```python
import argparse
import os
from datetime import datetime
from collections import Counter, defaultdict
import json
import pickle

import numpy as np
import torch
from torch import nn
from torchvision import datasets, transforms, models
import torchvision
import matplotlib.pyplot as plt
from torchsummary import summary
from tqdm import tqdm
import pandas as pd
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("--resnet_ckpt_path", type=str, required=True,
                    help="Path to saved ResNet checkpoint (.pth)")
args = parser.parse_args()

test_run = args.test_run
resnet_ckpt_path = args.resnet_ckpt_path

# ============================================================
# Device configuration
# ============================================================
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Device:", device)
print("test_run", test_run)
print("resnet_ckpt_path:", resnet_ckpt_path)

# ============================================================
# DATA LOADING
# ============================================================
transformations = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

base_data_dir = f"/leonardo_work/IscrC_ArtLLMs/alitariqnagi_work/training_data_split_run_{test_run}"

train_set_mix = datasets.ImageFolder(
    os.path.join(base_data_dir, "train"),
    transform=transformations
)
```

```python
val_set_mix = datasets.ImageFolder(
    os.path.join(base_data_dir, "val"),
    transform=transformations
)
test_set_mix = datasets.ImageFolder(
    os.path.join(base_data_dir, "test"),
    transform=transformations
)

print("Train size:", len(train_set_mix))
print("Val size:", len(val_set_mix))
print("Test size:", len(test_set_mix))

class_names = train_set_mix.classes
print("Classes:", class_names)

train_loader_mix = torch.utils.data.DataLoader(
    train_set_mix, batch_size=32, shuffle=True, num_workers=8
)
val_loader_mix = torch.utils.data.DataLoader(
    val_set_mix, batch_size=32, shuffle=False, num_workers=8
)
test_loader_mix = torch.utils.data.DataLoader(
    test_set_mix, batch_size=32, shuffle=False, num_workers=8
)

# Optional: quick visualization
dataiter = iter(train_loader_mix)
images, labels = next(dataiter)


def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis("off")


imshow(torchvision.utils.make_grid(images))
plt.show()

model = models.resnet50(weights='ResNet50_Weights.IMAGENET1K_V1')

for param in model.parameters():
    param.requires_grad = True

model.fc = nn.Sequential(
    nn.Linear(2048, 512),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(512, len(class_names))
)

if torch.cuda.device_count() > 1:
    print(f"Using DataParallel Mode with cuda count {torch.cuda.device_count()}")
    model = torch.nn.DataParallel(model)

model.to(device)
print(summary(model, input_size=(3, 224, 224)))

```

```python
# ============================================================
# LOAD FINAL CHECKPOINT (NO TRAINING HERE)
# ============================================================
final_ckpt_path = resnet_ckpt_path
print("Loading final model checkpoint:", final_ckpt_path)

# Load on CPU first (safer across GPU setups)
state_dict = torch.load(final_ckpt_path, map_location="cpu")

# If checkpoint was saved using DataParallel, keys will start with "module."
if any(k.startswith("module.") for k in state_dict.keys()):
    print("[INFO] Detected DataParallel checkpoint. Stripping 'module.' prefix...")
    state_dict = {k.replace("module.", "", 1): v for k, v in state_dict.items()}

model.load_state_dict(state_dict, strict=True)

model.to(device)
model.eval()
print("Model loaded.\n")

# ============================================================
# CONFORMAL FUNCTIONS
# ============================================================

# different values will be overwritten in the loop
alpha = 0.1


##########################--------------------------------------
# Calculation of Calibration Scores
# ---------------------------------------------------------------
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


##################################################################################################################################################

```

```python

##########################--------------------------------------
# Calculation of Prediction Sets
# ---------------------------------------------------------------
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)
            val_smx = val_smx.cpu().numpy()

            masks = val_smx >= (1 - qhat)

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

            true_labels.extend(y.tolist())

    return prediction_sets, true_labels


####################################################################################################################################################


##########################--------------------------------------
# EVALUATION
# ---------------------------------------------------------------
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


####################################################################################################################################################


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)
    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})")

    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()


# ============================================================
# MAIN CONFORMAL EVALUATION FOR MULTIPLE ALPHA VALUES
# ============================================================

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

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 results directory:", base_results_root)

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

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=model,
        data_loader=test_loader_mix,
        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("######################################################################")

    set_sizes = [len(prediction_set) for prediction_set in prediction_sets]
    counts = Counter(set_sizes)
    total = sum(counts.values())
    sizes = sorted(counts.keys())
    frequencies = [counts[size] for size in sizes]
    percentages = [100 * freq / total for freq in frequencies]

    data = [{'size': len(prediction_set), 'correct': int(true in prediction_set)}
            for prediction_set, true in zip(prediction_sets, true_labels)]
    df = pd.DataFrame(data)
    coverage_stats = df.groupby('size').agg({'correct': ['mean', 'sum', 'count']})
    coverage_stats.columns = ['coverage', 'correct_count', 'total_count']
    coverage_stats = coverage_stats.reindex(sizes, fill_value=0)

    cumulative_correct = coverage_stats['correct_count'].cumsum()
    cumulative_coverage = 100 * cumulative_correct / len(true_labels)
    max_freq = max(frequencies)
    cumulative_scaled = (cumulative_coverage / 100) * max_freq

    fig, ax = plt.subplots(figsize=(13, 6))
    bars = ax.bar(sizes, frequencies, color='pink', edgecolor='black', label='Frequency', alpha=0.7)
    ax.plot(sizes, cumulative_scaled, color='red', marker='o', linestyle='-',
            alpha=0.4, linewidth=2, label='Cumulative Coverage')

    for bar, size, pct in zip(bars, sizes, percentages):
        height = bar.get_height()
        count = counts[size]
        coverage_val = coverage_stats.loc[size, 'coverage'] * 100
        ax.annotate(f'{count} ({pct:.1f}%)\nCov: {coverage_val:.1f}%',
                    xy=(bar.get_x() + bar.get_width() / 2, height / 2),
                    ha='center', va='center', fontsize=9)

    for bar, cum_cov in zip(bars, cumulative_coverage):
        height = bar.get_height()
        ax.annotate(f'CumCov: {cum_cov:.1f}%',
                    xy=(bar.get_x() + bar.get_width() / 2, height),
                    xytext=(0, 8),
                    textcoords="offset points",
                    ha='center', va='bottom', fontsize=9, color='darkorange')

    mean_size = np.mean(set_sizes)
    median_size = np.median(set_sizes)
    ax.axvline(mean_size, color='gray', linestyle='--', label=f'Mean = {mean_size:.2f}')
    ax.axvline(median_size, color='orange', linestyle='--', label=f'Median = {median_size:.2f}')

    ax.set_xlabel("Prediction Set Size")
    ax.set_ylabel("Frequency")
    ax.set_title(f"Prediction Set Size Distribution with Coverage (Test Set), alpha={alpha:.2f}")
    ax.set_xticks(sizes)
    ax.grid(axis='y', linestyle='--', alpha=0.6)
    ax.legend(loc='upper right')

    plt.tight_layout()
    plt.savefig(os.path.join(
        results_directory,
        f"Prediction_Set_Size_Distribution_with_Coverage_(Test_Set)_{test_run}_alpha_{alpha:.2f}_{timestamp}.png"
    ))
    plt.show()

    fsc_metric, per_class_cov = feature_stratified_coverage_by_class(
        prediction_sets,
        true_labels,
        alpha=alpha,
        class_names=class_names
    )

    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)

    fsc_metric, class_coverages = feature_stratified_coverage_by_class(
        prediction_sets,
        true_labels,
        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_metric, bin_coverages = size_stratified_coverage(
        prediction_sets,
        true_labels,
        alpha=alpha,
        bins=[[1], [2], [3], list(range(4, 100))]
    )
```

```python

    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)

    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 alphas corresponding to
coverage 0.65-0.95.")
print("Results saved under:", base_results_root)

```