In [1]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.decomposition import PCA
import numpy as np
from sklearn.metrics import accuracy_score,precision_score, recall_score, f1_score
from RevGEN_MLP import RevGEN_MLP
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap
from sklearn.ensemble import IsolationForest,RandomForestClassifier
import torch
import torch.nn as nn
import torch.optim as optim
```

RevGEN-MLP

Loading and Preparing Data for Training

In [2]:
```
# Load full Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split into train/test
X_train, X_test, y_train, y_test= train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
# One-hot encode labels
encoder = OneHotEncoder(sparse_output=False)
y_train_encoded = encoder.fit_transform(y_train.reshape(-1, 1))
y_test_encoded =encoder.transform(y_test.reshape(-1, 1))

# Scale features
ANN_scaler = StandardScaler()
X_train_scaled = ANN_scaler.fit_transform(X_train)
X_test_scaled = ANN_scaler.transform(X_test)
```

Training RevGEN-MLP

In [3]:
```
num_layer = 1
num_epochs = 160
```

In [4]:
```
import numpy as np

# Set seed for reproducibility
np.random.seed(42)

# Initialize model
model = RevGEN_MLP(
    n_layers=num_layer,
    x=X_train_scaled[0].reshape(-1, 1),
    y_actual=y_train_encoded[0].reshape(-1, 1),
    epochs=num_epochs,
    loss_function="cross_entropy"
)

# Training loop
for epoch in range(num_epochs):
    loss_epoch = 0
    correct_train = 0

    # Shuffle training indices
    indices = np.random.permutation(X_train_scaled.shape[0])

    for i in indices:
        x_sample = X_train_scaled[i].reshape(-1, 1)
        y_sample = y_train_encoded[i].reshape(-1, 1)

        # Train and accumulate loss
        model.train(input=x_sample, target=y_sample)
        loss_epoch += model.loss_fn(input=x_sample, target=y_sample)

        # Predict and count correct predictions
        pred = model.forward(x_sample)
        if np.argmax(pred[:3]) == np.argmax(y_sample):
            correct_train += 1
```

```
    # Compute average training metrics
    avg_train_loss = loss_epoch / X_train_scaled.shape[0]
    train_accuracy = correct_train / X_train_scaled.shape[0]

    # Evaluate on test set every 10 epochs
    if epoch % 10 == 0 or epoch == num_epochs - 1:
        correct_test = 0
        test_loss_epoch = 0

        for i in range(X_test_scaled.shape[0]):
            x_sample = X_test_scaled[i].reshape(-1, 1)
            y_sample = y_test_encoded[i].reshape(-1, 1)

            pred = model.forward(x_sample)
            if np.argmax(pred[:3]) == np.argmax(y_sample):
                correct_test += 1

            test_loss_epoch += model.loss_fn(input=x_sample, target=y_sample)

        avg_test_loss = test_loss_epoch / X_test_scaled.shape[0]
        test_accuracy = correct_test / X_test_scaled.shape[0]

        print(f"Epoch {epoch:3d} | "
              f"Train Loss: {avg_train_loss:.4f} | Train Acc: {train_accuracy * 100:.2f}% | "
              f"Test Loss: {avg_test_loss:.4f} | Test Acc: {test_accuracy * 100:.2f}%")
```

```
Epoch   0 | Train Loss: 1.4200 | Train Acc: 48.57% | Test Loss: 1.3995 | Test Acc: 40.00%
Epoch  10 | Train Loss: 1.1029 | Train Acc: 46.67% | Test Loss: 1.1234 | Test Acc: 46.67%
Epoch  20 | Train Loss: 0.9182 | Train Acc: 47.62% | Test Loss: 0.9430 | Test Acc: 44.44%
Epoch  30 | Train Loss: 0.5783 | Train Acc: 81.90% | Test Loss: 0.6211 | Test Acc: 77.78%
Epoch  40 | Train Loss: 0.4351 | Train Acc: 81.90% | Test Loss: 0.4981 | Test Acc: 77.78%
Epoch  50 | Train Loss: 0.3668 | Train Acc: 81.90% | Test Loss: 0.4451 | Test Acc: 77.78%
Epoch  60 | Train Loss: 0.3263 | Train Acc: 84.76% | Test Loss: 0.4178 | Test Acc: 80.00%
Epoch  70 | Train Loss: 0.2960 | Train Acc: 87.62% | Test Loss: 0.3980 | Test Acc: 80.00%
Epoch  80 | Train Loss: 0.2712 | Train Acc: 87.62% | Test Loss: 0.3811 | Test Acc: 82.22%
Epoch  90 | Train Loss: 0.2491 | Train Acc: 89.52% | Test Loss: 0.3639 | Test Acc: 82.22%
Epoch 100 | Train Loss: 0.2288 | Train Acc: 89.52% | Test Loss: 0.3456 | Test Acc: 82.22%
Epoch 110 | Train Loss: 0.2095 | Train Acc: 90.48% | Test Loss: 0.3266 | Test Acc: 84.44%
Epoch 120 | Train Loss: 0.1900 | Train Acc: 92.38% | Test Loss: 0.3003 | Test Acc: 86.67%
Epoch 130 | Train Loss: 0.1663 | Train Acc: 94.29% | Test Loss: 0.2733 | Test Acc: 86.67%
Epoch 140 | Train Loss: 0.1277 | Train Acc: 97.14% | Test Loss: 0.2248 | Test Acc: 86.67%
Epoch 150 | Train Loss: 0.1018 | Train Acc: 97.14% | Test Loss: 0.1852 | Test Acc: 91.11%
Epoch 159 | Train Loss: 0.0895 | Train Acc: 97.14% | Test Loss: 0.1643 | Test Acc: 91.11%
```

In [5]:

```
correct = 0
y_preds = []
y_true = []

for i in range(X_test_scaled.shape[0]):
    x_sample = X_test_scaled[i].reshape(-1, 1)
    y_sample = y_test_encoded[i].reshape(-1, 1)

    pred = model.forward(x_sample)
    pred_class = np.argmax(pred[0:3])
    true_class = np.argmax(y_sample)

    y_preds.append(pred_class)
    y_true.append(true_class)

    if pred_class == true_class:
        correct += 1



test_accuracy = correct / X_test_scaled.shape[0]
precision = precision_score(y_true, y_preds, average='macro', zero_division=0)
recall = recall_score(y_true, y_preds, average='macro', zero_division=0)
f1 = f1_score(y_true, y_preds, average='macro', zero_division=0)

print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
print(f"Precision:     {precision * 100:.2f}%")
print(f"Recall:        {recall * 100:.2f}%")
```

```
Test Accuracy: 91.11%
Precision:     91.55%
Recall:        91.11%
```

Invertibility Check

Small reconstruction errors close to zero are expected due to floating-point precision limits

In [6]:

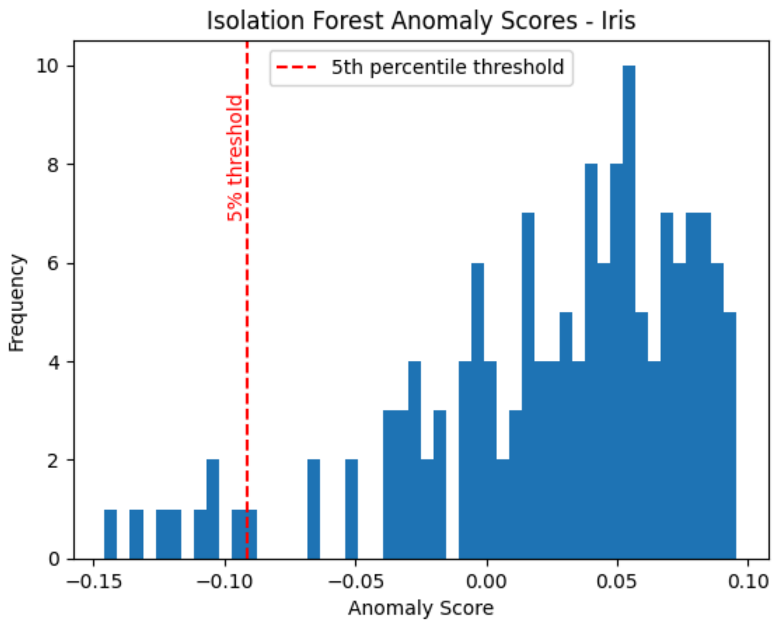
```
x_sample = X_test_scaled[0].reshape(-1, 1)

# Unscale
```

```
x_sample_unscaled = ANN_scaler.inverse_transform(x_sample.reshape(1, -1))
print("Original Sample:", x_sample_unscaled)

# Forward pass
pred = model.forward(x_sample)
print("\nOutput (Classes):", pred[0:3].ravel())
print("Output (Latent Variable):", pred[3:].ravel())
# Reconstruct
reconstructed_sample = model.reverse(pred)

# Unscale reconstruction
reconstructed_sample_unscaled = ANN_scaler.inverse_transform(
    reconstructed_sample.reshape(1, -1)
)
print("\nReconstructed Sample:", reconstructed_sample_unscaled.ravel())

mse_scaled = np.mean((x_sample - reconstructed_sample)**2)
mse_unscaled = np.mean((x_sample_unscaled - reconstructed_sample_unscaled)**2)
print("\nMSE Error (Scaled Data): ", mse_scaled)
print("MSE Error (Unscaled Data): ", mse_unscaled)
```

```
Original Sample: [[7.3 2.9 6.3 1.8]]

Output (Classes): [4.91879267e-05 3.73161612e-02 9.62634651e-01]
Output (Latent Variable): [-0.03801191]

Reconstructed Sample: [7.3 2.9 6.3 1.8]

MSE Error (Scaled Data):  1.6687082876931165e-26
MSE Error (Unscaled Data):  8.043879065070573e-27
```

Generation

In [7]:
```
# Fit the model
detector = IsolationForest(random_state=42).fit(X)
scores = detector.decision_function(X)

# Plot histogram
plt.hist(scores, bins=50)
threshold = np.percentile(scores, 5)
plt.axvline(threshold, color='red', linestyle='--', label='5th percentile threshold')

# Add label
plt.text(threshold, plt.ylim()[1]*0.9, '5% threshold', color='red', rotation=90, va='top', ha='right')

plt.title("Isolation Forest Anomaly Scores - Iris")
plt.xlabel("Anomaly Score")
plt.ylabel("Frequency")
plt.legend()
plt.show()
```



In [8]:
```
def generate_data_with_epsilon(model,epsilon):
    pred_classes = []
    inputs = []
    original_inputs = []
    original_classes = []
    original_prob = []
```

```
    pred_probability = []

    for test,truth in zip(X_test_scaled,y_test_encoded):
        x_sample = test.reshape(-1, 1)
        pred = model.forward(x_sample)

        class_probs = pred[0:3].ravel()
        max_prob = np.max(class_probs)
        if np.argmax(pred[:3]) == np.argmax(truth):
            if max_prob >= 0.8:
                original_prob.append(max_prob)
                pred_class = np.argmax(class_probs)
                original_inputs.append(ANN_scaler.inverse_transform(x_sample.reshape(1, -1))[0])
                original_classes.append(pred_class)
                vector = pred.copy()

                vector[3] = vector[3] + epsilon

                new_input = model.reverse(input=vector)
                new_pred = model.forward(new_input)
                max_prob = np.max(new_pred[0:3])
                pred_probability.append(max_prob)
                new_pred_class = np.argmax(new_pred[0:3])

                new_input_unscaled = ANN_scaler.inverse_transform(new_input.reshape(1, -1))[0]

                pred_classes.append(new_pred_class)
                inputs.append(new_input_unscaled)
    data = {
    "sepal length (cm)": [],
    "sepal width (cm)": [],
    "petal length (cm)": [],
    "petal width (cm)": [],
    "Class": [],
    "Probability": []
    }

    for sample in inputs:
        data["sepal length (cm)"].append(sample[0])
        data["sepal width (cm)"].append(sample[1])
        data["petal length (cm)"].append(sample[2])
        data["petal width (cm)"].append(sample[3])

    for cls in pred_classes:
        data["Class"].append(cls)

    for prob in pred_probability:
        data["Probability"].append(prob*100)

    generated_df = pd.DataFrame(data=data)
    return generated_df
```

Anomaly Score Threshold

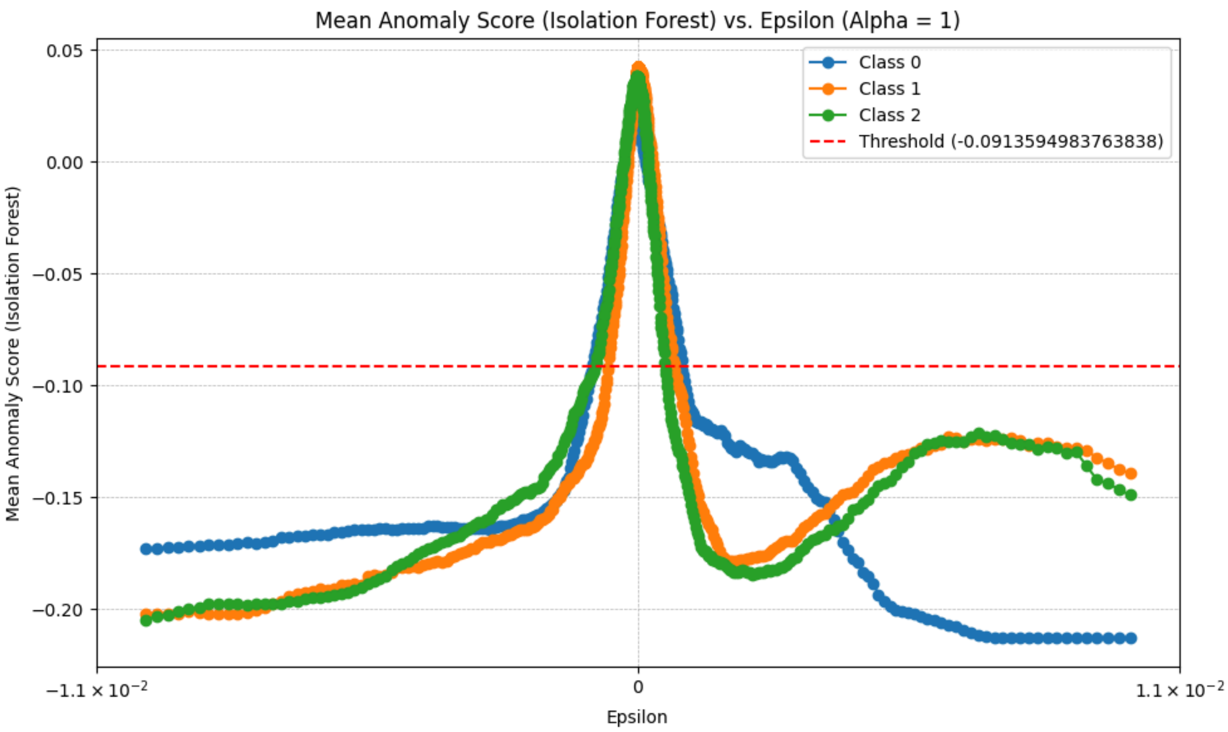
The following code decides the anomaly score threshold used to guide generation. The first threshold represents moderately anomalous data whereas the second threshold represents extremely anomalous data.

In [9]:
```
threshold_1 = np.percentile(scores,5)
threshold_2 = -0.15
```

In [10]:
```
exponents = np.linspace(-8, -2 , 600)
positive_epsilons = 10 ** exponents
negative_epsilons = -positive_epsilons
epsilons = np.sort(np.concatenate([negative_epsilons, positive_epsilons]))

threshold = threshold_1 # Change threshold as needed

results = []

for epsilon in epsilons:
    generated_df = generate_data_with_epsilon(model, epsilon)
    anomaly_score_results = []

    for cls in generated_df["Class"].unique():
        subset = generated_df[generated_df["Class"] == cls]
        features = subset.drop(columns=["Class", "Probability"]).to_numpy()
        score = detector.decision_function(features)

        anomaly_score_results.append({
            "Class": cls,
            "Score": score.mean()
        })

    scored_df = pd.DataFrame(anomaly_score_results)
```

```
    mean_score_by_class = scored_df.set_index("Class")["Score"]
    below_threshold = mean_score_by_class[mean_score_by_class < threshold].to_dict()

    results.append({
        "epsilon": epsilon,
        "mean_score": mean_score_by_class.to_dict(),
    })
```

In [11]:
```
class_names = []
for r in results:
    for cls in r["mean_score"]:
        if cls not in class_names:
            class_names.append(cls)
class_names.sort()

# Plot
plt.figure(figsize=(10, 6))
for cls in class_names:
    eps = []
    overlaps = []
    for r in results:
        eps.append(r["epsilon"])
        overlaps.append(r["mean_score"].get(cls))
    plt.plot(eps, overlaps, marker='o', label=f'Class {cls}')


# Threshold line
plt.axhline(y=threshold, color='red', linestyle='--', label=f'Threshold ({threshold})')

# Log scale on x-axis

plt.xlabel("Epsilon")
plt.ylabel("Mean Anomaly Score (Isolation Forest)")

plt.title("Mean Anomaly Score (Isolation Forest) vs. Epsilon (Alpha = 1)")
plt.legend()
plt.xscale("symlog")
plt.grid(True, which="both", ls="--", linewidth=0.5)
plt.tight_layout()
plt.show()
```



In [12]:
```
best_epsilons = {}
best_eps_overall = None
max_abs_eps = 0  # Track largest absolute epsilon

for cls in class_names:
    best_pos_eps = None
    best_neg_eps = None
    best_pos_diff = float('inf')
    best_neg_diff = float('inf')

    for r in results:
        score = r["mean_score"].get(cls)
        eps = r["epsilon"]
```

```
        if score is not None and score < threshold:
            diff = threshold - score

            if eps > 0 and diff < best_pos_diff:
                best_pos_diff = diff
                best_pos_eps = eps

            elif eps < 0 and diff < best_neg_diff:
                best_neg_diff = diff
                best_neg_eps = eps

    best_epsilons[cls] = {
        "best_positive": best_pos_eps,
        "best_negative": best_neg_eps
    }

    # Update global best epsilon if this one is larger in magnitude
    for eps in [best_pos_eps, best_neg_eps]:
        if eps is not None and abs(eps) > max_abs_eps:
            max_abs_eps = abs(eps)
            best_eps_overall = eps

# Display results
for cls, eps_dict in best_epsilons.items():
    print(f"Class {cls}:")
    print(f"  Best positive epsilon: {eps_dict['best_positive']}")
    print(f"  Best negative epsilon: {eps_dict['best_negative']}")

print(f"\nBest overall epsilon across all classes: {best_eps_overall}")
```

```
Class 0:
  Best positive epsilon: 0.0009083720712343382
  Best negative epsilon: -0.0009295665071788839
Class 1:
  Best positive epsilon: 0.0007553186652841878
  Best negative epsilon: -0.0005997403546276515
Class 2:
  Best positive epsilon: 0.0005596457224360915
  Best negative epsilon: -0.0008674219025756638

Best overall epsilon across all classes: -0.0009295665071788839
```

Generated Confidently Classified Anomalies

In [13]:

```
epsilon = max_abs_eps


pred_classes = []
inputs = []
original_inputs = []
original_classes = []
original_prob = []
pred_probability = []

for test,truth in zip(X_test_scaled,y_test_encoded):
    x_sample = test.reshape(-1, 1)
    pred = model.forward(x_sample)

    class_probs = pred[0:3].ravel()
    max_prob = np.max(class_probs)
    if np.argmax(pred[:3]) == np.argmax(truth):
        if max_prob >= 0.8:
            original_prob.append(max_prob)
            pred_class = np.argmax(class_probs)
            original_inputs.append(ANN_scaler.inverse_transform(x_sample.reshape(1, -1))[0])
            original_classes.append(pred_class)

            # Revese pass for when epsilon is substracted to latent variables
            vector_neg_epsilon = pred.copy()
            vector_neg_epsilon[3] = vector_neg_epsilon[3] - epsilon

            new_input_neg = model.reverse(input=vector_neg_epsilon)
            new_pred = model.forward(new_input_neg)
            max_prob = np.max(new_pred[0:3])
            pred_probability.append(max_prob)
            new_pred_class_neg = np.argmax(new_pred[0:3])

            # Revese pass for when epsilon is added to latent variables
            vector_pos_epsilon = pred.copy()
            vector_pos_epsilon[3] = vector_pos_epsilon[3] + epsilon

            new_input_pos = model.reverse(input=vector_pos_epsilon)
            new_pred = model.forward(new_input_pos)
            max_prob_pos = np.max(new_pred[0:3])
```

```
            pred_probability.append(max_prob_pos)
            new_pred_class_pos = np.argmax(new_pred[0:3])

            new_input_unscaled_pos = ANN_scaler.inverse_transform(new_input_pos.reshape(1, -1))[0]
            new_input_unscaled_neg = ANN_scaler.inverse_transform(new_input_neg.reshape(1, -1))[0]

            pred_classes.append(new_pred_class_pos)
            inputs.append(new_input_unscaled_pos)

            pred_classes.append(new_pred_class_neg)
            inputs.append(new_input_unscaled_neg)



data = {
    "sepal length (cm)": [],
    "sepal width (cm)": [],
    "petal length (cm)": [],
    "petal width (cm)": [],
    "Class": [],
    "Probability": []
}

for sample in inputs:
    data["sepal length (cm)"].append(sample[0])
    data["sepal width (cm)"].append(sample[1])
    data["petal length (cm)"].append(sample[2])
    data["petal width (cm)"].append(sample[3])

for cls in pred_classes:
    data["Class"].append(cls)

for prob in pred_probability:
    data["Probability"].append(prob*100)

generated_df = pd.DataFrame(data=data)
```

In [14]:
```
generated_df = generated_df.drop_duplicates()
```

In [15]:
```
# Predict
prediction_list = detector.predict(inputs).tolist()

# Count anomalies
anomalies = prediction_list.count(-1) / len(inputs)
print(f"Anomaly rate: {anomalies:.2%}")

# Convert inputs and predictions to NumPy arrays
inputs = np.array(inputs)
predictions = np.array(prediction_list)

# Extract anomalous inputs
anomalous_inputs = inputs[predictions == -1]



# Keep only rows where prediction == -1 (anomaly)
generated_df_anomalies = generated_df[predictions == -1].copy()

# Ensure generated_df aligns with inputs
generated_df_anomalies = generated_df_anomalies.reset_index(drop=True)
```

```
Anomaly rate: 97.06%
```

In [16]:
```
generated_df_anomalies=generated_df_anomalies.drop_duplicates().reset_index(drop=True)
```

In [17]:
```
prediction_list = detector.predict(X).tolist()

anomalies = prediction_list.count(-1)/len(X)

print(anomalies)
```

```
0.25333333333333335
```

In [18]:
```
generated_df_anomalies
```

Out[18]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Class	Probability
0	9.388680	4.084944	6.718618	1.619101	2	96.263465
1	5.211320	1.715056	5.881382	1.980899	2	96.263465
2	8.188680	4.084944	5.118618	1.219101	1	83.450492
3	4.011320	1.715056	4.281382	1.580899	1	83.450492
4	4.287438	2.606465	4.941069	2.758364	2	99.493944
...	...	...	...	...	...	...
61	3.124274	3.101826	0.318312	0.537370	0	99.852406
62	8.588680	4.184944	5.618618	1.819101	2	92.867839
63	4.411320	1.815056	4.781382	2.180899	2	92.867839
64	8.688680	4.084944	5.018618	1.119101	1	91.750577
65	4.511320	1.715056	4.181382	1.480899	1	91.750577

66 rows × 6 columns

Transferability Of Confidently Clasified Anomalies

Testing Settings

In [19]:
```
confidence_threshold = 0.8
```

Testing on Randomn Forest

In [20]:
```
RF_model = RandomForestClassifier(n_estimators=200,random_state=42)
RF_model.fit(X_train,y_train)


# Predict on test set
y_pred = RF_model.predict(X_test)


# Calculate accuracy
test_acc = accuracy_score(y_test, y_pred)
test_precision = precision_score(y_test, y_pred,average="macro")
test_recall = recall_score(y_test, y_pred,average="macro")
print("Test Accuracy:", test_acc)
print("Test Precision:", test_precision)
print("Test Recall:", test_recall)
```

```
Test Accuracy: 0.9111111111111111
Test Precision: 0.9155354449472096
Test Recall: 0.9111111111111111
```

In [21]:
```
generated_anomalies = generated_df_anomalies.drop(columns =["Class","Probability"])
```

In [22]:
```
# Predict probabilities for all inputs at once
probs_all = RF_model.predict_proba(generated_anomalies.values)

# Get max probabilities and predicted classes
max_probs = np.max(probs_all, axis=1)
pred_classes = np.argmax(probs_all, axis=1)

# Filter by threshold
mask = max_probs >= confidence_threshold
RF_anomalies_list = generated_anomalies.values[mask]
max_prob_rf = pred_classes[mask]
high_confidence_count = np.sum(mask)
```

In [23]:
```
robustness = high_confidence_count/len(anomalous_inputs)
print(robustness)
```

```
0.8333333333333334
```

In [24]:
```
feature_names = ["sepal length (cm)", "sepal width (cm)", "petal length (cm)", "petal width (cm)"]
RF_df = pd.DataFrame(RF_anomalies_list, columns = feature_names)
```

In [25]:
```
len(RF_anomalies_list)
```

Out[25]: 55

Testing On Neural Networks

In [26]:
```
# Set random seed
np.random.seed(42)
torch.manual_seed(42)
torch.cuda.manual_seed_all(42)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Model architecture
class SimpleNet(nn.Module):
    def __init__(self, input_dim):
        super(SimpleNet, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 20),
            nn.LeakyReLU(),
            nn.Linear(20, 3)
        )

    def forward(self, x):
        return self.net(x)

# Convert your data to PyTorch tensors
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test_scaled, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

# Initialize model, loss, optimizer
test_model = SimpleNet(input_dim=X_train_tensor.shape[1])
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(test_model.parameters(), lr=0.005)

# Training loop
num_epochs = 250
for epoch in range(num_epochs):
    test_model.train()
    optimizer.zero_grad()
    outputs = test_model(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()

    if epoch % 10 == 0 or epoch == num_epochs - 1:
        test_model.eval()
        with torch.no_grad():
            test_outputs = test_model(X_test_tensor)
            preds = torch.argmax(test_outputs, dim=1)
            acc = accuracy_score(y_test_tensor, preds)
            print(f"Epoch {epoch:3d} | Loss: {loss.item():.4f} | Test Accuracy: {acc * 100:.2f}%")
```

```
Epoch   0 | Loss: 1.1598 | Test Accuracy: 31.11%
Epoch  10 | Loss: 0.9245 | Test Accuracy: 68.89%
Epoch  20 | Loss: 0.7205 | Test Accuracy: 68.89%
Epoch  30 | Loss: 0.5427 | Test Accuracy: 77.78%
Epoch  40 | Loss: 0.4048 | Test Accuracy: 82.22%
Epoch  50 | Loss: 0.3084 | Test Accuracy: 86.67%
Epoch  60 | Loss: 0.2461 | Test Accuracy: 84.44%
Epoch  70 | Loss: 0.2030 | Test Accuracy: 84.44%
Epoch  80 | Loss: 0.1696 | Test Accuracy: 88.89%
Epoch  90 | Loss: 0.1428 | Test Accuracy: 91.11%
Epoch 100 | Loss: 0.1209 | Test Accuracy: 93.33%
Epoch 110 | Loss: 0.1032 | Test Accuracy: 93.33%
Epoch 120 | Loss: 0.0892 | Test Accuracy: 93.33%
Epoch 130 | Loss: 0.0782 | Test Accuracy: 93.33%
Epoch 140 | Loss: 0.0695 | Test Accuracy: 93.33%
Epoch 150 | Loss: 0.0626 | Test Accuracy: 93.33%
Epoch 160 | Loss: 0.0568 | Test Accuracy: 93.33%
Epoch 170 | Loss: 0.0521 | Test Accuracy: 93.33%
Epoch 180 | Loss: 0.0482 | Test Accuracy: 93.33%
Epoch 190 | Loss: 0.0449 | Test Accuracy: 93.33%
Epoch 200 | Loss: 0.0422 | Test Accuracy: 93.33%
Epoch 210 | Loss: 0.0399 | Test Accuracy: 93.33%
Epoch 220 | Loss: 0.0379 | Test Accuracy: 93.33%
Epoch 230 | Loss: 0.0361 | Test Accuracy: 93.33%
Epoch 240 | Loss: 0.0345 | Test Accuracy: 93.33%
Epoch 249 | Loss: 0.0332 | Test Accuracy: 93.33%
```

In [27]:
```
test_model.eval()
correct = 0
y_preds = []
y_true = []

for i in range(X_test_scaled.shape[0]):
```

```
    x_sample = torch.tensor(X_test_scaled[i].reshape(1, -1), dtype=torch.float32)
    y_sample = y_test_encoded[i].reshape(-1)  # Assuming one-hot encoded

    with torch.no_grad():
        logits = test_model(x_sample)
        probs = torch.softmax(logits, dim=1).numpy().flatten()
        pred_class = np.argmax(probs)
        true_class = np.argmax(y_sample)

    y_preds.append(pred_class)
    y_true.append(true_class)

    if pred_class == true_class:
        correct += 1

# Single sample prediction (optional)
x_sample = torch.tensor(X_test_scaled[0].reshape(1, -1), dtype=torch.float32)
with torch.no_grad():
    pred = test_model(x_sample)

# Metrics
test_accuracy = correct / len(X_test_scaled)
precision = precision_score(y_true, y_preds, average='macro', zero_division=0)
recall = recall_score(y_true, y_preds, average='macro', zero_division=0)
f1 = f1_score(y_true, y_preds, average='macro', zero_division=0)

print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
print(f"Precision:     {precision * 100:.2f}%")
print(f"Recall:        {recall * 100:.2f}%")
print(f"F1 Score:      {f1 * 100:.2f}%")
```

```
Test Accuracy: 93.33%
Precision:     93.45%
Recall:        93.33%
F1 Score:      93.33%

Precision:     93.45%
Recall:        93.33%
F1 Score:      93.33%
```

In [28]:

```
MLP_anomalies_list = []
high_confidence_count = 0
test = []
# Scale inputs
scaled_anomalous_inputs = ANN_scaler.transform(generated_anomalies.values)

# Set model to eval mode
test_model.eval()

# Loop through each anomalous input
for x in scaled_anomalous_inputs:
    x_tensor = torch.tensor(x.reshape(1, -1), dtype=torch.float32)  # shape: [1, input_dim]
    with torch.no_grad():
        logits = test_model(x_tensor)  # shape: [1, 2]
        probs = torch.softmax(logits, dim=1).numpy().flatten()  # convert to numpy array
        max_prob = np.max(probs)

    if max_prob >= confidence_threshold:
        high_confidence_count += 1
        MLP_anomalies_list.append(x)
        if np.argmax(probs) ==1:
            test.append(x)

# Inverse transform to original feature space
MLP_anomalies_list = ANN_scaler.inverse_transform(MLP_anomalies_list)

# Compute robustness score
robustness = high_confidence_count / len(generated_anomalies.values)
print(f"Robustness: {robustness:.4f}")
```

```
Robustness: 0.8485
```

In [29]:

```
len(generated_anomalies.values)
```

Out[29]: 66

In [30]:

```
len(MLP_anomalies_list)
```

Out[30]: 56

In [31]:

```
feature_names = ["sepal length (cm)", "sepal width (cm)", "petal length (cm)", "petal width (cm)"]
MLP_df = pd.DataFrame(MLP_anomalies_list, columns = feature_names)
```

KNN Classifier

In [32]:
```
from sklearn.neighbors import KNeighborsClassifier

n = 5
neigh = KNeighborsClassifier(n_neighbors=n)
neigh.fit(X_train_scaled, y_train)

# Predict on test set
y_pred = neigh.predict(X_test_scaled)


# Calculate accuracy
test_acc = accuracy_score(y_test, y_pred)
test_precision = precision_score(y_test, y_pred,average="macro")
test_recall = recall_score(y_test, y_pred,average="macro")
print("Test Accuracy:", test_acc)
print("Test Precision:", test_precision)
print("Test Recall:", test_recall)
```

```
Test Accuracy: 0.9111111111111111
Test Precision: 0.9298245614035089
Test Recall: 0.9111111111111111
```

In [33]:
```
KNN_anomalies_list = []
high_confidence_count = 0

scaled_anomalous_inputs = ANN_scaler.transform(generated_anomalies.values)

for x in scaled_anomalous_inputs:
    probs = neigh.predict_proba(x.reshape(1, -1))  # Get class probabilities
    max_prob = np.max(probs)                       # Highest class probability

    if max_prob >= confidence_threshold:
        high_confidence_count += 1
        KNN_anomalies_list.append(x)

robustness = high_confidence_count / len(scaled_anomalous_inputs)
KNN_anomalies_list = ANN_scaler.inverse_transform(KNN_anomalies_list)
print(robustness)
```

```
0.8636363636363636
```

In [34]:
```
len(KNN_anomalies_list)
```

Out[34]:
```
57
```

In [35]:
```
feature_names = ["sepal length (cm)", "sepal width (cm)", "petal length (cm)", "petal width (cm)"]
knn_df = pd.DataFrame(KNN_anomalies_list, columns = feature_names)
```

Checking Shared Vulnerabilities

In [36]:
```
import pandas as pd

# Drop duplicates in each DataFrame
df_rf = RF_df.drop_duplicates()
df_knn = knn_df.drop_duplicates()
df_nn = MLP_df.drop_duplicates()

# Merge on all columns to find common rows
common_rows = df_rf.merge(df_knn, how='inner').merge(df_nn, how='inner')

print("Number of common rows:", len(common_rows))
```

```
Number of common rows: 41
```

In [37]:
```
# Columns to compare
generated_df_copy = generated_df_anomalies.copy()
generated_df_copy.drop_duplicates().reset_index(drop=True)
cols = generated_df.columns[:4]  # First four columns

test_df1 = pd.merge(generated_df_copy[cols],knn_df,how="outer",on=feature_names,indicator=True)
test_df2 = pd.merge(generated_df_copy[cols],RF_df,how="outer",on=feature_names,indicator=True)
test_df3 = pd.merge(generated_df_copy[cols],MLP_df,how="outer",on=feature_names,indicator=True)

generated_df_copy['knn_overlap'] = test_df1['_merge']
generated_df_copy['RF_overlap'] = test_df2['_merge']
generated_df_copy['MLP_overlap'] = test_df3['_merge']

overlap_map = {"both": True, "left_only": False}
generated_df_copy["knn_overlap"] = generated_df_copy["knn_overlap"].map(overlap_map)
generated_df_copy['RF_overlap'] = generated_df_copy["RF_overlap"].map(overlap_map)
generated_df_copy['MLP_overlap'] = generated_df_copy["MLP_overlap"].map(overlap_map)

generated_df_copy["all_overlap"] = (
    (generated_df_copy["knn_overlap"] == True) &
```

```
    (generated_df_copy["RF_overlap"] == True) &
    (generated_df_copy["MLP_overlap"] == True)
)
```

In [38]:
```
generated_df_copy["all_overlap"].value_counts()
```

Out[38]:
```
all_overlap
True     41
False    25
Name: count, dtype: int64
```

Decision Boundaries - PCA

In [39]:
```
feature_names = ["sepal length (cm)", "sepal width (cm)", "petal length (cm)", "petal width (cm)"]

# Create DataFrame with original (unscaled) values
iris_df = pd.DataFrame(X, columns=feature_names)
iris_df["Class"] = y  # Numeric class labels (0, 1, 2)
iris_df["Source"] = "Original"  # Mark as original data
```

In [40]:
```
def plot_pca_decision_boundary_KNN(classifier, original_data, generated_data,features,model_name,scaler =
                                   ANN_scaler,resolution=0.02):
    pca_scaler = StandardScaler()
    original_scaled = pca_scaler.fit_transform(original_data[features])
    generated_scaled = pca_scaler.transform(generated_data[features])

    pca = PCA(n_components=2)
    original_pca = pca.fit_transform(original_scaled)
    generated_pca = pca.transform(generated_scaled)

    all_pca = np.vstack([original_pca, generated_pca])
    x_min, x_max = all_pca[:, 0].min() - 1, all_pca[:, 0].max() + 1
    y_min, y_max = all_pca[:, 1].min() - 1, all_pca[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, resolution),
                         np.arange(y_min, y_max, resolution))


    X_mesh_pca = np.c_[xx.ravel(), yy.ravel()]
    X_mesh_input_scaled = pca.inverse_transform(X_mesh_pca)
    X_mesh_input_unscaled = pca_scaler.inverse_transform(X_mesh_input_scaled)

    sample = scaler.transform(X_mesh_input_unscaled)


    zz = (classifier.predict(sample))
    zz = zz.reshape(xx.shape)


    #print("Unique predictions on mesh grid:", np.unique(zz))
    generated_df_copy.loc[generated_df_copy["knn_overlap"] == True, "Source"] = "Generated (Above Threshold)"
    generated_df_copy.loc[generated_df_copy["knn_overlap"] != True, "Source"] = "Generated (Below Threshold)"

    cmap = ListedColormap(["#e41a1c", "#377eb8", "#4daf4a"])  # Setosa, Versicolour, Virginica
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, zz, levels=np.arange(-0.5, 3.5, 1), cmap=cmap, alpha=0.3)

    class_map = {0: "Setosa", 1: "Versicolor", 2: "Virginica"}
    original_data["ClassName"] = original_data["Class"].map(class_map)
    generated_data["ClassName"] = generated_data["Class"].map(class_map)

    original_data["PCA1"] = original_pca[:, 0]
    original_data["PCA2"] = original_pca[:, 1]
    generated_data["PCA1"] = generated_pca[:, 0]
    generated_data["PCA2"] = generated_pca[:, 1]

    palette = {
        "Setosa": "#e41a1c",
        "Versicolor": "#377eb8",
        "Virginica": "#4daf4a"
    }

    combined_data = pd.concat([original_data, generated_data], ignore_index=True)

    sns.scatterplot(
        data=original_data,
        x="PCA1",
        y="PCA2",
        hue="ClassName",
        palette=palette,
        s=80,
        edgecolor="black",
        linewidth=0.7,
        alpha=0.7
```

```
    )

    palette = {
        "Generated (Above Threshold)": "#e4931a",
        "Generated (Below Threshold)": "#9d9e9f",
    }
    sns.scatterplot(
        data=generated_data,
        x="PCA1",
        y="PCA2",
        hue="Source",
        style="Source",
        palette= palette,
        markers= {"Generated (Above Threshold)" : "X","Generated (Below Threshold)" : 'D'},
        s=80,
        edgecolor="black",
        linewidth=0.7,
        alpha=0.7
    )


    plt.legend(title="Class/Generated Confidence", bbox_to_anchor=(1.05, 1), loc='upper left',fontsize="small")


    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")
    plt.title(f"Decision Boundaries in PCA Space with Original and Generated Data ({model_name})")
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

In [41]:

```
def plot_pca_decision_boundary_ANN(classifier, original_data, generated_data,
                                   features,ANN_scaler, model_name, resolution=0.02, device='cpu'):

    # Scale and apply PCA
    pca_scaler = StandardScaler()
    original_scaled = pca_scaler.fit_transform(original_data[features])
    generated_scaled = pca_scaler.transform(generated_data[features])

    pca = PCA(n_components=2)
    original_pca = pca.fit_transform(original_scaled)
    generated_pca = pca.transform(generated_scaled)

    # Mesh grid
    all_pca = np.vstack([original_pca, generated_pca])
    x_min, x_max = all_pca[:, 0].min() - 1, all_pca[:, 0].max() + 1
    y_min, y_max = all_pca[:, 1].min() - 1, all_pca[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, resolution),
                         np.arange(y_min, y_max, resolution))

    X_mesh_pca = np.c_[xx.ravel(), yy.ravel()]
    X_mesh_input = pca.inverse_transform(X_mesh_pca)
    X_mesh_input_unscaled = pca_scaler.inverse_transform(X_mesh_input)
    sample = ANN_scaler.transform(X_mesh_input_unscaled)

    # Convert to PyTorch tensor
    sample_tensor = torch.tensor(sample, dtype=torch.float32).to(device)

    # Inference
    classifier.eval()
    with torch.no_grad():
        outputs = classifier(sample_tensor)
        preds = torch.argmax(outputs, dim=1).cpu().numpy()

    zz = preds.reshape(xx.shape)

    # Annotate generated data
    generated_data.loc[generated_data["MLP_overlap"] == True, "Source"] = "Generated (Above Threshold)"
    generated_data.loc[generated_data["MLP_overlap"] != True, "Source"] = "Generated (Below Threshold)"

    # Plot decision boundary
    cmap = ListedColormap(["#e41a1c", "#377eb8", "#4daf4a"])
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, zz, levels=np.arange(-0.5, 3.5, 1), cmap=cmap, alpha=0.3)

    # Map class labels
    class_map = {0: "Setosa", 1: "Versicolor", 2: "Virginica"}
    original_data["ClassName"] = original_data["Class"].map(class_map)
    generated_data["ClassName"] = generated_data["Class"].map(class_map)

    original_data["PCA1"] = original_pca[:, 0]
    original_data["PCA2"] = original_pca[:, 1]
    generated_data["PCA1"] = generated_pca[:, 0]
    generated_data["PCA2"] = generated_pca[:, 1]

    # Plot original data
```

```
    palette = {
        "Setosa": "#e41a1c",
        "Versicolor": "#377eb8",
        "Virginica": "#4daf4a"
    }
    sns.scatterplot(
        data=original_data,
        x="PCA1",
        y="PCA2",
        hue="ClassName",
        palette=palette,
        s=80,
        edgecolor="black",
        linewidth=0.7,
        alpha=0.7
    )

    # Plot generated data
    palette = {
        "Generated (Above Threshold)": "#e4931a",
        "Generated (Below Threshold)": "#9d9e9f",
    }
    sns.scatterplot(
        data=generated_data,
        x="PCA1",
        y="PCA2",
        hue="Source",
        style="Source",
        palette=palette,
        markers={"Generated (Above Threshold)": "X", "Generated (Below Threshold)": 'D'},
        s=80,
        edgecolor="black",
        linewidth=0.7,
        alpha=0.7
    )

    plt.legend(title="Class/Generated Confidence", bbox_to_anchor=(1.05, 1), loc='upper left', fontsize="small")
    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")
    plt.title(f"Decision Boundaries with Original and Generated Data ({model_name})")
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

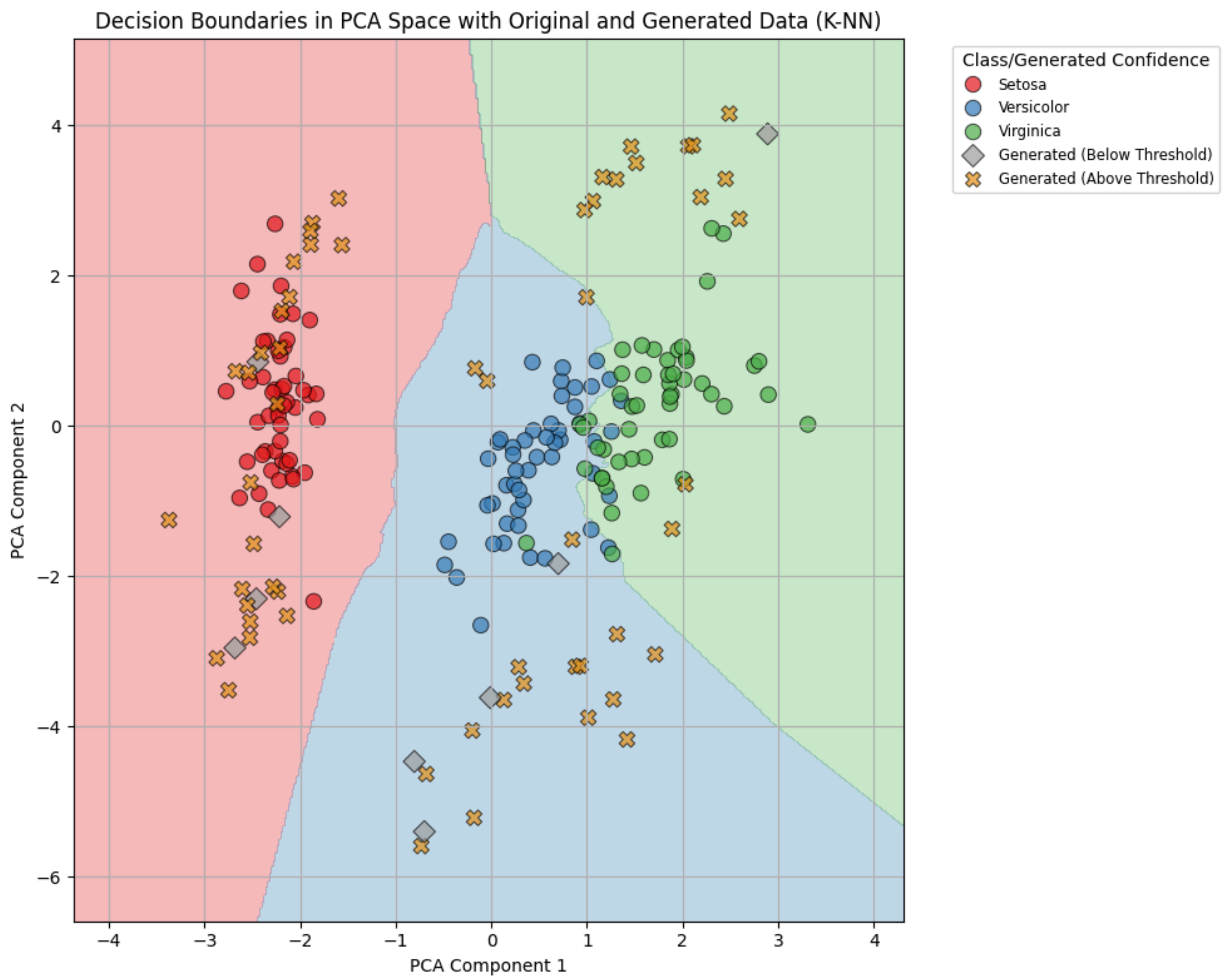
In [42]:
```
def plot_pca_decision_boundary_RF(classifier, original_data, generated_data,features,model_name
                              ,resolution=0.02):
    pca_scaler = StandardScaler()
    original_scaled = pca_scaler.fit_transform(original_data[features])
    generated_scaled = pca_scaler.transform(generated_data[features])

    pca = PCA(n_components=2)
    original_pca = pca.fit_transform(original_scaled)
    generated_pca = pca.transform(generated_scaled)

    all_pca = np.vstack([original_pca, generated_pca])
    x_min, x_max = all_pca[:, 0].min() - 1, all_pca[:, 0].max() + 1
    y_min, y_max = all_pca[:, 1].min() - 1, all_pca[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, resolution),
                         np.arange(y_min, y_max, resolution))


    X_mesh_pca = np.c_[xx.ravel(), yy.ravel()]
    X_mesh_input_scaled = pca.inverse_transform(X_mesh_pca)
    X_mesh_input_unscaled = pca_scaler.inverse_transform(X_mesh_input_scaled)

    zz = classifier.predict(X_mesh_input_unscaled)
    zz = zz.reshape(xx.shape)

    #print("Unique predictions on mesh grid:", np.unique(zz))
    generated_df_copy.loc[generated_df_copy["RF_overlap"] == True, "Source"] = "Generated (Above Threshold)"
    generated_df_copy.loc[generated_df_copy["RF_overlap"] != True, "Source"] = "Generated (Below Threshold)"

    cmap = ListedColormap(["#e41a1c", "#377eb8", "#4daf4a"])  # Setosa, Versicolour, Virginica
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, zz, levels=np.arange(-0.5, 3.5, 1), cmap=cmap, alpha=0.3)

    class_map = {0: "Setosa", 1: "Versicolor", 2: "Virginica"}
    original_data["ClassName"] = original_data["Class"].map(class_map)
    generated_data["ClassName"] = generated_data["Class"].map(class_map)

    original_data["PCA1"] = original_pca[:, 0]
    original_data["PCA2"] = original_pca[:, 1]
    generated_data["PCA1"] = generated_pca[:, 0]
    generated_data["PCA2"] = generated_pca[:, 1]
```

```
    palette = {
        "Setosa": "#e41a1c",
        "Versicolor": "#377eb8",
        "Virginica": "#4daf4a"
    }

    combined_data = pd.concat([original_data, generated_data], ignore_index=True)

    sns.scatterplot(
        data=original_data,
        x="PCA1",
        y="PCA2",
        hue="ClassName",
        palette=palette,
        s=80,
        edgecolor="black",
        linewidth=0.7,
        alpha=0.7
    )

    palette = {
        "Generated (Above Threshold)": "#e4931a",
        "Generated (Below Threshold)": "#9d9e9f",
    }
    sns.scatterplot(
        data=generated_data,
        x="PCA1",
        y="PCA2",
        hue="Source",
        style="Source",
        palette= palette,
        markers= {"Generated (Above Threshold)" : "X","Generated (Below Threshold)" : 'D'},
        s=80,
        edgecolor="black",
        linewidth=0.7,
        alpha=0.7
    )


    plt.legend(title="Class/Generated Confidence", bbox_to_anchor=(1.05, 1), loc='upper left',fontsize="small")


    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")
    plt.title(f"Decision Boundaries in PCA Space with Original and Generated Data ({model_name})")
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

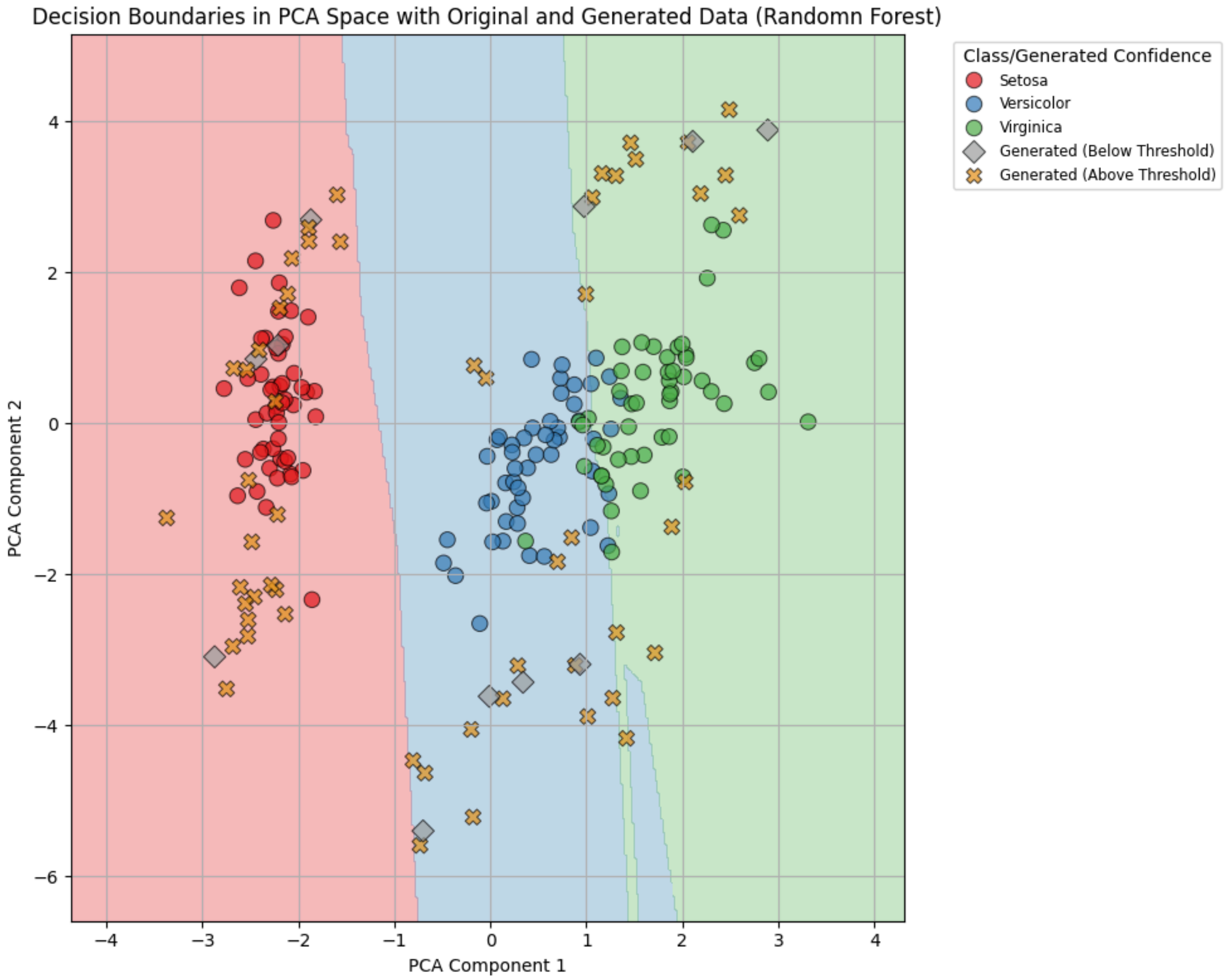
In [43]:
```
def plot_pca_decision_boundary_Invertible(classifier, original_data, generated_data, features,model_name,
                                           resolution=0.02):

    pca_scaler = StandardScaler()
    original_scaled = pca_scaler.fit_transform(original_data[features])
    generated_scaled = pca_scaler.transform(generated_data[features])

    pca = PCA(n_components=2)
    original_pca = pca.fit_transform(original_scaled)
    generated_pca = pca.transform(generated_scaled)

    all_pca = np.vstack([original_pca, generated_pca])
    x_min, x_max = all_pca[:, 0].min() - 1, all_pca[:, 0].max() + 1
    y_min, y_max = all_pca[:, 1].min() - 1, all_pca[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, resolution),
                         np.arange(y_min, y_max, resolution))


    X_mesh_pca = np.c_[xx.ravel(), yy.ravel()]
    X_mesh_input = pca.inverse_transform(X_mesh_pca)
    X_mesh_input_unscaled = pca_scaler.inverse_transform(X_mesh_input)
    sample = ANN_scaler.transform(X_mesh_input_unscaled)

    zz = np.array([
        np.argmax(classifier.forward(x.reshape(-1, 1))[:3]) for x in sample
    ])

    zz = zz.reshape(xx.shape)

    #print("Unique predictions on mesh grid:", np.unique(zz))
    generated_df_copy.loc[generated_df_copy["MLP_overlap"] == True, "Source"] = "Generated (Above Threshold)"
    generated_df_copy.loc[generated_df_copy["MLP_overlap"] != True, "Source"] = "Generated (Below Threshold)"

    cmap = ListedColormap(["#e41a1c", "#377eb8", "#4daf4a"])  # Setosa, Versicolour, Virginica
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, zz, levels=np.arange(-0.5, 3.5, 1), cmap=cmap, alpha=0.3)
```

```
    class_map = {0: "Setosa", 1: "Versicolor", 2: "Virginica"}
    original_data["ClassName"] = original_data["Class"].map(class_map)
    generated_data["ClassName"] = generated_data["Class"].map(class_map)

    original_data["PCA1"] = original_pca[:, 0]
    original_data["PCA2"] = original_pca[:, 1]
    generated_data["PCA1"] = generated_pca[:, 0]
    generated_data["PCA2"] = generated_pca[:, 1]

    palette = {
        "Setosa": "#e41a1c",
        "Versicolor": "#377eb8",
        "Virginica": "#4daf4a"
    }

    combined_data = pd.concat([original_data, generated_data], ignore_index=True)

    sns.scatterplot(
        data=original_data,
        x="PCA1",
        y="PCA2",
        hue="ClassName",
        palette=palette,
        s=80,
        edgecolor="black",
        linewidth=0.7,
        alpha=0.7
    )


    sns.scatterplot(
        data=generated_data,
        x="PCA1",
        y="PCA2",
        color=  "#e4931a",
        marker=  "X",
        s=80,
        edgecolor="black",
        linewidth=0.7,
        alpha=0.7
    )


    plt.legend(title="Class/Generated Confidence", bbox_to_anchor=(1.05, 1), loc='upper left',fontsize="small")


    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")
    plt.title(f"Decision Boundaries in PCA Space with Original and Generated Data ({model_name})")
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

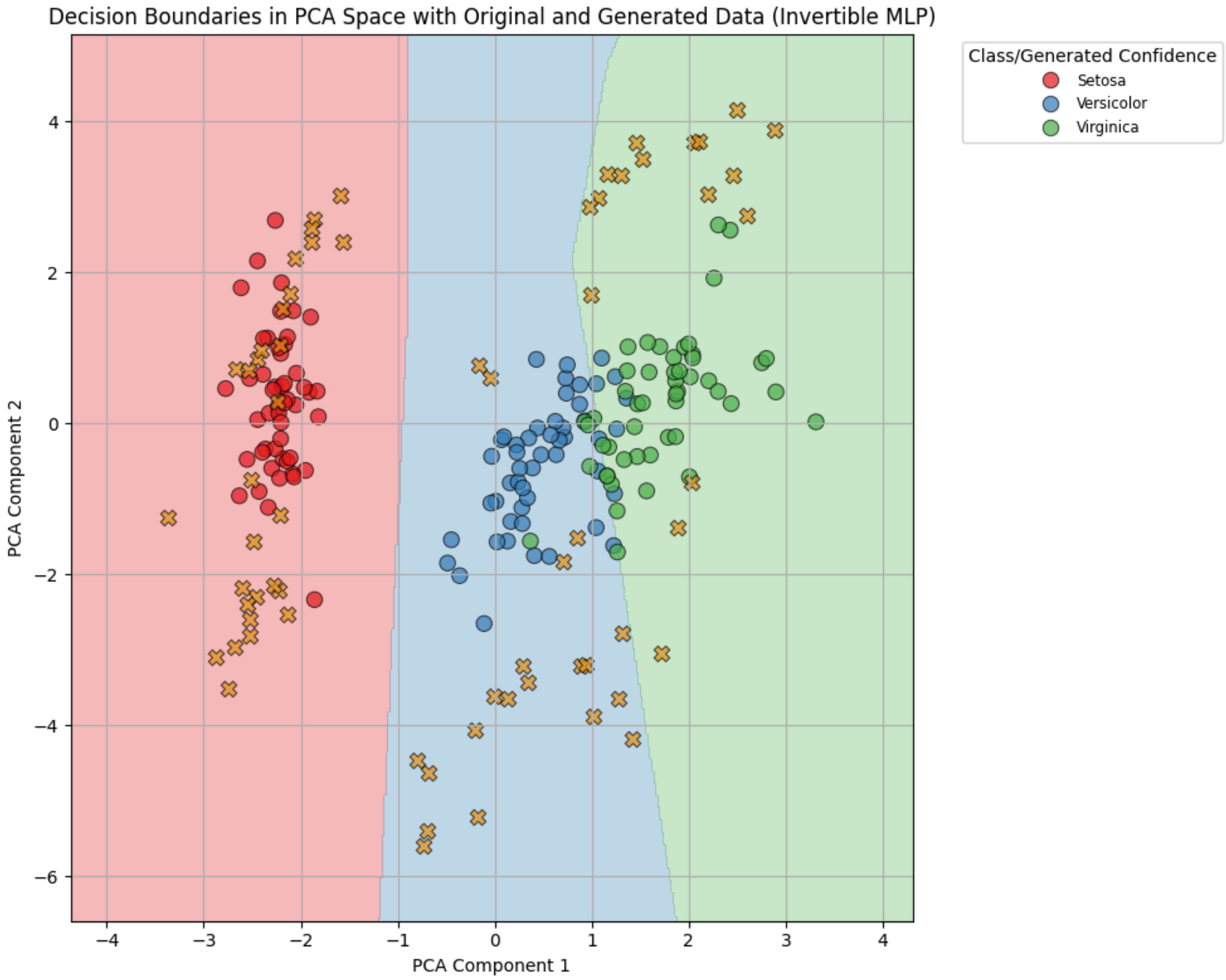
In [44]:
```
plot_pca_decision_boundary_KNN(neigh, iris_df, generated_df_copy,feature_names,model_name= "K-NN")
```



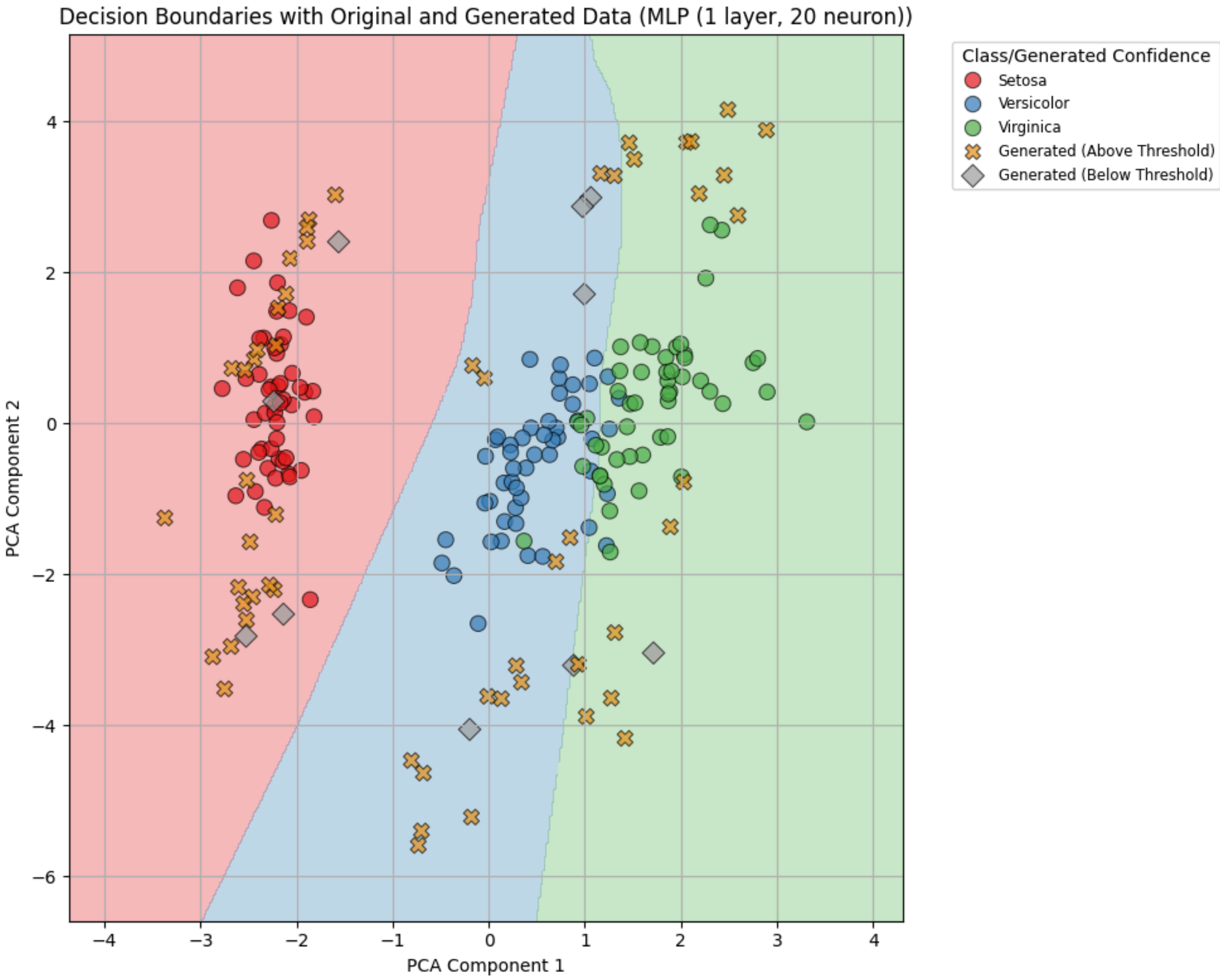
In [45]:
```
plot_pca_decision_boundary_RF(RF_model, iris_df, generated_df_copy,feature_names,model_name = "Randomn Forest")
```



In [46]:
```
plot_pca_decision_boundary_Invertible(model, iris_df, generated_df_copy,feature_names,model_name = "Invertible MLP")
```



In [47]:
```
plot_pca_decision_boundary_ANN(test_model, iris_df, generated_df_copy,feature_names,
                               ANN_scaler,model_name = "MLP (1 layer, 20 neuron)")
```



In [48]:

```
original_data = iris_df.copy()
generated_data = generated_df_copy.copy()

pca_scaler = StandardScaler()
original_scaled = pca_scaler.fit_transform(original_data[feature_names])
generated_scaled = pca_scaler.transform(generated_data[feature_names])

# Fit PCA only on scaled original data
pca = PCA(n_components=2) # Only use two principal components
pca.fit(original_scaled)

# Project both datasets using same PCA
original_pca = pca.transform(original_scaled)
generated_pca = pca.transform(generated_scaled)

# Add PCA results to DataFrames
original_data["PCA1"] = original_pca[:, 0]
original_data["PCA2"] = original_pca[:, 1]
generated_data["PCA1"] = generated_pca[:, 0]
generated_data["PCA2"] = generated_pca[:, 1]
generated_data.loc[generated_df_copy["all_overlap"] == True, "Source"] = "Generated Shared"
generated_data.loc[generated_df_copy["all_overlap"] != True, "Source"] = "Generated Not Shared"
# Plot
plt.figure(figsize=(10, 7))
palette = {
    "Setosa": "#e41a1c",
    "Versicolor": "#377eb8",
    "Virginica": "#4daf4a"
}
combined_data = pd.concat([original_data, generated_data], ignore_index=True)
# Create the plot

sns.scatterplot(
    data=original_data,
    x="PCA1",
    y="PCA2",
    hue="ClassName",
    palette=palette,
    s=80,
    edgecolor="black",
    linewidth=0.7,
    alpha=0.7
)
```

```
palette = {
    "Generated Shared": "#e4931a",
    "Generated Not Shared": "#9d9e9f",
}
sns.scatterplot(
    data=generated_data,
    x="PCA1",
    y="PCA2",
    hue="Source",
    style="Source",
    palette= palette,
    markers= {"Generated Shared" : "X","Generated Not Shared" : 'D'},
    s=80,
    edgecolor="black",
    linewidth=0.7,
    alpha=0.7
)

plt.legend(title="Class / Source", bbox_to_anchor=(1.05, 1), loc='upper left')

plt.title("PCA Projection on Scaled Features (Fitted on Original Data Only)", fontsize=14)
plt.grid(True)
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.tight_layout()
plt.show()
```