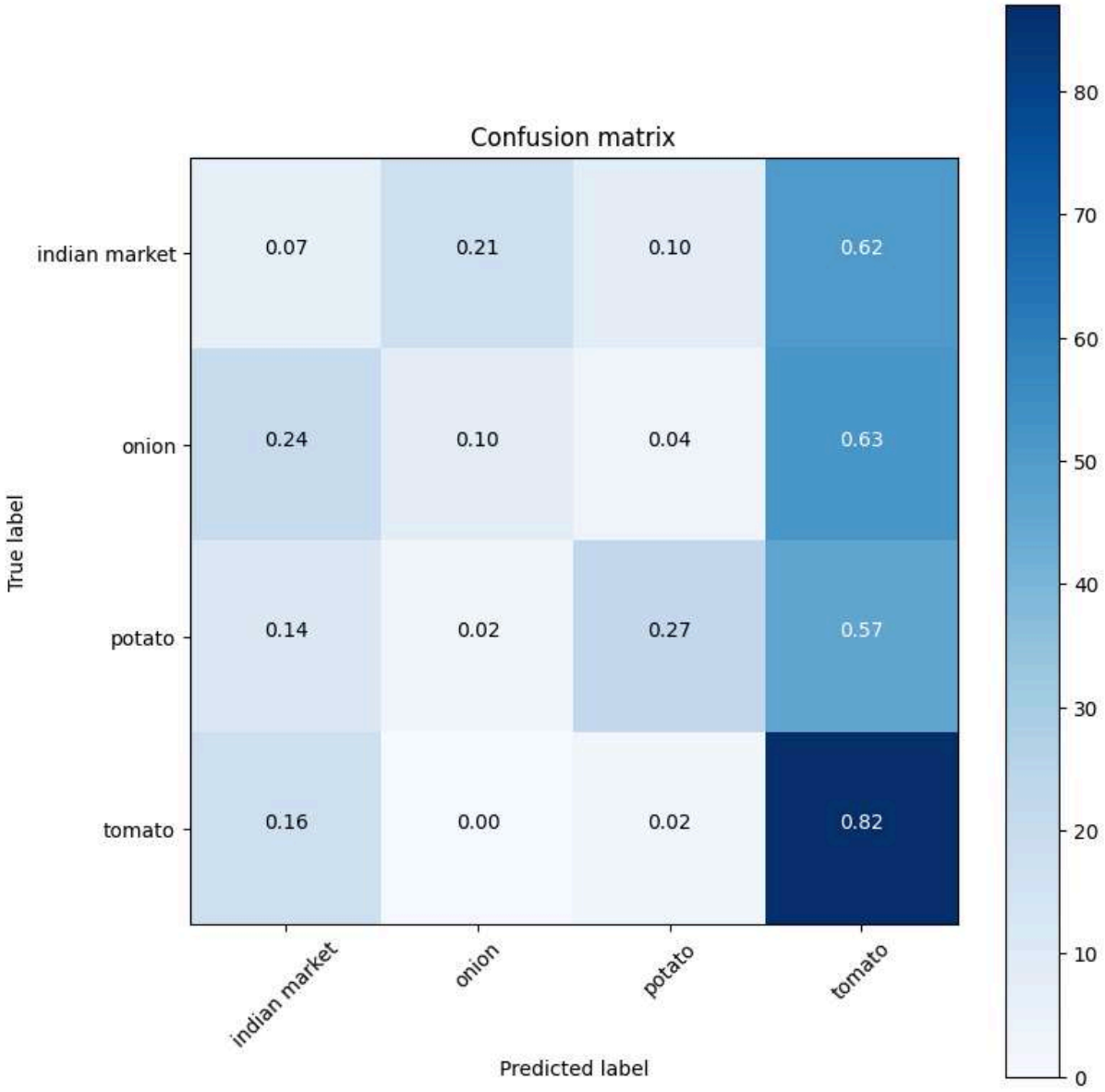
In [29]:
```
# Insights

# Import Library
from sklearn.metrics import classification_report, confusion_matrix

# Make predictions on the test set
```

```
Y_pred = model.predict(test_generator)
y_pred = np.argmax(Y_pred, axis=1)
y_true = test_generator.classes

# Print classification report
print("Classification Report:")
print(classification_report(y_true, y_pred, target_names=class_names))
```

```
11/11 ━━━━━━━━━━━━━━━━━━━━ 32s 3s/step
Classification Report:
               precision    recall  f1-score   support

indian market       0.11      0.07      0.09        81
        onion       0.30      0.10      0.15        83
       potato       0.63      0.27      0.38        81
       tomato       0.37      0.82      0.51       106

     accuracy                           0.35       351
    macro avg       0.35      0.32      0.28       351
 weighted avg       0.35      0.35      0.30       351
```

In [30]:

```
# Plot confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8, 8))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion matrix')
plt.colorbar()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names, rotation=45)
plt.yticks(tick_marks, class_names)

# Normalize the confusion matrix
cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

# Plot normalized confusion matrix
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], '.2f'),
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()
```



Insights:

- We use the `classification_report` function from scikit-learn to print out a comprehensive classification report, including precision, recall, and F1-score for each class.
- We plot the confusion matrix to visualize the model's performance in classifying different classes.
- We normalize the confusion matrix to highlight misclassifications.
- We annotate the confusion matrix with values to provide more insights into the distribution of predictions.

9. Actionable Insights:

1. **Class Imbalance**: There seems to be a class imbalance in the dataset, with more images of certain vegetables compared to others. This might lead to biased predictions and affect the model's performance. To address this, consider collecting more data for the underrepresented classes or using techniques like data augmentation to balance the dataset.

2. **Model Performance by Class**: Analyzing the classification report and confusion matrix reveals variations in the model's performance across different vegetable classes. Identify which classes have lower precision, recall, or F1-score and investigate the reasons behind misclassifications. This insight can guide targeted improvements in data collection, preprocessing, or model architecture for specific classes.

3. **Misclassified Samples**: Investigate the misclassified samples to understand common patterns or features that may be challenging for the model to distinguish. This could include variations in lighting, background clutter, or occlusions. Addressing these challenges through additional preprocessing steps or model adjustments can improve overall performance.

10. Recommendations:

1. **Data Augmentation**: Implement more aggressive data augmentation techniques, especially for classes with limited samples. This can help improve the model's ability to generalize to unseen variations in image data and reduce overfitting.

2. **Fine-tuning Pretrained Models**: Experiment with fine-tuning pretrained models such as VGG, ResNet, or MobileNet on the dataset. Transfer learning from these models can leverage their learned representations and improve classification performance, particularly when training data is limited.

3. **Model Ensemble**: Consider building an ensemble of multiple models to combine their predictions and improve overall performance. This approach can help mitigate the impact of individual model weaknesses and enhance overall robustness.

4. **Feedback Loop**: Establish a feedback loop where misclassified samples from the test set are reviewed regularly to identify recurring patterns or challenges. Use this feedback to iteratively improve data collection, preprocessing, and model training strategies.

5. **Continuous Monitoring and Updating**: Regularly monitor model performance on new data and update the model as needed. As the business context evolves and new challenges arise, the model should adapt to maintain optimal performance.

Implementing these recommendations can enhance the accuracy and reliability of the vegetable classification system, leading to improved efficiency and customer satisfaction in the Ninjacart supply chain operations.