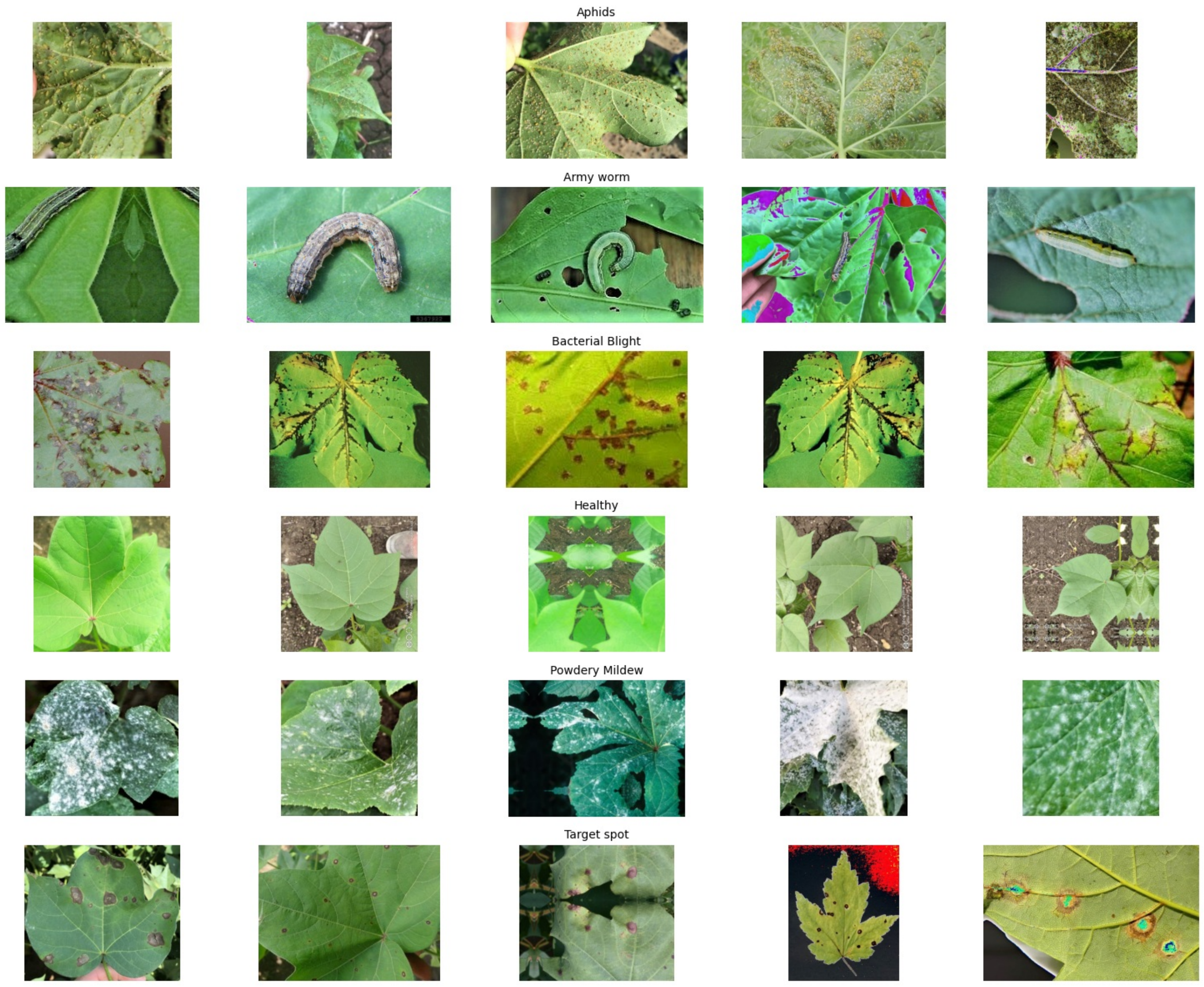
In [1]:
```python
import kagglehub

# Download latest version
path = kagglehub.dataset_download("sohansakib75/cotton-kaggle")

print("Path to dataset files:", path)
```
```
Path to dataset files: /kaggle/input/cotton-kaggle
```

In [2]:
```python
import os
import kagglehub

# Download dataset
path = kagglehub.dataset_download("sohansakib75/cotton-kaggle")

print("Dataset root path:", path)

# List folders and files inside
print("Contents inside dataset folder:")
for item in os.listdir(path):
    item_path = os.path.join(path, item)
    if os.path.isdir(item_path):
        print(f" {item}/")
    else:
        print(f" {item}")
```
```
Dataset root path: /kaggle/input/cotton-kaggle
Contents inside dataset folder:
 Dataset/
```

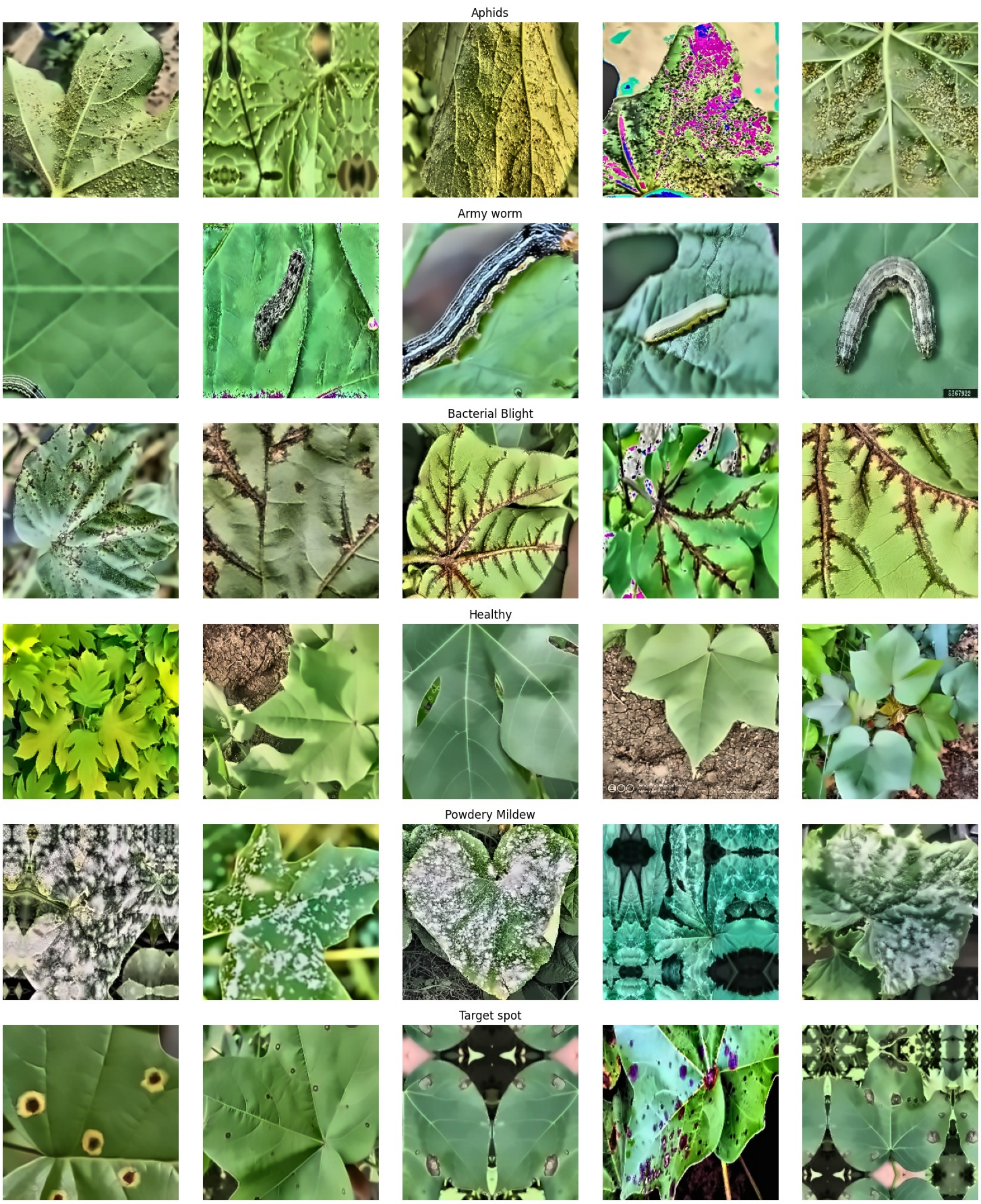
In [3]:
```python
import os
import glob

dataset_path = os.path.join(path, "Dataset")

print("Path to Dataset folder:", dataset_path)

# Loop through each subfolder (classes)
for subdir in sorted(os.listdir(dataset_path)):
    subpath = os.path.join(dataset_path, subdir)
    if os.path.isdir(subpath):
        # Count images by common formats
        image_files = glob.glob(os.path.join(subpath, "*.jpg")) + \
                      glob.glob(os.path.join(subpath, "*.jpeg")) + \
                      glob.glob(os.path.join(subpath, "*.png"))
        print(f"{subdir}: {len(image_files)} images")
```
```
Path to Dataset folder: /kaggle/input/cotton-kaggle/Dataset
Aphids: 400 images
Army worm: 400 images
Bacterial Blight: 400 images
Healthy: 400 images
Powdery Mildew: 400 images
Target spot: 400 images
```

In [4]:
```python
import os
import glob
import matplotlib.pyplot as plt
import random

dataset_path = "/kaggle/input/cotton-kaggle/Dataset"

# Classes
classes = sorted(os.listdir(dataset_path))

# Plot 5 images per class
fig, axes = plt.subplots(len(classes), 5, figsize=(15, 12))

for i, cls in enumerate(classes):
    cls_path = os.path.join(dataset_path, cls)
    image_files = glob.glob(os.path.join(cls_path, "*.jpg")) + \
                  glob.glob(os.path.join(cls_path, "*.jpeg")) + \
                  glob.glob(os.path.join(cls_path, "*.png"))

    # Randomly pick 5 images
    sample_files = random.sample(image_files, 5)

    for j, img_path in enumerate(sample_files):
        img = plt.imread(img_path)
        axes[i, j].imshow(img)
        axes[i, j].axis("off")
        if j == 2:  # center column
            axes[i, j].set_title(cls, fontsize=10)
```

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



In [5]:
```python
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import random

# Paths
input_dir = "/kaggle/input/cotton-kaggle/Dataset"
output_dir = "/kaggle/working/preprocessed_dataset"
os.makedirs(output_dir, exist_ok=True)

img_size = (224, 224)
num_samples = 5

classes = sorted(os.listdir(input_dir))
fig, axes = plt.subplots(len(classes), num_samples, figsize=(15, 3*len(classes)))

# CLAHE object for adaptive histogram equalization
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))

for i, cls in enumerate(classes):
    cls_path = os.path.join(input_dir, cls)
    if os.path.isdir(cls_path):
        save_cls_path = os.path.join(output_dir, cls)
        os.makedirs(save_cls_path, exist_ok=True)

        files = [f for f in os.listdir(cls_path) if f.lower().endswith((".jpg", ".jpeg", ".png"))]
        sample_files = random.sample(files, min(num_samples, len(files)))

        for j, file in enumerate(files):
            img_path = os.path.join(cls_path, file)
            img = cv2.imread(img_path)

            # 1⃣  Resize
            img_resized = cv2.resize(img, img_size)
            # 2⃣ CLAHE on Y channel (enhance contrast)
            yuv = cv2.cvtColor(img_resized, cv2.COLOR_BGR2YUV)
            yuv[:,:,0] = clahe.apply(yuv[:,:,0])
            img_clahe = cv2.cvtColor(yuv, cv2.COLOR_YUV2BGR)

            # 3⃣ Non-Local Means Denoising (remove camera noise)
            img_denoised = cv2.fastNlMeansDenoisingColored(img_clahe, None, h=10, hColor=10, templateWindowSize=

            # 4⃣ Normalize to [0,1]
            img_final = img_denoised.astype(np.float32) / 255.0

            # Save preprocessed image
            save_img = (img_final * 255).astype(np.uint8)
            cv2.imwrite(os.path.join(save_cls_path, file), save_img)

            # Plot sample images
            if file in sample_files:
                axes[i, sample_files.index(file)].imshow(cv2.cvtColor(save_img, cv2.COLOR_BGR2RGB))
                axes[i, sample_files.index(file)].axis("off")
                if sample_files.index(file) == num_samples // 2:
                    axes[i, sample_files.index(file)].set_title(cls, fontsize=12)

plt.tight_layout()
plt.show()
print("\n Preprocessed images saved in:", output_dir)
```



```
Preprocessed images saved in: /kaggle/working/preprocessed_dataset
```

In [6]:

```python
import os
import shutil
import random

# Preprocessed dataset path
preprocessed_dir = "/kaggle/working/preprocessed_dataset"

# Split paths
split_base = "/kaggle/working/cotton_split"
train_dir = os.path.join(split_base, "train")
val_dir   = os.path.join(split_base, "val")
test_dir  = os.path.join(split_base, "test")

# Create split folders
for d in [train_dir, val_dir, test_dir]:
    os.makedirs(d, exist_ok=True)

# Split ratios
```

```
train_ratio = 0.8
val_ratio = 0.1
test_ratio = 0.1

classes = sorted(os.listdir(preprocessed_dir))

for cls in classes:
    cls_path = os.path.join(preprocessed_dir, cls)
    files = [f for f in os.listdir(cls_path) if f.lower().endswith((".jpg", ".jpeg", ".png"))]
    random.shuffle(files)

    n_total = len(files)
    n_train = int(train_ratio * n_total)
    n_val = int(val_ratio * n_total)
    n_test = n_total - n_train - n_val

    splits = {
        train_dir: files[:n_train],
        val_dir: files[n_train:n_train+n_val],
        test_dir: files[n_train+n_val:]
    }

    for split_folder, split_files in splits.items():
        cls_split_path = os.path.join(split_folder, cls)
        os.makedirs(cls_split_path, exist_ok=True)
        for f in split_files:
            shutil.copy(os.path.join(cls_path, f), os.path.join(cls_split_path, f))

print(" Dataset split into 80-10-10 and saved in:", split_base)
```

```
Dataset split into 80-10-10 and saved in: /kaggle/working/cotton_split
```

In [7]:
```
import os

split_base = "/kaggle/working/cotton_split"
splits = ["train", "val", "test"]

for split in splits:
    split_path = os.path.join(split_base, split)
    print(f"\n {split.capitalize()} Split:")
    for cls in sorted(os.listdir(split_path)):
        cls_path = os.path.join(split_path, cls)
        num_images = len([f for f in os.listdir(cls_path) if f.lower().endswith((".jpg", ".jpeg", ".png"))])
        print(f"{cls}: {num_images} images")
```

```
 Train Split:
Aphids: 320 images
Army worm: 320 images
Bacterial Blight: 320 images
Healthy: 320 images
Powdery Mildew: 320 images
Target spot: 320 images

 Val Split:
Aphids: 40 images
Army worm: 40 images
Bacterial Blight: 40 images
Healthy: 40 images
Powdery Mildew: 40 images
Target spot: 40 images

 Test Split:
Aphids: 40 images
Army worm: 40 images
Bacterial Blight: 40 images
Healthy: 40 images
Powdery Mildew: 40 images
Target spot: 40 images
```

In [8]:
```
import os
import cv2
import numpy as np
import random

train_dir = "/kaggle/working/cotton_split/train"
aug_train_dir = "/kaggle/working/cotton_train_aug"
os.makedirs(aug_train_dir, exist_ok=True)

# Augmentation functions
def random_flip(img):
    flip_code = random.choice([-1, 0, 1])
    return cv2.flip(img, flip_code)

def random_rotate(img):
    angle = random.uniform(-25, 25)
    h, w = img.shape[:2]
    M = cv2.getRotationMatrix2D((w//2, h//2), angle, 1)
    return cv2.warpAffine(img, M, (w, h), borderMode=cv2.BORDER_REFLECT)

def random_zoom(img):
    zoom_factor = random.uniform(0.8, 1.2)
    h, w = img.shape[:2]
    new_h, new_w = int(h*zoom_factor), int(w*zoom_factor)
    img_resized = cv2.resize(img, (new_w, new_h))
    if zoom_factor < 1:
        pad_h = (h - new_h) // 2
        pad_w = (w - new_w) // 2
        img_padded = cv2.copyMakeBorder(img_resized, pad_h, h-new_h-pad_h,
                                        pad_w, w-new_w-pad_w, cv2.BORDER_REFLECT)
        return img_padded
    else:
        start_h = (new_h - h)//2
        start_w = (new_w - w)//2
        return img_resized[start_h:start_h+h, start_w:start_w+w]

def random_brightness(img):
    factor = random.uniform(0.7, 1.3)
    img = img.astype(np.float32) * factor
    img = np.clip(img, 0, 255).astype(np.uint8)
    return img

augmentations = [random_flip, random_rotate, random_zoom, random_brightness]

# Apply augmentations
classes = sorted(os.listdir(train_dir))
for cls in classes:
    cls_path = os.path.join(train_dir, cls)
    save_cls_path = os.path.join(aug_train_dir, cls)
    os.makedirs(save_cls_path, exist_ok=True)

    for file in os.listdir(cls_path):
        if not file.lower().endswith((".jpg", ".jpeg", ".png")):
            continue
        img_path = os.path.join(cls_path, file)
        img = cv2.imread(img_path)

        # Save original
        cv2.imwrite(os.path.join(save_cls_path, file), img)

        # 3 random augmentations
        for k in range(3):
            aug_img = img.copy()
            aug_funcs = random.sample(augmentations, 2)
            for func in aug_funcs:
                aug_img = func(aug_img)
            filename, ext = os.path.splitext(file)
            aug_name = f"{filename}_aug{k+1}{ext}"
            cv2.imwrite(os.path.join(save_cls_path, aug_name), aug_img)

# Print image count per class
print("\n Image count per class after augmentation:")
for cls in classes:
    cls_path = os.path.join(aug_train_dir, cls)
    count = len([f for f in os.listdir(cls_path) if f.lower().endswith((".jpg", ".jpeg", ".png"))])
    print(f"{cls}: {count} images")

print(f"\n All train images and augmented images saved in: {aug_train_dir}")
```

```
 Image count per class after augmentation:
Aphids: 1280 images
Army worm: 1280 images
Bacterial Blight: 1280 images
Healthy: 1280 images
Powdery Mildew: 1280 images
Target spot: 1280 images

 All train images and augmented images saved in: /kaggle/working/cotton_train_aug
```

In [18]:
```
import torch, gc, time
from torch import nn, optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
import numpy as np
from sklearn.model_selection import KFold

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

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

```python
# Dataset
# ----------------------
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])
full_dataset = datasets.ImageFolder('/kaggle/working/cotton_train_aug', transform=transform)
num_classes = len(full_dataset.classes)

# ----------------------
# Multi-Head Self-Attention
# ----------------------
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, in_channels, num_heads=4, dropout=0.3):
        super().__init__()
        assert in_channels % num_heads == 0
        self.mha = nn.MultiheadAttention(embed_dim=in_channels, num_heads=num_heads, dropout=dropout, batch_firs
        self.dropout = nn.Dropout(dropout)
        self.norm = nn.LayerNorm(in_channels)

    def forward(self, x):
        b, c, h, w = x.size()
        x_flat = x.view(b, c, h*w).permute(0,2,1)
        attn_out, _ = self.mha(x_flat, x_flat, x_flat)
        attn_out = self.dropout(attn_out)
        attn_out = self.norm(attn_out + x_flat)
        return attn_out.permute(0,2,1).view(b, c, h, w)

# ----------------------
# SqueezeNet + MHSA
# ----------------------
class SqueezeNet_MHSA(nn.Module):
    def __init__(self, num_classes, num_heads=4, dropout=0.3):
        super().__init__()
        self.backbone = models.squeezenet1_1(weights=models.SqueezeNet1_1_Weights.DEFAULT)
        for p in self.backbone.parameters():
            p.requires_grad = False
        self.features = self.backbone.features
        last_channels = 512
        self.mhsa = MultiHeadSelfAttention(last_channels, num_heads=num_heads, dropout=dropout)
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Dropout(dropout),
            nn.Linear(last_channels, num_classes)
        )

    def forward(self, x):
        f = self.features(x)
        f = self.mhsa(f)
        return self.classifier(f)

# ----------------------
# 5-Fold Cross-Validation
# ----------------------
kf = KFold(n_splits=5, shuffle=True, random_state=42)
fold_accuracies = []

for fold, (train_idx, val_idx) in enumerate(kf.split(full_dataset)):
    print(f"\n--- Fold {fold+1} ---")
    start_fold = time.time()

    train_subset = Subset(full_dataset, train_idx)
    val_subset = Subset(full_dataset, val_idx)

    train_loader = DataLoader(train_subset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=32, shuffle=False)

    model = SqueezeNet_MHSA(num_classes=num_classes).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5)

    best_val_acc = 0
    patience_counter = 0

    for epoch in range(50):
        model.train()
        running_loss = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        # Validation
        model.eval()
        correct, total, val_loss = 0, 0, 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, preds = torch.max(outputs, 1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
        val_acc = correct / total
        val_loss /= len(val_loader)
        scheduler.step(val_loss)

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= 5:
                break

    fold_time = time.time() - start_fold
    print(f"Fold {fold+1} Accuracy: {best_val_acc:.4f} | Fold Time: {fold_time:.2f}s")
    fold_accuracies.append(best_val_acc)

    # ----------------------
    # Clear GPU & RAM
    # ----------------------
    del model, optimizer, criterion, train_loader, val_loader
    torch.cuda.empty_cache()
    gc.collect()

fold_accuracies = np.array(fold_accuracies)
print(f"\n 5-Fold CV Accuracy: {fold_accuracies.mean():.4f} ± {fold_accuracies.std():.4f}")
```

```
--- Fold 1 ---
Fold 1 Accuracy: 0.9798 | Fold Time: 284.79s

--- Fold 2 ---
Fold 2 Accuracy: 0.9798 | Fold Time: 280.33s

--- Fold 3 ---
Fold 3 Accuracy: 0.9714 | Fold Time: 207.99s

--- Fold 4 ---
Fold 4 Accuracy: 0.9753 | Fold Time: 281.59s

--- Fold 5 ---
Fold 5 Accuracy: 0.9701 | Fold Time: 191.48s

 5-Fold CV Accuracy: 0.9753 ± 0.0041
```

In [19]:
```
import torch, gc, time, psutil, numpy as np
from torch import nn, optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
from sklearn.model_selection import KFold

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# ----------------------
# Dataset
# ----------------------
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])
full_dataset = datasets.ImageFolder('/kaggle/working/cotton_train_aug', transform=transform)
num_classes = len(full_dataset.classes)

# ----------------------
# Multi-Head Self-Attention
# ----------------------
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, in_channels, num_heads=4, dropout=0.3):
        super().__init__()
        assert in_channels % num_heads == 0
        self.mha = nn.MultiheadAttention(embed_dim=in_channels, num_heads=num_heads, dropout=dropout, batch_firs
        self.dropout = nn.Dropout(dropout)
        self.norm = nn.LayerNorm(in_channels)

    def forward(self, x):
        b, c, h, w = x.size()
        x_flat = x.view(b, c, h*w).permute(0,2,1)
        attn_out, _ = self.mha(x_flat, x_flat, x_flat)
        attn_out = self.dropout(attn_out)
        attn_out = self.norm(attn_out + x_flat)
        return attn_out.permute(0,2,1).view(b, c, h, w)

# ----------------------
# VGG19 + MHSA
# ----------------------
class VGG19_MHSA(nn.Module):
    def __init__(self, num_classes, num_heads=4, dropout=0.3, freeze_backbone=True):
        super().__init__()
        self.backbone = models.vgg19(weights=models.VGG19_Weights.DEFAULT).features
        if freeze_backbone:
            for p in self.backbone.parameters():
                p.requires_grad = False
        last_channels = 512
        self.mhsa = MultiHeadSelfAttention(last_channels, num_heads=num_heads, dropout=dropout)
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Dropout(dropout),
            nn.Linear(last_channels, num_classes)
        )

    def forward(self, x):
        f = self.backbone(x)
        f = self.mhsa(f)
        return self.classifier(f)

# ----------------------
# 5-Fold Cross-Validation
# ----------------------
kf = KFold(n_splits=5, shuffle=True, random_state=42)
fold_accuracies = []

for fold, (train_idx, val_idx) in enumerate(kf.split(full_dataset)):
    print(f"\n--- Fold {fold+1} ---")

    train_subset = Subset(full_dataset, train_idx)
    val_subset = Subset(full_dataset, val_idx)

    train_loader = DataLoader(train_subset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=32, shuffle=False)

    model = VGG19_MHSA(num_classes=num_classes, num_heads=4, dropout=0.3).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5)

    best_val_acc = 0
    num_epochs = 50
    patience_counter = 0
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        # Validation
        model.eval()
        correct, total = 0, 0
        val_loss = 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, preds = torch.max(outputs, 1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
        val_acc = correct / total
        scheduler.step(val_loss / len(val_loader))

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= 5:
                break

    print(f"Fold {fold+1} Accuracy: {best_val_acc:.4f}")
    fold_accuracies.append(best_val_acc)

    # ----------------------
    # Clear RAM & GPU
    # ----------------------
    del model, optimizer, criterion, train_loader, val_loader
    torch.cuda.empty_cache()
    gc.collect()

fold_accuracies = np.array(fold_accuracies)
print(f"\n 5-Fold CV Accuracy: {fold_accuracies.mean():.4f} ± {fold_accuracies.std():.4f}")
```

```
--- Fold 1 ---
Fold 1 Accuracy: 0.9740

--- Fold 2 ---
Fold 2 Accuracy: 0.9837

--- Fold 3 ---
Fold 3 Accuracy: 0.9805

--- Fold 4 ---
Fold 4 Accuracy: 0.9798

--- Fold 5 ---
Fold 5 Accuracy: 0.9772

 5-Fold CV Accuracy: 0.9790 ± 0.0033
```

In [20]:
```
import os, time, psutil, torch, gc
from torch import nn, optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
import numpy as np
from sklearn.model_selection import KFold

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# ----------------------
# Dataset
# ----------------------
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])
full_dataset = datasets.ImageFolder('/kaggle/working/cotton_train_aug', transform=transform)
num_classes = len(full_dataset.classes)

# ----------------------
# Multi-Head Self-Attention
# ----------------------
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, in_channels, num_heads=4, dropout=0.3):
        super().__init__()
        assert in_channels % num_heads == 0
        self.mha = nn.MultiheadAttention(embed_dim=in_channels, num_heads=num_heads, dropout=dropout, batch_firs
        self.dropout = nn.Dropout(dropout)
        self.norm = nn.LayerNorm(in_channels)

    def forward(self, x):
        b, c, h, w = x.size()
        x_flat = x.view(b, c, h*w).permute(0,2,1)
        attn_out, _ = self.mha(x_flat, x_flat, x_flat)
        attn_out = self.dropout(attn_out)
        attn_out = self.norm(attn_out + x_flat)
        return attn_out.permute(0,2,1).view(b, c, h, w)
```

```python
# ----------------------
# ResNet101 + MHSA
# ----------------------
class ResNet101_MHSA(nn.Module):
    def __init__(self, num_classes, num_heads=4, dropout=0.3, freeze_backbone=True):
        super().__init__()
        self.backbone = models.resnet101(weights=models.ResNet101_Weights.DEFAULT)
        if freeze_backbone:
            for p in self.backbone.parameters():
                p.requires_grad = False
        self.backbone.fc = nn.Identity()
        self.mhsa = MultiHeadSelfAttention(in_channels=2048, num_heads=num_heads, dropout=dropout)
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Dropout(dropout),
            nn.Linear(2048, num_classes)
        )

    def forward(self, x):
        features = self.backbone(x).unsqueeze(-1).unsqueeze(-1)
        attn_features = self.mhsa(features)
        out = self.classifier(attn_features)
        return out

# ----------------------
# 5-Fold Cross-Validation
# ----------------------
kf = KFold(n_splits=5, shuffle=True, random_state=42)
fold_accuracies = []

for fold, (train_idx, val_idx) in enumerate(kf.split(full_dataset)):
    print(f"\n--- Fold {fold+1} ---")

    train_subset = Subset(full_dataset, train_idx)
    val_subset = Subset(full_dataset, val_idx)

    train_loader = DataLoader(train_subset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=32, shuffle=False)

    model = ResNet101_MHSA(num_classes=num_classes).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5)

    best_val_acc = 0
    patience_counter = 0

    for epoch in range(50):
        model.train()
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        # Validation
        model.eval()
        correct, total = 0, 0
        val_loss = 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, preds = torch.max(outputs, 1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)

        val_acc = correct / total
        val_loss /= len(val_loader)
        scheduler.step(val_loss)

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= 5:
                break
    print(f"Fold {fold+1} Accuracy: {best_val_acc:.4f}")
    fold_accuracies.append(best_val_acc)

    # Clear GPU & RAM after each fold
    del model, optimizer, criterion, train_loader, val_loader
    torch.cuda.empty_cache()
    gc.collect()

fold_accuracies = np.array(fold_accuracies)
print(f"\n 5-Fold CV Accuracy: {fold_accuracies.mean():.4f} ± {fold_accuracies.std():.4f}")
```

```
--- Fold 1 ---
Fold 1 Accuracy: 0.9785

--- Fold 2 ---
Fold 2 Accuracy: 0.9876

--- Fold 3 ---
Fold 3 Accuracy: 0.9798

--- Fold 4 ---
Fold 4 Accuracy: 0.9818

--- Fold 5 ---
Fold 5 Accuracy: 0.9883

 5-Fold CV Accuracy: 0.9832 ± 0.0040
```

In [21]:

```
import os, time, psutil, torch, gc
from torch import nn, optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
import numpy as np
from sklearn.model_selection import KFold

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# ----------------------
# Dataset
# ----------------------
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])
full_dataset = datasets.ImageFolder('/kaggle/working/cotton_train_aug', transform=transform)
num_classes = len(full_dataset.classes)

# ----------------------
# Multi-Head Self-Attention
# ----------------------
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, in_channels, num_heads=4, dropout=0.3):
        super().__init__()
        assert in_channels % num_heads == 0
        self.mha = nn.MultiheadAttention(embed_dim=in_channels, num_heads=num_heads, dropout=dropout, batch_firs
        self.dropout = nn.Dropout(dropout)
        self.norm = nn.LayerNorm(in_channels)

    def forward(self, x):
        b, c, h, w = x.size()
        x_flat = x.view(b, c, h*w).permute(0,2,1)
        attn_out, _ = self.mha(x_flat, x_flat, x_flat)
        attn_out = self.dropout(attn_out)
        attn_out = self.norm(attn_out + x_flat)
        return attn_out.permute(0,2,1).view(b, c, h, w)

# ----------------------
# MobileNetV2 + MHSA
# ----------------------
class MobileNetV2_MHSA(nn.Module):
    def __init__(self, num_classes, num_heads=4, dropout=0.3, freeze_backbone=True):
        super().__init__()
        self.backbone = models.mobilenet_v2(weights=models.MobileNet_V2_Weights.DEFAULT)
        if freeze_backbone:
            for p in self.backbone.parameters():
                p.requires_grad = False
        self.features = self.backbone.features
        last_channels = 1280
        self.mhsa = MultiHeadSelfAttention(last_channels, num_heads=num_heads, dropout=dropout)
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
```

```python
            nn.Dropout(dropout),
            nn.Linear(last_channels, num_classes)
        )

    def forward(self, x):
        features = self.features(x)
        attn_features = self.mhsa(features)
        out = self.classifier(attn_features)
        return out

# ----------------------
# 5-Fold Cross-Validation
# ----------------------
kf = KFold(n_splits=5, shuffle=True, random_state=42)
fold_accuracies = []

for fold, (train_idx, val_idx) in enumerate(kf.split(full_dataset)):
    print(f"\n--- Fold {fold+1} ---")

    train_subset = Subset(full_dataset, train_idx)
    val_subset = Subset(full_dataset, val_idx)

    train_loader = DataLoader(train_subset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=32, shuffle=False)

    model = MobileNetV2_MHSA(num_classes=num_classes).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5)

    best_val_loss = float('inf')
    patience_counter = 0

    for epoch in range(50):
        # ----------------------
        # Training
        # ----------------------
        model.train()
        running_loss = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        # ----------------------
        # Validation
        # ----------------------
        model.eval()
        val_loss, correct, total = 0, 0, 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, preds = torch.max(outputs, 1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)

        val_loss /= len(val_loader)
        val_acc = correct / total
        scheduler.step(val_loss)

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= 5:
                break  # Early stopping

    print(f"Fold {fold+1} Validation Accuracy: {val_acc:.4f}")
    fold_accuracies.append(val_acc)

    # ----------------------
    # Clear GPU & RAM
    # ----------------------
    del model, optimizer, criterion, train_loader, val_loader
    torch.cuda.empty_cache()
    gc.collect()

fold_accuracies = np.array(fold_accuracies)
print(f"\n 5-Fold CV Accuracy: {fold_accuracies.mean():.4f} ± {fold_accuracies.std():.4f}")
```

```
--- Fold 1 ---
Fold 1 Validation Accuracy: 0.9824

--- Fold 2 ---
Fold 2 Validation Accuracy: 0.9876

--- Fold 3 ---
Fold 3 Validation Accuracy: 0.9857

--- Fold 4 ---
Fold 4 Validation Accuracy: 0.9863

--- Fold 5 ---
Fold 5 Validation Accuracy: 0.9850

 5-Fold CV Accuracy: 0.9854 ± 0.0017
```

In [22]:
```
import os, time, psutil, torch, gc
from torch import nn, optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
import numpy as np
from sklearn.model_selection import KFold

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# ----------------------
# Dataset
# ----------------------
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])
full_dataset = datasets.ImageFolder('/kaggle/working/cotton_train_aug', transform=transform)
num_classes = len(full_dataset.classes)

# ----------------------
# Multi-Head Self-Attention
# ----------------------
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, in_channels, num_heads=4, dropout=0.3):
        super().__init__()
        assert in_channels % num_heads == 0
        self.mha = nn.MultiheadAttention(embed_dim=in_channels, num_heads=num_heads, dropout=dropout, batch_firs
        self.dropout = nn.Dropout(dropout)
        self.norm = nn.LayerNorm(in_channels)

    def forward(self, x):
        b, c, h, w = x.size()
        x_flat = x.view(b, c, h*w).permute(0,2,1)
        attn_out, _ = self.mha(x_flat, x_flat, x_flat)
        attn_out = self.dropout(attn_out)
        attn_out = self.norm(attn_out + x_flat)
        return attn_out.permute(0,2,1).view(b, c, h, w)

# ----------------------
# EfficientNet-B0 + MHSA
# ----------------------
class EfficientNetB0_MHSA(nn.Module):
    def __init__(self, num_classes, num_heads=4, dropout=0.3, freeze_backbone=True):
        super().__init__()
        self.backbone = models.efficientnet_b0(weights=models.EfficientNet_B0_Weights.DEFAULT)
        if freeze_backbone:
            for p in self.backbone.parameters():
                p.requires_grad = False
        self.features = self.backbone.features
        last_channels = 1280
        self.mhsa = MultiHeadSelfAttention(last_channels, num_heads=num_heads, dropout=dropout)
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Dropout(dropout),
            nn.Linear(last_channels, num_classes)
        )

    def forward(self, x):
        features = self.features(x)
        attn_features = self.mhsa(features)
```

```python
        out = self.classifier(attn_features)
        return out

# ----------------------
# 5-Fold Cross-Validation
# ----------------------
kf = KFold(n_splits=5, shuffle=True, random_state=42)
fold_accuracies = []

for fold, (train_idx, val_idx) in enumerate(kf.split(full_dataset)):
    print(f"\n--- Fold {fold+1} ---")

    train_subset = Subset(full_dataset, train_idx)
    val_subset = Subset(full_dataset, val_idx)

    train_loader = DataLoader(train_subset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=32, shuffle=False)

    model = EfficientNetB0_MHSA(num_classes=num_classes).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5)

    best_val_loss = float('inf')
    patience_counter = 0

    for epoch in range(50):
        # ----------------------
        # Training
        # ----------------------
        model.train()
        running_loss = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        # ----------------------
        # Validation
        # ----------------------
        model.eval()
        val_loss, correct, total = 0, 0, 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, preds = torch.max(outputs, 1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)

        val_loss /= len(val_loader)
        val_acc = correct / total
        scheduler.step(val_loss)

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= 5:
                break  # Early stopping

    print(f"Fold {fold+1} Validation Accuracy: {val_acc:.4f}")
    fold_accuracies.append(val_acc)

    # ----------------------
    # Clear GPU & RAM
    # ----------------------
    del model, optimizer, criterion, train_loader, val_loader
    torch.cuda.empty_cache()
    gc.collect()

fold_accuracies = np.array(fold_accuracies)
print(f"\n 5-Fold CV Accuracy: {fold_accuracies.mean():.4f} ± {fold_accuracies.std():.4f}")
```

```
--- Fold 1 ---
Fold 1 Validation Accuracy: 0.9922

--- Fold 2 ---
Fold 2 Validation Accuracy: 0.9889

--- Fold 3 ---
Fold 3 Validation Accuracy: 0.9915

--- Fold 4 ---
Fold 4 Validation Accuracy: 0.9883

--- Fold 5 ---
Fold 5 Validation Accuracy: 0.9928

 5-Fold CV Accuracy: 0.9908 ± 0.0018
```

In [23]:

```python
# ----------------------
# IMPORTS
# ----------------------
import os, time, psutil
import torch
from torch import nn, optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
import numpy as np
from sklearn.model_selection import KFold

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# ----------------------
# DATASET
# ----------------------
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])
])

full_dataset = datasets.ImageFolder('/kaggle/working/cotton_train_aug', transform=transform)
num_classes = len(full_dataset.classes)

# ----------------------
# MULTI-HEAD SELF-ATTENTION
# ----------------------
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, in_channels, num_heads=4, dropout=0.3):
        super().__init__()
        self.mha = nn.MultiheadAttention(embed_dim=in_channels, num_heads=num_heads,
                                         dropout=dropout, batch_first=True)
        self.dropout = nn.Dropout(dropout)
        self.norm = nn.LayerNorm(in_channels)
    def forward(self, x):
        b, c, h, w = x.size()
        x_flat = x.view(b, c, h*w).permute(0,2,1)
        attn_out, _ = self.mha(x_flat, x_flat, x_flat)
        attn_out = self.dropout(attn_out)
        attn_out = self.norm(attn_out + x_flat)
        return attn_out.permute(0,2,1).view(b, c, h, w)

# ----------------------
# SHUFFLENETV2 + MHSA
# ----------------------
class ShuffleNetV2_MHSA(nn.Module):
    def __init__(self, num_classes, num_heads=4, dropout=0.3):
        super().__init__()
        self.backbone = models.shufflenet_v2_x1_0(weights=models.ShuffleNet_V2_X1_0_Weights.DEFAULT)
        for param in self.backbone.parameters():
            param.requires_grad = False
        self.backbone.fc = nn.Identity()
        self.mhsa = MultiHeadSelfAttention(in_channels=1024, num_heads=num_heads, dropout=dropout)
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Dropout(dropout), nn.Linear(1024, num_classes)
        )
    def forward(self, x):
        f = self.backbone(x)
        f = f.unsqueeze(-1).unsqueeze(-1)
        f = self.mhsa(f)
        return self.classifier(f)

# ----------------------
# 5-FOLD CROSS-VALIDATION
# ----------------------
```

```python
kf = KFold(n_splits=5, shuffle=True, random_state=42)
fold_accuracies = []

for fold, (train_idx, val_idx) in enumerate(kf.split(full_dataset)):
    print(f"\n--- Fold {fold+1} ---")
    train_subset = Subset(full_dataset, train_idx)
    val_subset   = Subset(full_dataset, val_idx)

    train_loader = DataLoader(train_subset, batch_size=32, shuffle=True)
    val_loader   = DataLoader(val_subset, batch_size=32, shuffle=False)

    model = ShuffleNetV2_MHSA(num_classes).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-4)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5)

    best_val_loss = float('inf')
    patience_counter = 0
    for epoch in range(50):
        model.train()
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            loss = criterion(model(inputs), labels)
            loss.backward()
            optimizer.step()

        # Validation
        model.eval()
        val_loss, correct, total = 0, 0, 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, preds = torch.max(outputs, 1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)

        val_loss /= len(val_loader)
        val_acc = correct / total
        scheduler.step(val_loss)

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict(), f"best_model_fold{fold+1}.pth")
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= 5:
                break

    print(f"Fold {fold+1} Accuracy: {val_acc:.4f}")
    fold_accuracies.append(val_acc)

fold_accuracies = np.array(fold_accuracies)
print(f"\n 5-Fold CV Accuracy: {fold_accuracies.mean():.4f} ± {fold_accuracies.std():.4f}")
```

```
--- Fold 1 ---
Fold 1 Accuracy: 0.9902

--- Fold 2 ---
Fold 2 Accuracy: 0.9824

--- Fold 3 ---
Fold 3 Accuracy: 0.9785

--- Fold 4 ---
Fold 4 Accuracy: 0.9850

--- Fold 5 ---
Fold 5 Accuracy: 0.9772

 5-Fold CV Accuracy: 0.9827 ± 0.0047
```