```python

import argparse
import os
from datetime import datetime
from pathlib import Path
from collections import Counter, defaultdict
import json
import pickle

import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
from torch import nn
from torchvision import transforms
from torchinfo import summary
from tqdm import tqdm
from scipy.stats import pearsonr, spearmanr
import torch.nn.functional as F
from torch.nn.functional import softmax
import numpy

# ============================================================
# Parsing the arguments---------------------------------------
# ============================================================
parser = argparse.ArgumentParser()
parser.add_argument("--test_run", type=int, required=True, help="Experiment run index")
parser.add_argument("--vit_ckpt_path", type=str, required=True, help="Path to saved ViT checkpoint (.pth)")
parser.add_argument(
    "--data_dir",
    type=str,
    required=True,
    help="Path to dataset root containing train/, val/, and test/ folders"
)
args = parser.parse_args()

test_run = args.test_run
vit_ckpt_path = args.vit_ckpt_path

# ============================================================
# Device configuration
# ============================================================
device = "cuda" if torch.cuda.is_available() else "cpu"
print("###################################################################")
print(f"Vision Transformer Conformal Prediction For Test Run: {test_run}")
print(f"The device for running the experiment is {device}")
print("###################################################################")

# Basic version prints
print(f"torch version: {torch.__version__}")
print(f"torchvision version: {torchvision.__version__}")

# ============================================================
# Dataset paths and summary
# ============================================================
DATA_DIRECTORY = args.data_dir #f"/leonardo_work/IscrC_SKIDD-AI/alitariqnagi_work/training_data_split_run_{test_run}"
main_folder = DATA_DIRECTORY
```

```python
splits = ['train', 'test', 'val']

summary_dict = {}
for split in splits:
    split_path = os.path.join(main_folder, split)
    total_files = 0
    summary_dict[split] = {}

    if not os.path.exists(split_path):
        print(f"Directory not found: {split_path}")
        continue

    for disease in os.listdir(split_path):
        disease_path = os.path.join(split_path, disease)
        if not os.path.isdir(disease_path):
            continue

        num_files = len([
            f for f in os.listdir(disease_path)
            if os.path.isfile(os.path.join(disease_path, f))
        ])

        summary_dict[split][disease] = num_files
        total_files += num_files

    print(f"\n{split.upper()} - Total files: {total_files}")
    for disease, count in summary_dict[split].items():
        print(f"  {disease}: {count} files")

train_dir = f"{DATA_DIRECTORY}/train"
val_dir   = f"{DATA_DIRECTORY}/val"
test_dir  = f"{DATA_DIRECTORY}/test"

# ============================================================
# Data transforms and loaders
# ============================================================
IMG_SIZE = 224
manual_transforms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
])
print(f"Manually created transforms: {manual_transforms}")

from going_modular import data_setup
from going_modular.helper_functions import set_seeds
set_seeds()

BATCH_SIZE = 32

# Create an initial dataloader to get class_names
train_dataloader, _, class_names = data_setup.create_dataloaders(
    train_dir=train_dir,
    test_dir=val_dir,
    transform=manual_transforms,
    batch_size=BATCH_SIZE
)

print(f"The class names are: {class_names}")

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

```python
# ViT model loading checkpoint
# ============================================================
pretrained_vit_weights = torchvision.models.ViT_B_16_Weights.DEFAULT
pretrained_vit = torchvision.models.vit_b_16(weights=pretrained_vit_weights).to(device)

# Replace classifier head
pretrained_vit.heads = nn.Linear(in_features=768, out_features=len(class_names)).to(device)

summary(model=pretrained_vit,
        input_size=(32, 3, 224, 224),
        col_names=["input_size", "output_size", "num_params", "trainable"],
        col_width=20,
        row_settings=["var_names"])

# Get transforms from ViT weights
pretrained_vit_transforms = pretrained_vit_weights.transforms()
print(pretrained_vit_transforms)

from going_modular.data_setup import create_dataloaders_no_shuffle
val_dataloader_pretrained, test_dataloader_pretrained, class_names = create_dataloaders_no_shuffle(
    train_dir=val_dir,
    test_dir=test_dir,
    transform=pretrained_vit_transforms,
    batch_size=32
)

# ============================================================
# Load saved ViT checkpoint
# ============================================================
print(f"[INFO] Loading pretrained ViT weights from: {vit_ckpt_path}")

# Handling DataParallel if multiple GPUs
if torch.cuda.device_count() > 1:
    print(f"[INFO] Using DataParallel on {torch.cuda.device_count()} GPUs")
    pretrained_vit = torch.nn.DataParallel(pretrained_vit)

pretrained_vit.to(device)

state_dict = torch.load(vit_ckpt_path, map_location=device)
try:
    pretrained_vit.load_state_dict(state_dict)
except RuntimeError as e:
    print("[WARNING] Error loading state_dict, possibly DataParallel/non-DataParallel mismatch.")
    print("Error was:", e)
    raise

pretrained_vit.eval()
print("[INFO] ViT checkpoint loaded successfully.\n")

# ============================================================
# Conformal prediction functions (unchanged)
# ============================================================

# alpha will be overridden with different valies in the loop
alpha = 0.1

```

```python
def calibration_scores(model, data_loader, device):
    model.eval()
    scores_calibration_set = []

    with torch.no_grad():
        for X, y in tqdm(data_loader,
                         desc="##################Proceeding with the Calculation of Calibration Scores##################"):
            X = X.to(device)
            y = y.to(device)

            y = y.cpu().numpy()
            pred_logits = model(X)
            cal_smx = softmax(pred_logits, dim=1).cpu().numpy()
            scores_per_batch = 1 - cal_smx[np.arange(y.shape[0]), y]
            scores_calibration_set.extend(scores_per_batch)

    scores_calibration_set = numpy.array(scores_calibration_set)
    return scores_calibration_set


def predict_with_conformal_sets(model, data_loader, calibration_scores_array, alpha, device):
    model.eval()
    prediction_sets = []
    true_labels = []

    n = len(calibration_scores_array)
    q_level = np.ceil((n + 1) * (1 - alpha)) / n
    qhat = np.quantile(calibration_scores_array, q_level, method='higher')

    with torch.no_grad():
        for X, y in tqdm(data_loader,
                         desc="##################Proceeding with the Calculation of Testing Scores##################"):
            X = X.to(device)
            y = y.cpu().numpy()
            pred_logits = model(X)

            val_smx = softmax(pred_logits, dim=1).cpu().numpy()
            masks = val_smx >= (1 - qhat)

            for m in masks:
                prediction_sets.append(np.where(m)[0].tolist())

            true_labels.extend(y.tolist())

    return prediction_sets, true_labels


def evaluate(prediction_sets, labels):
    number_of_correct_predictions = 0
    for pred_set, true_label in zip(prediction_sets, labels):
        if true_label in pred_set:
            number_of_correct_predictions += 1
    empirical_test_coverage = number_of_correct_predictions / len(labels)
    average_prediction_set_size = np.mean([len(s) for s in prediction_sets])
    return empirical_test_coverage, average_prediction_set_size
```

```python
def feature_stratified_coverage_by_class(prediction_sets, true_labels, alpha=0.1, class_names=None):
    class_to_indices = defaultdict(list)
    for idx, label in enumerate(true_labels):
        class_to_indices[label].append(idx)

    class_coverages = {}
    for cls, indices in class_to_indices.items():
        covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
        coverage = covered / len(indices)
        class_coverages[cls] = coverage

    print("\n Feature-Stratified Coverage (by True Class Label):")
    for cls, cov in sorted(class_coverages.items()):
        name = class_names[cls] if class_names else str(cls)
        print(f"  Class {name:<20}: Coverage = {cov:.3f} (Target ≥ {1 - alpha:.2f})")

    fsc_metric = min(class_coverages.values())
    print(f"\n FSC Metric (minimum class-wise coverage): {fsc_metric:.3f}")
    return fsc_metric, class_coverages


def plot_class_wise_coverage(class_coverages, alpha=0.1, class_names=None, title="Class-wise Coverage (FSC)"):
    classes = list(class_coverages.keys())
    coverages = [class_coverages[c] for c in classes]
    labels = [class_names[c] if class_names else str(c) for c in classes]

    plt.figure(figsize=(12, 6))
    bars = plt.bar(labels, coverages, color="pink", edgecolor="black")

    target = 1 - alpha
    plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage ({target:.2f})")

    for bar, cov in zip(bars, coverages):
        if cov < target:
            bar.set_color('salmon')

    plt.xticks(rotation=45, ha="right")
    plt.ylim(0, 1.05)
    plt.ylabel("Coverage")
    plt.title(title)
    plt.legend()
    plt.tight_layout()
    filename = f"Class_wise_Coverage_(FSC)_{test_run}_{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


def plot_coverage_vs_class_frequency(class_coverages, true_labels, alpha=0.1, class_names=None,
                                     title="Coverage vs Class Frequency"):
    class_indices = sorted(class_coverages.keys())
    coverages = [class_coverages[c] for c in class_indices]

    total = len(true_labels)
    counts = Counter(true_labels)
    frequencies = [counts[c] / total for c in class_indices]

```

```
    labels = [class_names[c] if class_names else str(c) for c in class_indices]
    x = np.arange(len(class_indices))

    fig, ax1 = plt.subplots(figsize=(12, 6))

    bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink', edgecolor='black')
    ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 - alpha:.2f})")
    ax1.set_ylim(0, 1.05)
    ax1.set_ylabel("Coverage", color='pink')
    ax1.tick_params(axis='y', labelcolor='pink')

    for bar, cov in zip(bars1, coverages):
        if cov < (1 - alpha):
            bar.set_color('salmon')

    ax2 = ax1.twinx()
    bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency", color='gray', alpha=0.5)
    ax2.set_ylabel("Class Frequency", color='gray')
    ax2.tick_params(axis='y', labelcolor='gray')

    plt.xticks(x, labels, rotation=45, ha="right")
    plt.title(title)
    fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
    plt.tight_layout()
    filename = f"plot_coverage_vs_class_frequency_{test_run}_{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


def plot_coverage_vs_class_frequency_with_correlation(class_coverages, true_labels, alpha=0.1, class_names=None,
                                                      title="Coverage vs Class Frequency"):
    class_indices = sorted(class_coverages.keys())
    coverages = [class_coverages[c] for c in class_indices]

    total = len(true_labels)
    counts = Counter(true_labels)
    frequencies = [counts[c] / total for c in class_indices]

    labels = [class_names[c] if class_names else str(c) for c in class_indices]
    x = np.arange(len(class_indices))

    pearson_corr, pearson_p = pearsonr(frequencies, coverages)
    spearman_corr, spearman_p = spearmanr(frequencies, coverages)

    print(f"\n Correlation between class frequency and coverage:")
    print(f"  - Pearson  r = {pearson_corr:.3f}, p = {pearson_p:.3e}")
    print(f"  - Spearman ρ = {spearman_corr:.3f}, p = {spearman_p:.3e}")

    fig, ax1 = plt.subplots(figsize=(12, 6))

    bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink', edgecolor='black')
    ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 - alpha:.2f})")
    ax1.set_ylim(0, 1.05)
```

```python
    ax1.set_ylabel("Coverage", color='pink')
    ax1.tick_params(axis='y', labelcolor='pink')

    for bar, cov in zip(bars1, coverages):
        if cov < (1 - alpha):
            bar.set_color('salmon')

    ax2 = ax1.twinx()
    bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency", color='gray', alpha=0.5)
    ax2.set_ylabel("Class Frequency", color='gray')
    ax2.tick_params(axis='y', labelcolor='gray')

    plt.xticks(x, labels, rotation=45, ha="right")
    plt.title(title)
    fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
    plt.tight_layout()
    filename = f"plot_coverage_vs_class_frequency_with_correlation_{test_run}_{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


def size_stratified_coverage(prediction_sets, true_labels, alpha=0.1,
                             bins=[[1], [2], [3, 4, 5], list(range(6, 100))]):
    bin_groups = defaultdict(list)

    for i, pred_set in enumerate(prediction_sets):
        set_size = len(pred_set)
        for bin_range in bins:
            if set_size in bin_range:
                bin_groups[str(bin_range)].append(i)
                break

    bin_coverages = {}
    for bin_key, indices in bin_groups.items():
        covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
        coverage = covered / len(indices) if indices else 0
        bin_coverages[bin_key] = coverage

    print("\n Size-Stratified Coverage (SSC):")
    for bin_key, cov in bin_coverages.items():
        print(f"  Set Size {bin_key:10s}: Coverage = {cov:.3f} (Target ≥ {1 - alpha:.2f})")

    ssc = min(bin_coverages.values())
    print(f"\n SSC Metric (min bin-wise coverage): {ssc:.3f}")
    return ssc, bin_coverages


def plot_ssc_coverage(bin_coverages, alpha=0.1, title="Size-Stratified Coverage"):
    bin_labels = list(bin_coverages.keys())
    coverages = [bin_coverages[k] for k in bin_labels]

    plt.figure(figsize=(10, 5))
    bars = plt.bar(bin_labels, coverages, color='pink', edgecolor='black')

    target = 1 - alpha
    plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage ({target:.2f})")
```

```python

    for bar, cov in zip(bars, coverages):
        if cov < target:
            bar.set_color('grey')

    plt.xticks(rotation=45, ha='right')
    plt.ylim(0, 1.05)
    plt.ylabel("Coverage")
    plt.title(title)
    plt.legend()
    plt.tight_layout()
    filename = f"plot_ssc_coverage_{test_run}__{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


def plot_prediction_set_size_by_class(prediction_sets, true_labels, class_names=None, results_dir="", test_run="",
                                      timestamp=""):
    size_per_class = defaultdict(list)
    for pred_set, true_label in zip(prediction_sets, true_labels):
        size_per_class[true_label].append(len(pred_set))

    sorted_classes = sorted(size_per_class.keys())
    means = [np.mean(size_per_class[c]) for c in sorted_classes]
    stds = [np.std(size_per_class[c]) for c in sorted_classes]
    labels = [class_names[c] if class_names else str(c) for c in sorted_classes]

    plt.figure(figsize=(12, 6))
    plt.bar(labels, means, yerr=stds, capsize=5, color="lightgreen", edgecolor="black")
    plt.ylabel("Avg Prediction Set Size")
    plt.xlabel("Class")
    plt.title("Average Prediction Set Size by Class")
    plt.xticks(rotation=45, ha="right")
    plt.grid(axis="y", linestyle="--", alpha=0.5)
    plt.tight_layout()

    filename = f"prediction_set_size_by_class_{test_run}__{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


def plot_calibration_score_distribution(calibration_scores, results_dir="", test_run="", timestamp=""):
    plt.figure(figsize=(10, 5))
    plt.hist(calibration_scores, bins=30, color="pink", edgecolor="black", alpha=0.8)
    plt.xlabel("Nonconformity Score (1 - P(True Class))")
    plt.ylabel("Frequency")
    plt.title("Calibration Score Distribution")
    plt.grid(axis='y', linestyle='--', alpha=0.6)
    plt.tight_layout()

    filename = f"calibration_score_distribution_{test_run}__{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


# ============================================================
# Multiple alpha values conformal evaluation
```

```python
# ============================================================
timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")

base_results_root = f"Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}"
os.makedirs(base_results_root, exist_ok=True)
print("Base conformal results directory:", base_results_root)

print("\n====== Computing calibration scores on validation set ======\n")
results_directory = base_results_root
scores_calibration = calibration_scores(
    model=pretrained_vit,
    data_loader=val_dataloader_pretrained,
    device=device
)

np.save(
    os.path.join(base_results_root,
                 f"calibration_scores_experiment_run_{test_run}_{timestamp}.npy"),
    scores_calibration
)
with open(
    os.path.join(base_results_root,
                 f"scores_calibration_run_{test_run}__{timestamp}.pkl"),
    "wb"
) as f:
    pickle.dump(scores_calibration, f)

plot_calibration_score_distribution(
    calibration_scores=scores_calibration,
    results_dir=base_results_root,
    test_run=test_run,
    timestamp=timestamp
)

coverage_levels = np.arange(0.65, 0.95 + 1e-9, 0.05)
all_alpha_metrics = []

for coverage_target in coverage_levels:
    alpha = 1.0 - coverage_target

    print("\n====================================================")
    print(f" Target coverage: {coverage_target:.2f}  |  alpha = {alpha:.2f}")
    print("====================================================\n")

    cov_tag = int(round(coverage_target * 100))
    results_directory = os.path.join(base_results_root, f"coverage_{cov_tag}")
    os.makedirs(results_directory, exist_ok=True)

    prediction_sets, true_labels = predict_with_conformal_sets(
        model=pretrained_vit,
        data_loader=test_dataloader_pretrained,
        calibration_scores_array=scores_calibration,
        alpha=alpha,
        device=device
    )
    test_set_coverage, avg_size = evaluate(prediction_sets=prediction_sets, labels=true_labels)

    log_path = os.path.join(
```

```
        results_directory,
        f"logs_experiment_run_{test_run}_alpha_{alpha:.2f}_{timestamp}.txt"
    )
    with open(log_path, "w") as f:
        f.write("########################### Results ###################################\n")
        f.write(f"test_run {test_run}\n")
        f.write(f"Target coverage: {coverage_target:.3f}\n")
        f.write(f"Alpha: {alpha:.3f}\n")
        f.write(f"Coverage: {test_set_coverage:.3f}\n")
        f.write(f"Average Prediction Set Size for Test Set: {avg_size:.3f}\n")
        f.write("######################################################################\n")

    print("###########################Results:###################################")
    print(f"test_run {test_run}")
    print(f"Target coverage: {coverage_target:.3f}")
    print(f"Alpha: {alpha:.3f}")
    print(f"Coverage: {test_set_coverage:.3f}")
    print(f"Average Prediction Set Size for Test Set: {avg_size:.3f}")
    print("######################################################################")

    with open(
        os.path.join(results_directory,
                     f"prediction_data_experiment_run_{test_run}__{timestamp}.pkl"),
        "wb"
    ) as f:
        pickle.dump(
            {
                "prediction_sets": prediction_sets,
                "true_labels": true_labels
            },
            f
        )

    # FSC & related plots
    fsc_metric, class_coverages = feature_stratified_coverage_by_class(
        prediction_sets,
        true_labels,
        alpha=alpha,
        class_names=class_names
    )
    plot_class_wise_coverage(class_coverages, alpha=alpha, class_names=class_names)
    plot_coverage_vs_class_frequency(class_coverages, true_labels, alpha=alpha, class_names=class_names)
    plot_coverage_vs_class_frequency_with_correlation(
        class_coverages=class_coverages,
        true_labels=true_labels,
        alpha=alpha,
        class_names=class_names
    )

    # SSC & related plots
    ssc_metric, bin_coverages = size_stratified_coverage(
        prediction_sets,
        true_labels,
        alpha=alpha,
        bins=[[1], [2], [3], list(range(4, 100))]
    )
    plot_ssc_coverage(bin_coverages, alpha=alpha)
```

```

    # Prediction set size by class
    plot_prediction_set_size_by_class(
        prediction_sets=prediction_sets,
        true_labels=true_labels,
        class_names=class_names,
        results_dir=results_directory,
        test_run=test_run,
        timestamp=timestamp
    )

    metrics = {
        "test_run": test_run,
        "timestamp": timestamp,
        "alpha": alpha,
        "target_coverage": coverage_target,
        "test_set_coverage": test_set_coverage,
        "average_prediction_set_size": avg_size,
        "FSC": fsc_metric,
        "SSC": ssc_metric,
        "class_coverages": {str(k): float(v) for k, v in class_coverages.items()},
        "bin_coverages": bin_coverages
    }

    with open(os.path.join(results_directory,
f"metrics_summary_{test_run}_alpha_{alpha:.2f}_{timestamp}.json"), "w") as f:
        json.dump(metrics, f, indent=4)

    all_alpha_metrics.append(metrics)

with open(os.path.join(base_results_root,
                       f"metrics_summary_all_alphas_{test_run}_{timestamp}.json"), "w") as f:
    json.dump(all_alpha_metrics, f, indent=4)

print("\nFinished multi-alpha conformal evaluation for coverage 0.65-0.95 (step 0.05).")
print("All conformal results saved under:", base_results_root)
```