In [1]:
```python
#Importing the required libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import sklearn.metrics
from sklearn.metrics import precision_score, recall_score, f1_score, confus
```

```
C:\Users\Dell\anaconda3\lib\site-packages\scipy\__init__.py:155: UserWarni
ng: A NumPy version >=1.18.5 and <1.25.0 is required for this version of S
ciPy (detected version 1.26.1
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

In [2]:
```python
# Loading Hadoop log file dataset into a Pandas DataFrame
df = pd.read_csv('Hadoop_log.csv')
df.head()
```

Out[2]:

| | LineId | Date | Time | Level | Process | Component |
|---|---|---|---|---|---|---|
| **0** | 1 | 18-10-2015 | 18:01:47,978 | INFO | main | org.apache.hadoop.mapreduce.v2.app.MRAppMaster |
| **1** | 2 | 18-10-2015 | 18:01:48,963 | INFO | main | org.apache.hadoop.mapreduce.v2.app.MRAppMaster |
| **2** | 3 | 18-10-2015 | 18:01:48,963 | INFO | main | org.apache.hadoop.mapreduce.v2.app.MRAppMaster |
| **3** | 4 | 18-10-2015 | 18:01:49,228 | INFO | main | org.apache.hadoop.mapreduce.v2.app.MRAppMaster |
| **4** | 5 | 18-10-2015 | 18:01:50,353 | INFO | main | org.apache.hadoop.mapreduce.v2.app.MRAppMaster |

In [3]:
```python
df.shape
```

Out[3]:
```
(2000, 9)
```

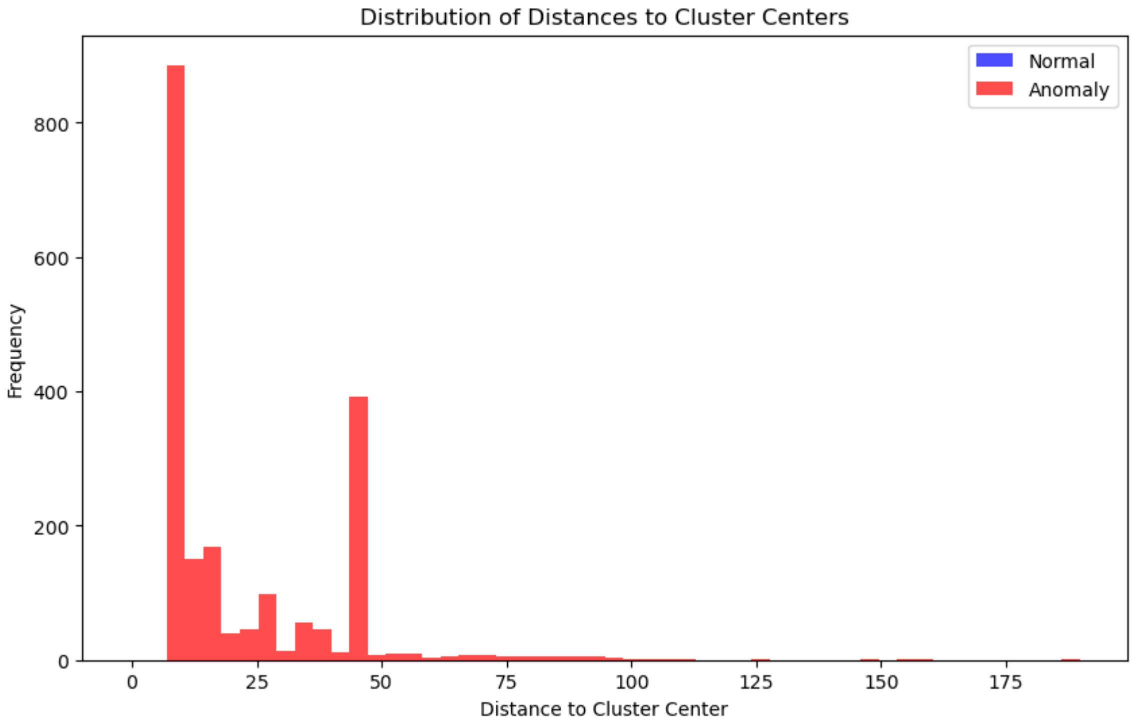
In [4]:
```
# Extract relevant text data for TF-IDF vectorization
text_data = df['Content'].tolist()

# TF-IDF vectorization
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(text_data)

# Convert the TF-IDF matrix to a DataFrame for better visualization
tfidf_df = pd.DataFrame(data=tfidf_matrix.toarray(), columns=tfidf_vectoriz
# Standardize the TF-IDF features
scaler = StandardScaler()
tfidf_scaled = scaler.fit_transform(tfidf_df)
print("Scaled TF-ID:\n", tfidf_scaled)
```

```
Scaled TF-ID:
 [[-2.23662720e-02 -4.43807543e-02 -2.23662720e-02 ... -2.23662720e-02
  -5.79118249e-02 -2.23662720e-02]
 [-2.23662720e-02 -4.43807543e-02 -2.23662720e-02 ... -2.23662720e-02
  -5.79118249e-02 -2.23662720e-02]
 [-2.23662720e-02 -4.43807543e-02 -2.23662720e-02 ... -2.23662720e-02
  -5.79118249e-02  4.47101778e+01]
 ...
 [-2.23662720e-02 -4.43807543e-02 -2.23662720e-02 ... -2.23662720e-02
  -5.79118249e-02 -2.23662720e-02]
 [-2.23662720e-02 -4.43807543e-02 -2.23662720e-02 ... -2.23662720e-02
  -5.79118249e-02 -2.23662720e-02]
 [-2.23662720e-02 -4.43807543e-02 -2.23662720e-02 ... -2.23662720e-02
  -5.79118249e-02 -2.23662720e-02]]
```

In [5]:
```python
# Choose the number of clusters (we may need to adjust this based on ourdat
num_clusters = 3

# Perform k-means clustering on the TF-IDF features
model = KMeans(n_clusters=num_clusters, random_state=42)
clusters = model.fit_predict(tfidf_scaled)

# Add the cluster labels to the original DataFrame
df['cluster'] = clusters

# Calculate the distance of each instance to its cluster center
distances = model.transform(tfidf_scaled)
df['distance_to_cluster'] = distances.min(axis=1)

# Set a threshold for anomaly detection (adjust this based on your data)
anomaly_threshold = 2.0

# Mark instances with distance exceeding the threshold as anomalies
df['is_anomaly'] = df['distance_to_cluster'] > anomaly_threshold

# Print the instances marked as anomalies
anomalies = df[df['is_anomaly']]
print("Anomalies:")
print(anomalies[['LineId','distance_to_cluster']])
```

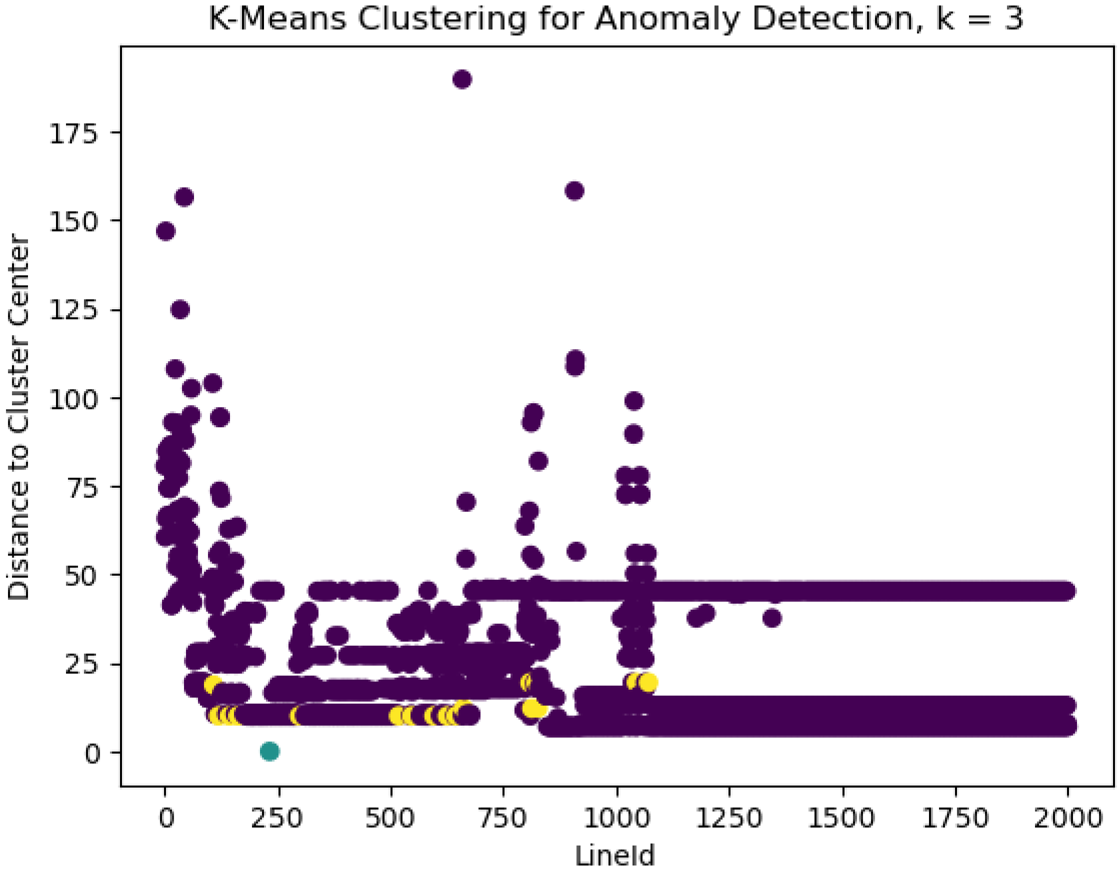
```
Anomalies:
      LineId  distance_to_cluster
0          1            80.648760
1          2            60.558590
2          3           146.858057
3          4            65.791646
4          5            84.804843
...      ...                  ...
1995    1996             6.996495
1996    1997            45.327590
1997    1998            12.977065
1998    1999             7.855210
1999    2000             6.996495

[1999 rows x 2 columns]
```

In [6]:
```
# Visualization of distances for normal instances
plt.figure(figsize=(10, 6))
plt.hist(df[df['is_anomaly'] == False]['distance_to_cluster'], bins=50, colo
plt.hist(df[df['is_anomaly'] == True]['distance_to_cluster'], bins=50, colo
plt.title('Distribution of Distances to Cluster Centers')
plt.xlabel('Distance to Cluster Center')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```



In [7]:

```
# Visualize the clustering
plt.scatter(df['LineId'], df['distance_to_cluster'], c=df['cluster'], cmap=
plt.title('K-Means Clustering for Anomaly Detection, k = 3')
plt.xlabel('LineId')
plt.ylabel('Distance to Cluster Center')
plt.show()
```

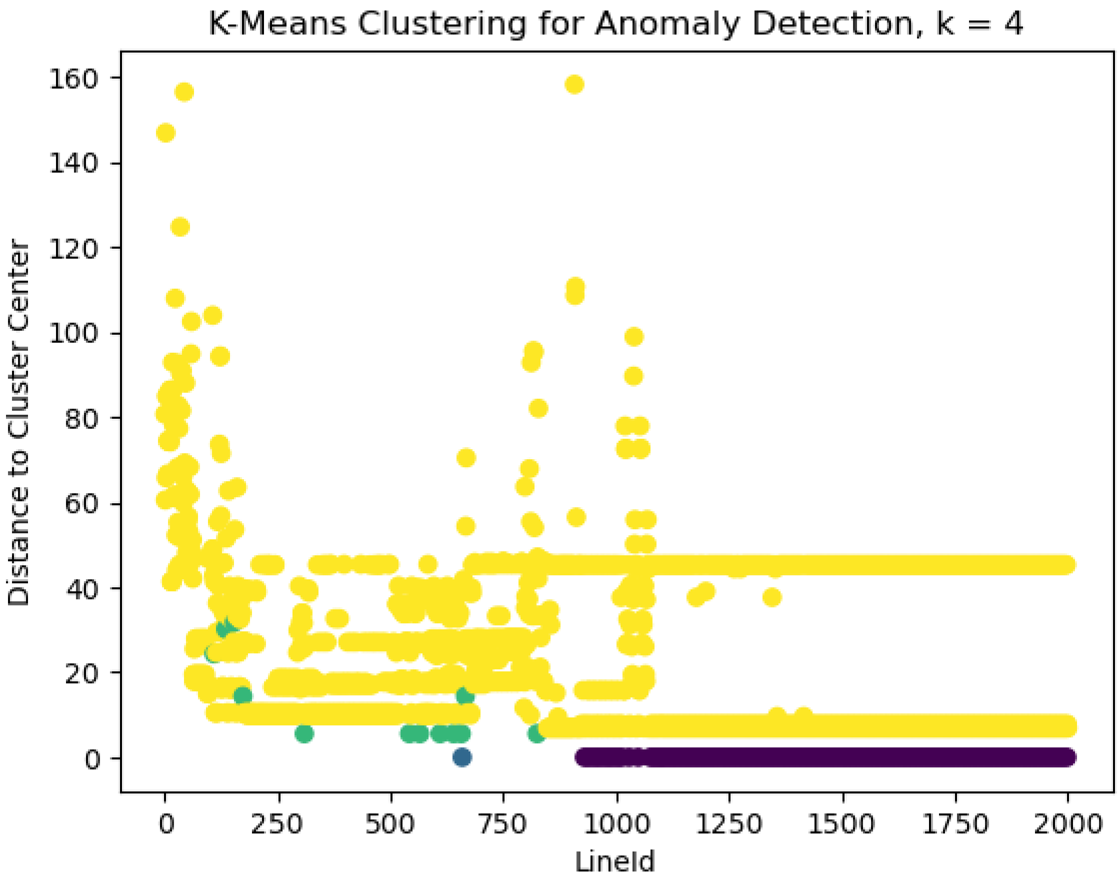


In [8]:
```python
# Choose the number of clusters (we may need to adjust this based on ourdata
num_clusters = 4

# Perform k-means clustering on the TF-IDF features
model = KMeans(n_clusters=num_clusters, random_state=42)
clusters = model.fit_predict(tfidf_scaled)

# Add the cluster labels to the original DataFrame
df['cluster'] = clusters

# Calculate the distance of each instance to its cluster center
distances = model.transform(tfidf_scaled)
df['distance_to_cluster'] = distances.min(axis=1)

# Set a threshold for anomaly detection (adjust this based on your data)
anomaly_threshold = 2.0

# Mark instances with distance exceeding the threshold as anomalies
df['is_anomaly'] = df['distance_to_cluster'] > anomaly_threshold

# Print the instances marked as anomalies
anomalies = df[df['is_anomaly']]
print("Anomalies:")
print(anomalies[['LineId','distance_to_cluster']])
```

```
Anomalies:
      LineId  distance_to_cluster
0          1            80.642022
1          2            60.551533
2          3           146.857826
3          4            65.785110
4          5            84.798741
...      ...                  ...
1994    1995             7.276481
1995    1996             6.898374
1996    1997            45.318255
1998    1999             7.791410
1999    2000             6.898374

[1853 rows x 2 columns]
```

In [9]:

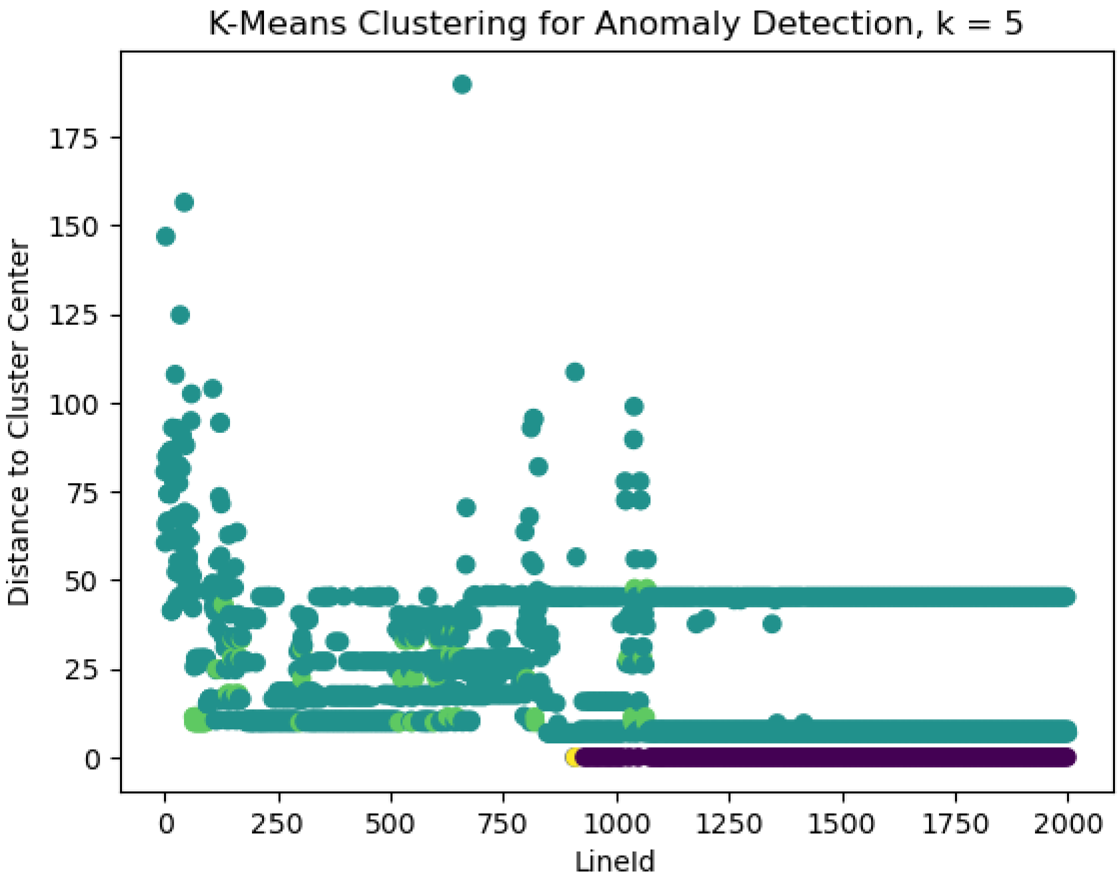
```
# Visualize the clustering
plt.scatter(df['LineId'], df['distance_to_cluster'], c=df['cluster'], cmap=
plt.title('K-Means Clustering for Anomaly Detection, k = 4')
plt.xlabel('LineId')
plt.ylabel('Distance to Cluster Center')
plt.show()
```



In [10]:
```python
# Choose the number of clusters (we may need to adjust this based on ourdat
num_clusters = 5

# Perform k-means clustering on the TF-IDF features
model = KMeans(n_clusters=num_clusters, random_state=42)
clusters = model.fit_predict(tfidf_scaled)

# Add the cluster labels to the original DataFrame
df['cluster'] = clusters

# Calculate the distance of each instance to its cluster center
distances = model.transform(tfidf_scaled)
df['distance_to_cluster'] = distances.min(axis=1)

# Set a threshold for anomaly detection (adjust this based on your data)
anomaly_threshold = 2.0

# Mark instances with distance exceeding the threshold as anomalies
df['is_anomaly'] = df['distance_to_cluster'] > anomaly_threshold

# Print the instances marked as anomalies
anomalies = df[df['is_anomaly']]
print("Anomalies:")
print(anomalies[['LineId','distance_to_cluster']])
```

```
Anomalies:
      LineId  distance_to_cluster
0          1            80.641628
1          2            60.551952
2          3           146.848434
3          4            65.803408
4          5            84.798606
...      ...                  ...
1994    1995             7.244981
1995    1996             6.863443
1996    1997            45.314075
1998    1999             7.791430
1999    2000             6.863443

[1852 rows x 2 columns]
```

In [11]:
```
# Visualize the clustering
plt.scatter(df['LineId'], df['distance_to_cluster'], c=df['cluster'], cmap=
plt.title('K-Means Clustering for Anomaly Detection, k = 5')
plt.xlabel('LineId')
plt.ylabel('Distance to Cluster Center')
plt.show()
```




In [12]:
```
true_labels = df['is_anomaly']

# Predicted labels based on the given threshold
predicted_labels = df['distance_to_cluster'] > anomaly_threshold

# Calculate metrics
precision = precision_score(true_labels, predicted_labels)
recall = recall_score(true_labels, predicted_labels)
f1 = f1_score(true_labels, predicted_labels)
print("Precision:", precision)
print("Recall:",recall)
print("F1_Score:",f1)
roc_auc = roc_auc_score(true_labels, df['distance_to_cluster'])
print("Roc Auc:",roc_auc)
conf_matrix = confusion_matrix(true_labels, predicted_labels)
print("Confusion Matrix:\n",conf_matrix)
```

```
Precision: 1.0
Recall: 1.0
F1_Score: 1.0
Roc Auc: 1.0
Confusion Matrix:
 [[ 148    0]
 [   0 1852]]
```

In [13]:

```
log_data = pd.read_csv("Hadoop_log.csv")
log_entries = log_data['Content'].values
# Convert log entries to numerical features
# TF-IDF transformation
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer()
tfidf_features = tfidf_vectorizer.fit_transform(log_entries)

# Standardize the features
scaler = StandardScaler()
scaled_features = scaler.fit_transform(tfidf_features.toarray())

# Train/test split
X_train, X_test = train_test_split(scaled_features, test_size=0.2, random_s

# Define the autoencoder model
model = Sequential()

# Encoder
model.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dense(32, activation='relu'))

# Decoder
model.add(Dense(64, activation='relu'))
model.add(Dense(X_train.shape[1], activation='linear'))  # Output layer wit

# Compile the model
model.compile(optimizer='adam', loss='mse')  # Mean Squared Error loss for 

# Train the model
model.fit(X_train, X_train, epochs=10, batch_size=32, validation_data=(X_te

# Make predictions on the test set
predictions = model.predict(X_test)
```

```
Epoch 1/10
50/50 [==============================] - 3s 23ms/step - loss: 1.0013 - val
_loss: 0.9941
Epoch 2/10
50/50 [==============================] - 0s 9ms/step - loss: 0.9676 - val_
loss: 0.9459
Epoch 3/10
50/50 [==============================] - 0s 9ms/step - loss: 0.9210 - val_
loss: 0.9097
Epoch 4/10
50/50 [==============================] - 0s 9ms/step - loss: 0.8878 - val_
loss: 0.8856
Epoch 5/10
50/50 [==============================] - 0s 9ms/step - loss: 0.8608 - val_
loss: 0.8764
Epoch 6/10
50/50 [==============================] - 0s 9ms/step - loss: 0.8406 - val_
loss: 0.8719
Epoch 7/10
50/50 [==============================] - 0s 10ms/step - loss: 0.8207 - val
_loss: 0.8679
Epoch 8/10
50/50 [==============================] - 0s 10ms/step - loss: 0.8091 - val
_loss: 0.8690
Epoch 9/10
50/50 [==============================] - 1s 11ms/step - loss: 0.7974 - val
_loss: 0.8667
Epoch 10/10
50/50 [==============================] - 1s 11ms/step - loss: 0.7823 - val
_loss: 0.8689
13/13 [==============================] - 0s 5ms/step
```

In [14]:

```
# Calculate reconstruction error (MSE)
mse = np.mean(np.square(X_test - predictions), axis=1)
# Choose a quantile as the threshold (e.g., 95th percentile)
threshold = np.percentile(mse, 95)
print("Threshold:",threshold)
```

```
Threshold: 2.2187053820065796
```

In [15]:

```
# Set a threshold for anomaly detection based on the reconstruction error
threshold = 2.3  # Adjust this based on your data and experimentation

# Identify anomalies
anomalies = mse > threshold

# Print or visualize the anomalies
print("Number of anomalies:", np.sum(anomalies))
print("Anomaly indices:", np.where(anomalies)[0])
```

```
Number of anomalies: 14
Anomaly indices: [ 10  29  33  40  54 112 184 194 209 285 296 309 320 354]
```

In [16]:

```
# Choose a threshold (e.g., 75th percentile)
threshold = np.percentile(mse, 75)
print("Threshold:", threshold)
```

```
Threshold: 1.354248300972447
```

In [17]:
```
# Set a threshold for anomaly detection based on the reconstruction error
threshold = 1.5  # Adjust this based on your data and experimentation

# Identify anomalies
anomalies = mse > threshold

# Print or visualize the anomalies
print("Number of anomalies:", np.sum(anomalies))
print("Anomaly indices:", np.where(anomalies)[0])
```

```
Number of anomalies: 96
Anomaly indices: [  0   4  10  12  16  18  21  28  29  31  33  38  40  49
52  54  55  56
  83  94  97 103 111 112 117 120 122 126 127 129 134 135 137 138 143 144
 165 168 175 176 178 182 183 184 185 192 193 194 207 209 214 215 217 226
 231 236 237 240 241 243 244 246 247 249 252 259 262 264 265 272 274 280
 285 292 296 302 305 309 311 319 320 328 331 336 348 349 351 352 354 362
 366 373 378 379 383 392]
```

In [18]:
```
# Assuming 'X_train' and 'X_test' are your input data
predictions_train = model.predict(X_train)
mse_train = np.mean(np.square(X_train - predictions_train), axis=1)

predictions_test = model.predict(X_test)
mse_test = np.mean(np.square(X_test - predictions_test), axis=1)
```

```
50/50 [==============================] - 0s 5ms/step
13/13 [==============================] - 0s 4ms/step
```

In [19]:
```
# Assuming true_labels are the ground truth labels for anomalies
# Make sure true_labels and anomalies have the same length
min_length = min(len(true_labels), len(anomalies))
true_labels = true_labels[:min_length]
anomalies = anomalies[:min_length]

precision = precision_score(true_labels, anomalies)
recall = recall_score(true_labels, anomalies)
f1 = f1_score(true_labels, anomalies)

print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Precision: 1.0
Recall: 0.24
F1 Score: 0.3870967741935484
```

In [20]:
```
# Calculate overall MSE on the test set
overall_mse = np.mean(np.square(X_test - predictions))

print("Overall MSE on Test Set:", overall_mse)
```

```
Overall MSE on Test Set: 0.8689211011124554
```

In [21]:

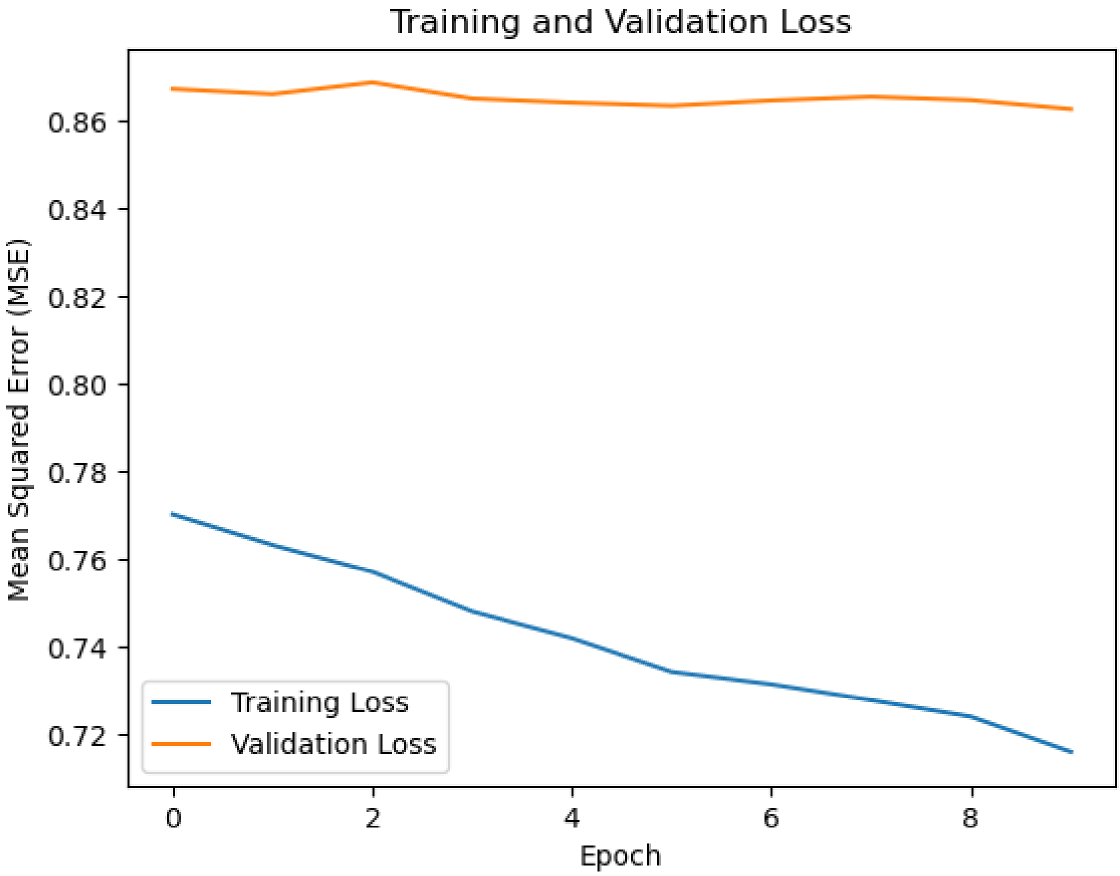
```
import random

# Choose random samples from the test set
num_samples_to_visualize = 5
random_indices = random.sample(range(len(X_test)), num_samples_to_visualize

# Visualize original and reconstructed samples
for index in random_indices:
    original_sample = X_test[index]
    reconstructed_sample = predictions[index]

    # Compare the original and reconstructed samples visually or using any c
    print("Original:", original_sample)
    print("Reconstructed:", reconstructed_sample)
    print("------------------------")
```

```
Original: [-0.02236627 -0.04438075 -0.02236627 -0.02236627 -0.02236627 -
0.02236627
  1.57080777 -0.02236627 -0.2806219  -0.02236627 -0.02236627 -0.02236627
 -0.06093248 -0.03802596 -0.02236627 -0.02236627 -0.02236627 -0.02236627
 -0.02236627 -0.02236627 -0.07088812 -0.03875891 -0.06337243 -0.0592646
 -0.03875891 -0.03875891 -0.03875891 -0.05006262 -0.03875891 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.05926376 -0.02236627 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.03972693 -0.02236627 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.04757794 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627
 -0.02236627 -0.02236627 -0.03052958 -0.02236627 -0.07088529 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.03052958 -0.02236627
 -0.02649157 -0.02236627 -0.02236627 -0.02236627 -0.02649157 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627
 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627 -0.02236627
 -0.02236627 -0.02236627 -0.03972693 -0.03802596  1.79385735 -0.02236627
```

In [22]:

```python
# Access training history
history = model.fit(X_train, X_train, epochs=10, batch_size=32, validation_

# Plot training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Error (MSE)')
plt.legend()
plt.show()
```

```
Epoch 1/10
50/50 [==============================] - 1s 12ms/step - loss: 0.7702 - val
_loss: 0.8672
Epoch 2/10
50/50 [==============================] - 0s 9ms/step - loss: 0.7632 - val_
loss: 0.8660
Epoch 3/10
50/50 [==============================] - 0s 9ms/step - loss: 0.7570 - val_
loss: 0.8687
Epoch 4/10
50/50 [==============================] - 1s 10ms/step - loss: 0.7480 - val
_loss: 0.8650
Epoch 5/10
50/50 [==============================] - 0s 9ms/step - loss: 0.7419 - val_
loss: 0.8641
Epoch 6/10
50/50 [==============================] - 0s 9ms/step - loss: 0.7342 - val_
loss: 0.8634
Epoch 7/10
50/50 [==============================] - 1s 11ms/step - loss: 0.7314 - val
_loss: 0.8646
Epoch 8/10
50/50 [==============================] - 1s 10ms/step - loss: 0.7277 - val
_loss: 0.8654
Epoch 9/10
50/50 [==============================] - 0s 10ms/step - loss: 0.7240 - val
_loss: 0.8647
Epoch 10/10
50/50 [==============================] - 0s 10ms/step - loss: 0.7159 - val
_loss: 0.8626
```


Training and Validation Loss
Mean Squared Error (MSE)
0.86
0.84
0.82
0.80
0.78
0.76
0.74
0.72
0
2
4
6
8
Epoch
Training Loss
Validation Loss

In [23]:
```
# Choose random samples from the test set
num_samples_to_visualize = 5
random_indices = np.random.choice(len(X_test), num_samples_to_visualize, re

# Visualize original and reconstructed samples
for index in random_indices:
    original_sample = X_test[index].reshape(tfidf_features.shape[1],)
    reconstructed_sample = predictions[index]

    # Plot the original and reconstructed samples
    plt.figure(figsize=(8, 4))
    plt.plot(original_sample, label='Original', color='blue')
    plt.plot(reconstructed_sample, label='Reconstructed', color='red', line
    plt.title('Autoencoder Reconstruction')
    plt.legend()
    plt.show()
```