Step 1 — Import Libraries

Goal:

Import all necessary libraries for PCA-based MNIST digit recognition.

In [32]:
```
import warnings
warnings.filterwarnings('ignore')

from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Builtin Models
from sklearn.datasets import fetch_openml
from collections import Counter
from sklearn.decomposition import PCA
from sklearn.metrics import confusion_matrix, classification_report, accuracy_
```

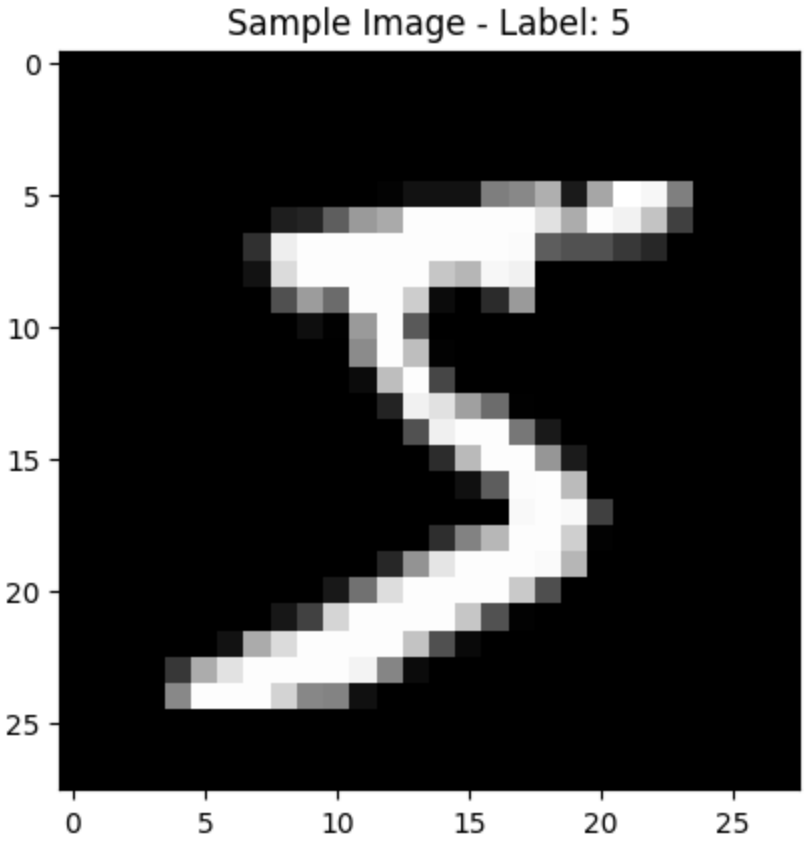
Step 2 — Load MNIST Dataset

Goal:

Load the MNIST dataset, inspect its shape, and display a sample image.

In [27]:
```
# Load MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist["data"], mnist["target"].astype(int)  # X: images, y: labels

print("Dataset shape (samples, features):", X.shape)
print("Number of classes:", len(np.unique(y)))

# Normalize pixel values to range [0,1]
#X = X / 255.0

# Display a sample image
plt.imshow(X[0].reshape(28,28), cmap='gray')
plt.title(f"Sample Image - Label: {y[0]}")
plt.show()
```

```
Dataset shape (samples, features): (70000, 784)
Number of classes: 10
```



Step 3 — Train-Test Split

Goal:
Split the dataset into **70% training** and **30% testing** sets, keeping the classes balanced.

In [28]:
```
# Balanced train-test split (70%-30%)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42
)

# Check class distribution
print("Class distribution in train set:", Counter(y_train))
print("Class distribution in test set:", Counter(y_test))

# Output shapes
print("\nTrain shape:", X_train.shape)
print("Test shape:", X_test.shape)
```

```
Class distribution in train set: Counter({np.int64(1): 5514, np.int64(7): 5105,
np.int64(3): 4999, np.int64(2): 4893, np.int64(9): 4871, np.int64(0): 4832, n
p.int64(6): 4813, np.int64(8): 4777, np.int64(4): 4777, np.int64(5): 4419})
Class distribution in test set: Counter({np.int64(1): 2363, np.int64(7): 2188,
np.int64(3): 2142, np.int64(2): 2097, np.int64(9): 2087, np.int64(0): 2071, n
p.int64(6): 2063, np.int64(8): 2048, np.int64(4): 2047, np.int64(5): 1894})

Train shape: (49000, 784)
Test shape: (21000, 784)
```

Step 4 — Hyperparameter Tuning (K_components)

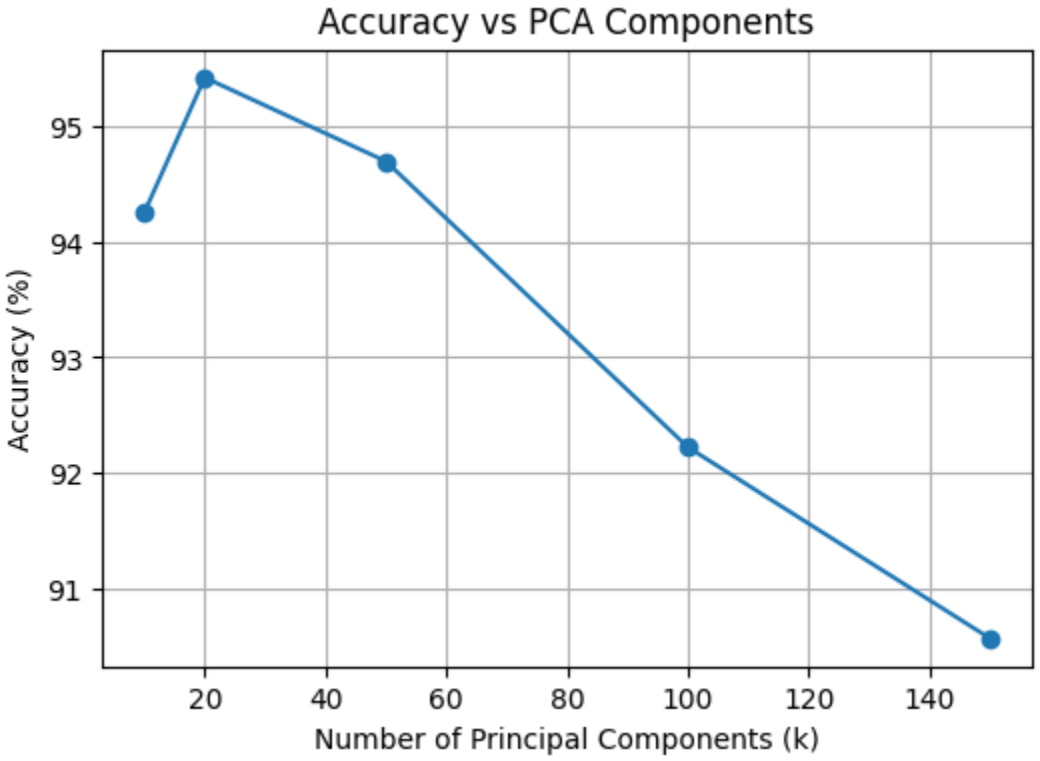
Goal:

- We try for different values and of k components and select the one which give us the highest accuracy.

In [33]:
```python
from sklearn.metrics import accuracy_score

# List of k_components to test
k_list = [10, 20, 50, 100, 150]
accuracy_list = []

for k in k_list:
    # Build PCA subspaces for each class
    class_pca = {}
    for digit in np.unique(y_train):
        X_digit = X_train[y_train == digit]
        mean_vector = np.mean(X_digit, axis=0)
        X_centered = X_digit - mean_vector
        pca = PCA(n_components=k)
        pca.fit(X_centered)
        class_pca[digit] = {'mean': mean_vector, 'pca': pca}

    # Classify test set
    y_test_pred = []
    for i in range(X_test.shape[0]):
        x = X_test[i]
        errors = [np.linalg.norm(x - (class_pca[d]['pca'].inverse_transform(
                    class_pca[d]['pca'].transform((x - class_pca[d]['mean']).r
                ) + class_pca[d]['mean'])) for d in class_pca]
        y_test_pred.append(np.argmin(errors))

    y_test_pred = np.array(y_test_pred)

    # Compute accuracy
    acc = accuracy_score(y_test, y_test_pred)
    accuracy_list.append(acc)
    print(f"k_components={k}, Accuracy={acc*100:.2f}%")
```

```python
# Plot accuracy vs k_components
plt.figure(figsize=(6,4))
plt.plot(k_list, np.array(accuracy_list)*100, marker='o')
plt.xlabel("Number of Principal Components (k)")
plt.ylabel("Accuracy (%)")
plt.title("Accuracy vs PCA Components")
plt.grid(True)
plt.show()
```

```
k_components=10, Accuracy=94.25%
k_components=20, Accuracy=95.42%
k_components=50, Accuracy=94.70%
k_components=100, Accuracy=92.22%
k_components=150, Accuracy=90.57%
```



Step 5 — Compute PCA Subspaces for Each Digit Class

Goal:
For each digit (0–9), build a PCA subspace to represent the class.

Tasks:

1. For each digit class:
    - Select all training images of that class

- Compute the mean image
- Apply PCA to get top `k` principal components

2. Store the mean and principal components for each class for later reconstruction

In [34]:
```
# Number of principal components per class
k_components = 20  # can be tuned

# Dictionary to store PCA model and mean for each class
class_pca = {}

for digit in np.unique(y_train):
    # Select all training images of this class
    X_digit = X_train[y_train == digit]

    # Compute the mean image
    mean_vector = np.mean(X_digit, axis=0)

    # Center the data
    X_centered = X_digit - mean_vector

    # Fit PCA on centered data
    pca = PCA(n_components=k_components)
    pca.fit(X_centered)

    # Store mean and PCA model for this class
    class_pca[digit] = {
        'mean': mean_vector,
        'pca': pca
    }

print("PCA subspaces computed for all classes.")
```

```
PCA subspaces computed for all classes.
```

Step 6 — Classify Test Images Using Reconstruction Error

Goal:

For each test image:

1. Project it onto the PCA subspace of each class
2. Reconstruct the image from the subspace
3. Compute the **reconstruction error**
4. Assign the class with the **smallest reconstruction error** as the predicted label

In [35]:
```
# Function to compute reconstruction error for a single sample and a given cla
def reconstruction_error(x, mean_vector, pca_model):
    x_centered = x - mean_vector
    # Project onto PCA subspace
    coeffs = pca_model.transform(x_centered.reshape(1, -1))
    # Reconstruct
    x_reconstructed = pca_model.inverse_transform(coeffs) + mean_vector
    # Compute squared L2 norm (can also use np.linalg.norm for L2)
    error = np.linalg.norm(x - x_reconstructed)
    return error

# Predict labels for test set
y_test_pred = []

for i in range(X_test.shape[0]):
    x = X_test[i]
    # Compute reconstruction error for all classes
    errors = []
    for digit in class_pca.keys():
        mean_vector = class_pca[digit]['mean']
        pca_model = class_pca[digit]['pca']
        err = reconstruction_error(x, mean_vector, pca_model)
        errors.append(err)
    # Predicted class = class with minimum reconstruction error
    pred_label = np.argmin(errors)
    y_test_pred.append(pred_label)

y_test_pred = np.array(y_test_pred)
print("Prediction on test set completed.")
```

```
Prediction on test set completed.
```

Step 7 — Evaluate PCA-Based Recognizer

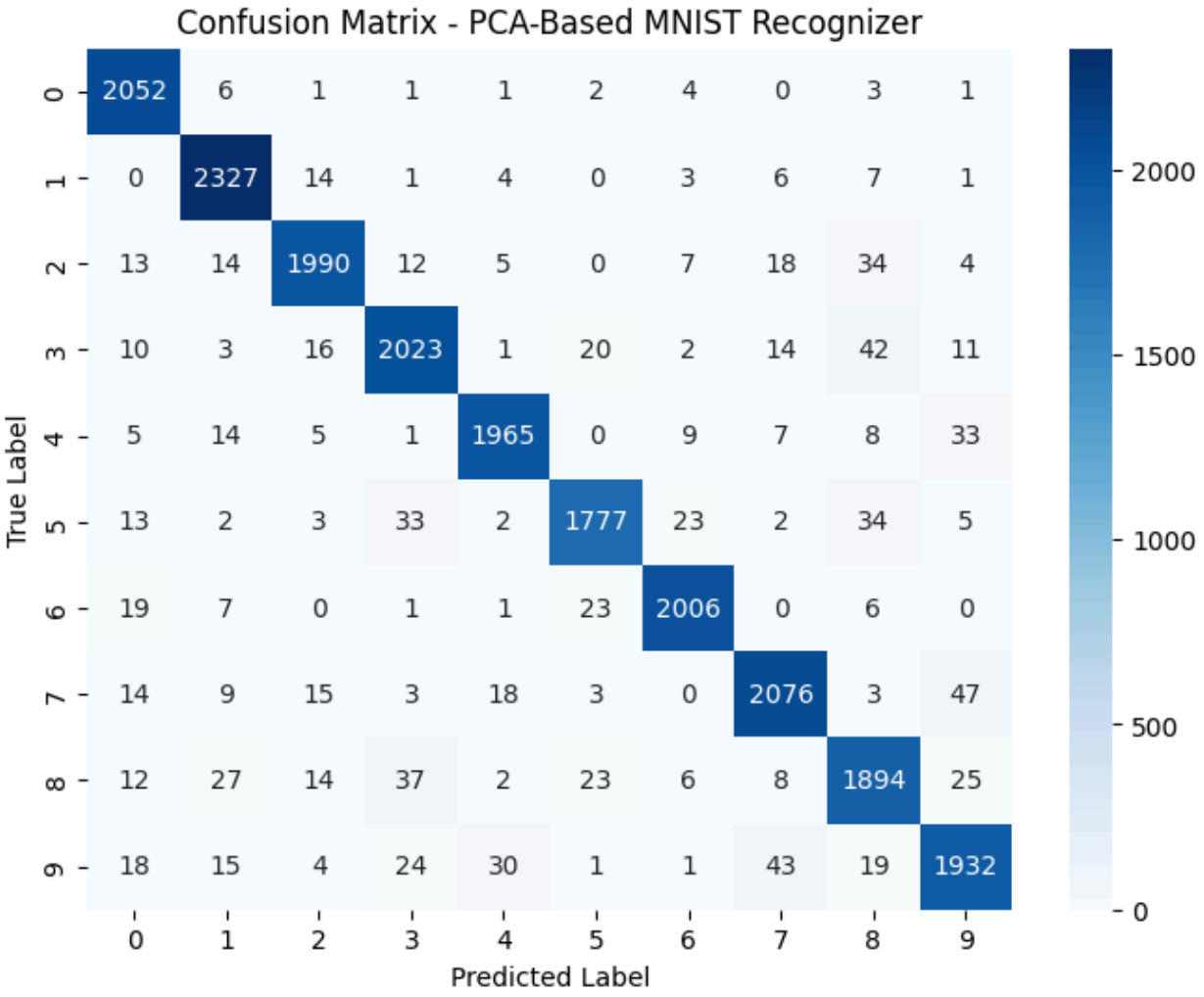
Goal:

- Compute **accuracy**
- Plot the **confusion matrix**
- Report **precision, recall, and F1-score** for each class *italicized text*

In [36]:
```
# Compute accuracy
accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {accuracy*100:.2f}%")

# Confusion matrix
cm = confusion_matrix(y_test, y_test_pred)

# Plot confusion matrix
plt.figure(figsize=(8,6))
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix - PCA-Based MNIST Recognizer")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()

# Classification metrics
print("\nClassification Report:")
print(classification_report(y_test, y_test_pred))
```

```
Test Accuracy: 95.44%
```



Classification Report:

	precision	recall	f1-score	support
0	0.95	0.99	0.97	2071
1	0.96	0.98	0.97	2363
2	0.97	0.95	0.96	2097
3	0.95	0.94	0.95	2142
4	0.97	0.96	0.96	2047
5	0.96	0.94	0.95	1894
6	0.97	0.97	0.97	2063
7	0.95	0.95	0.95	2188
8	0.92	0.92	0.92	2048
9	0.94	0.93	0.93	2087
accuracy			0.95	21000
macro avg	0.95	0.95	0.95	21000
weighted avg	0.95	0.95	0.95	21000