LAB8

Name : Kushal Sourav B

Regno: 2347125

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.layers import UpSampling2D
from tensorflow.keras import layers, models
from sklearn.metrics import mean_squared_error
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Conv2DTranspose, Flatten, Dense, Reshap
```

In [3]:
```
(x_train, _), (x_test, _) = cifar10.load_data()
```

In [4]:
```
# Normalize pixel values to [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
```

In [5]:
```
# Define the CNN Autoencoder
input_shape = x_train.shape[1:]
```

In [6]:
```
# Encoder
encoder = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D((2, 2), padding='same'),
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D((2, 2), padding='same')
])
```

In [7]:
```
# Decoder
decoder = models.Sequential([
    layers.Conv2DTranspose(64, (3, 3), activation='relu', padding='same'),
    layers.UpSampling2D((2, 2)),
    layers.Conv2DTranspose(32, (3, 3), activation='relu', padding='same'),
    layers.UpSampling2D((2, 2)),
    layers.Conv2DTranspose(3, (3, 3), activation='sigmoid', padding='same')
])
```

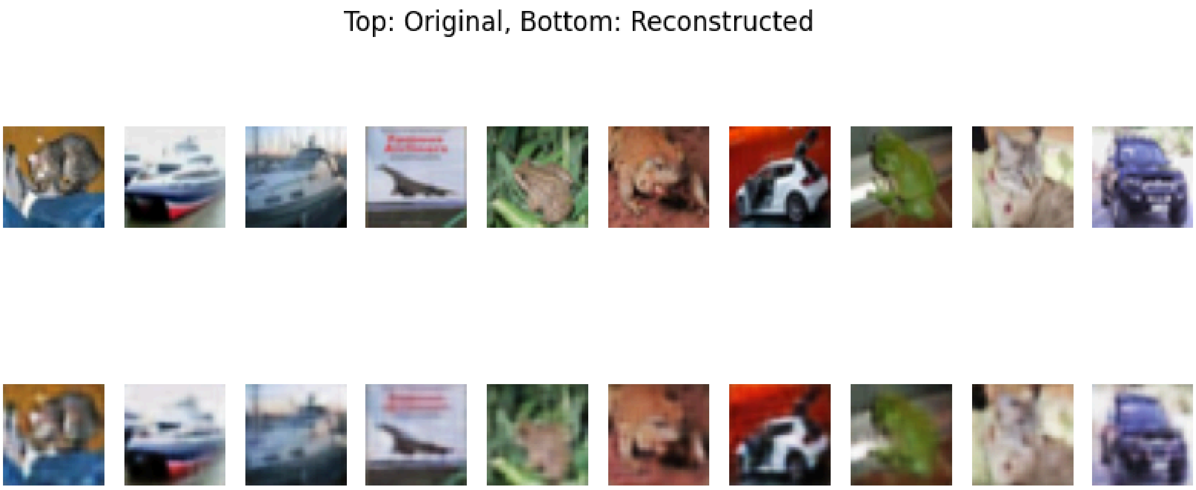
In [8]:
```
autoencoder = models.Sequential([encoder, decoder])
autoencoder.compile(optimizer='adam', loss='mse')
```

In [9]:
```
history = autoencoder.fit(x_train, x_train, epochs=20, batch_size=128, validation_split=0.2, verbose=1)
```

```
Epoch 1/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 47s 124ms/step - loss: 0.0224 - val_loss: 0.0066
Epoch 2/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 30s 95ms/step - loss: 0.0060 - val_loss: 0.0049
Epoch 3/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 29s 94ms/step - loss: 0.0049 - val_loss: 0.0045
Epoch 4/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 31s 99ms/step - loss: 0.0043 - val_loss: 0.0039
Epoch 5/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 29s 94ms/step - loss: 0.0040 - val_loss: 0.0037
Epoch 6/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 35s 111ms/step - loss: 0.0037 - val_loss: 0.0037
Epoch 7/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 36s 116ms/step - loss: 0.0036 - val_loss: 0.0034
Epoch 8/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 34s 107ms/step - loss: 0.0034 - val_loss: 0.0033
Epoch 9/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 101ms/step - loss: 0.0032 - val_loss: 0.0030
Epoch 10/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 31s 100ms/step - loss: 0.0031 - val_loss: 0.0029
Epoch 11/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 101ms/step - loss: 0.0030 - val_loss: 0.0028
Epoch 12/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 101ms/step - loss: 0.0029 - val_loss: 0.0027
Epoch 13/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 103ms/step - loss: 0.0028 - val_loss: 0.0026
Epoch 14/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 103ms/step - loss: 0.0027 - val_loss: 0.0026
Epoch 15/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 102ms/step - loss: 0.0026 - val_loss: 0.0024
Epoch 16/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 31s 101ms/step - loss: 0.0025 - val_loss: 0.0025
Epoch 17/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 102ms/step - loss: 0.0024 - val_loss: 0.0023
Epoch 18/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 101ms/step - loss: 0.0024 - val_loss: 0.0023
Epoch 19/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 102ms/step - loss: 0.0023 - val_loss: 0.0022
Epoch 20/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 32s 101ms/step - loss: 0.0023 - val_loss: 0.0022
```

In [10]:
```python
# Visualize Input vs Reconstructed Images
decoded_imgs = autoencoder.predict(x_test[:10])

plt.figure(figsize=(10, 4))
for i in range(10):
    # Original Images
    plt.subplot(2, 10, i + 1)
    plt.imshow(x_test[i])
    plt.axis('off')

    # Reconstructed Images
    plt.subplot(2, 10, i + 11)
    plt.imshow(decoded_imgs[i])
    plt.axis('off')
plt.suptitle("Top: Original, Bottom: Reconstructed")
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 162ms/step
```

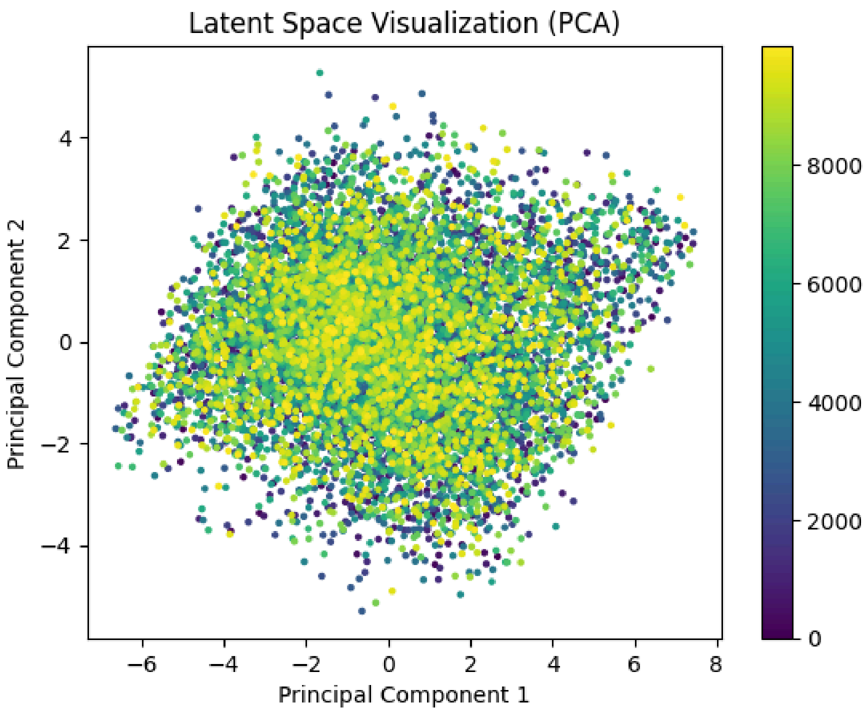


In [11]:
```
#MSE
reconstructed_imgs = autoencoder.predict(x_test)
mse = np.mean([mean_squared_error(x_test[i].flatten(), reconstructed_imgs[i].flatten()) for i in range(l
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
print("Mean Squared Error:", mse)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step
Mean Squared Error: 0.0022499263
```

In [12]:
```
# Extract latent space representations
latent_space = encoder.predict(x_test)

# Flatten the latent space
latent_space_flat = latent_space.reshape(latent_space.shape[0], -1)

#PCA
pca = PCA(n_components=2)
latent_pca = pca.fit_transform(latent_space_flat)

# Visualizing PCA
plt.scatter(latent_pca[:, 0], latent_pca[:, 1], c=np.arange(len(latent_pca)), cmap='viridis', s=5)
plt.colorbar()
plt.title("Latent Space Visualization (PCA)")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
```



Key Questions

1. How does the CNN autoencoder perform in reconstructing images?

The reconstructed images are visually similar to the original images, although some details may be blurred due to compression.with a Mean Squared Error (MSE) of 0.0023. This indicates that the average pixel-wise difference between the original and reconstructed images is minimal.

2. What insights do you gain from visualizing the latent space?

The PCA visualization of the latent space shows a dense central region, where most data points cluster closely together, indicating that the autoencoder has effectively captured common features across the dataset. The points are distributed fairly uniformly without distinct separations, suggesting that the autoencoder encodes data in a continuous latent space rather than learning class-specific features. The range of the principal components spans approximately -8 to 8 on both axes, reflecting the variability in the encoded features. The color gradient, as represented by the color bar, may correspond to a property such as data index, reconstruction error, or class labels, which highlights subtle patterns in feature representation. Overall, the lack of distinct clusters implies that the autoencoder prioritizes general data reconstruction over capturing discrete categories inherent to the dataset.

In [13]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, RepeatVector, TimeDistributed
from tensorflow.keras import layers
from sklearn.metrics import mean_squared_error
```

In [14]:
```
data = pd.read_csv('./HistoricalQuotes.csv')
data['Date'] = pd.to_datetime(data['Date'])
data.set_index('Date', inplace=True)
```

In [15]:
```
data.shape
```

Out[15]: (2518, 5)

In [16]:
```
data.describe()
```

Out[16]:

	Volume
count	2.518000e+03
mean	7.258009e+07
std	5.663113e+07
min	1.136205e+07
25%	3.053026e+07
50%	5.295469e+07
75%	9.861006e+07
max	4.624423e+08

In [17]:
```
data.head()
```

Out[17]:

Date	Close/Last	Volume	Open	High	Low
2020-02-28	$273.36	106721200	$257.26	$278.41	$256.37
2020-02-27	$273.52	80151380	$281.1	$286	$272.96
2020-02-26	$292.65	49678430	$286.53	$297.88	$286.5
2020-02-25	$288.08	57668360	$300.95	$302.53	$286.13
2020-02-24	$298.18	55548830	$297.26	$304.18	$289.23

In [18]:
```
#data[' Close/Last'] = data[' Close/Last'].replace('[\$,]', '', regex=True).astype(float)
data[' Close/Last'] = data[' Close/Last'].replace({'\$': '', ',': ''}, regex=True)
data[' Close/Last'] = data[' Close/Last'].astype(float)

stock_prices = data[' Close/Last'].values.reshape(-1, 1)
```

In [19]:
```
scaler = MinMaxScaler(feature_range=(0, 1))
stock_prices_scaled = scaler.fit_transform(stock_prices)
```

In [20]:
```
# Create sequences for LSTM (e.g., using 30 days as input for each sequence)
sequence_length = 30
X = []
y = []

for i in range(len(stock_prices_scaled) - sequence_length):
    X.append(stock_prices_scaled[i:i+sequence_length, 0])
    y.append(stock_prices_scaled[i+sequence_length, 0])

X = np.array(X)
y = np.array(y)
```

In [21]:
```
X = X.reshape((X.shape[0], X.shape[1], 1))
```

In [22]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [23]:
```
# Define the LSTM Autoencoder model
def build_lstm_autoencoder(input_shape):
    model = Sequential()

    # Encoder
    model.add(LSTM(128, activation='relu', input_shape=input_shape, return_sequences=False))

    # Latent space representation
    model.add(RepeatVector(sequence_length))

    # Decoder
    model.add(LSTM(128, activation='relu', return_sequences=True))
    model.add(TimeDistributed(Dense(1)))

    model.compile(optimizer='adam', loss='mse')
    return model

# Build the model
model = build_lstm_autoencoder((X_train.shape[1], 1))
model.summary()
```

```
c:\Users\USER\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\layers\rnn\rnn.py:204:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 128)	66,560
repeat_vector (RepeatVector)	(None, 30, 128)	0
lstm_1 (LSTM)	(None, 30, 128)	131,584
time_distributed (TimeDistributed)	(None, 30, 1)	129

Total params: 198,273 (774.50 KB)

Trainable params: 198,273 (774.50 KB)

Non-trainable params: 0 (0.00 B)

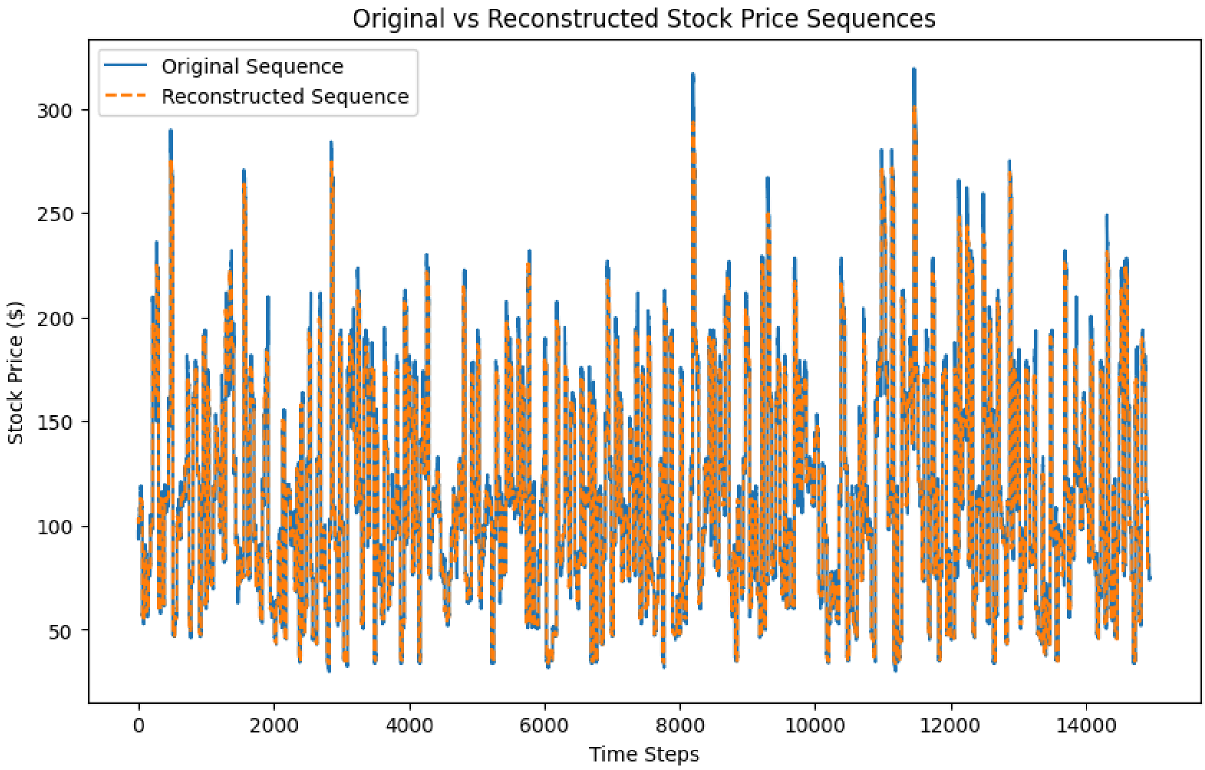
In [24]:
```
# Train the model
history = model.fit(X_train, X_train, epochs=20, batch_size=32, validation_data=(X_test, X_test), verbos
```

```
Epoch 1/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 38ms/step - loss: 0.0476 - val_loss: 0.0026
Epoch 2/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 30ms/step - loss: 0.0021 - val_loss: 7.5617e-04
Epoch 3/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 30ms/step - loss: 7.6230e-04 - val_loss: 4.3969e-04
Epoch 4/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 29ms/step - loss: 4.6208e-04 - val_loss: 3.2011e-04
Epoch 5/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 3.5181e-04 - val_loss: 3.9033e-04
Epoch 6/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 29ms/step - loss: 4.7196e-04 - val_loss: 3.4962e-04
Epoch 7/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 29ms/step - loss: 3.4334e-04 - val_loss: 2.8275e-04
Epoch 8/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 3.1735e-04 - val_loss: 2.6467e-04
Epoch 9/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 2.9779e-04 - val_loss: 2.5380e-04
Epoch 10/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 2.8341e-04 - val_loss: 2.4023e-04
Epoch 11/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 3.0966e-04 - val_loss: 2.4620e-04
Epoch 12/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 2.8737e-04 - val_loss: 2.3934e-04
Epoch 13/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 2.9158e-04 - val_loss: 2.5511e-04
Epoch 14/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 29ms/step - loss: 2.6068e-04 - val_loss: 2.5162e-04
Epoch 15/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 29ms/step - loss: 2.5000e-04 - val_loss: 2.8233e-04
Epoch 16/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 29ms/step - loss: 3.1460e-04 - val_loss: 3.2619e-04
Epoch 17/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 3.9594e-04 - val_loss: 3.3375e-04
Epoch 18/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 2.9963e-04 - val_loss: 2.5509e-04
Epoch 19/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 29ms/step - loss: 2.6783e-04 - val_loss: 2.1969e-04
Epoch 20/20
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - loss: 2.4409e-04 - val_loss: 2.3135e-04
```

In [25]:
```
# Predict the reconstructed sequences on test set
X_pred = model.predict(X_test)

# Inverse transform to get the original stock prices (from scaled data)
X_test_inv = scaler.inverse_transform(X_test.reshape(-1, 1))
X_pred_inv = scaler.inverse_transform(X_pred.reshape(-1, 1))

# Plotting original vs reconstructed stock prices
plt.figure(figsize=(10, 6))
plt.plot(X_test_inv, label="Original Sequence")
plt.plot(X_pred_inv, label="Reconstructed Sequence", linestyle='dashed')

plt.xlabel("Time Steps")
plt.ylabel("Stock Price ($)")
```

```
plt.legend()
plt.title("Original vs Reconstructed Stock Price Sequences")
plt.show()
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 60ms/step
```

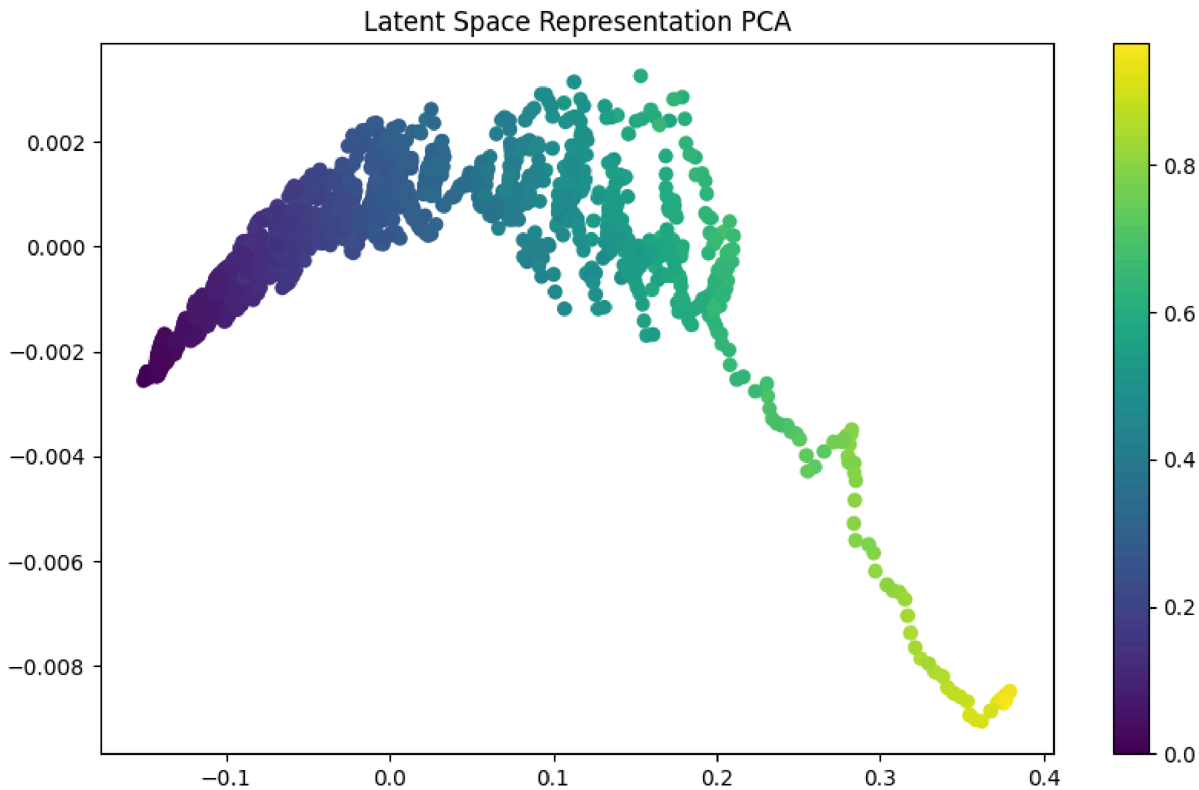


In [26]:
```
# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(X_test_inv, X_pred_inv)
print(f'Mean Squared Error (MSE): {mse}')
```

```
Mean Squared Error (MSE): 20.456886473895846
```

In [27]:
```
# Extract the encoder part of the model to get the latent representations
encoder = Sequential()
encoder.add(LSTM(128, activation='relu', input_shape=(X_train.shape[1], 1), return_sequences=False))

# Fit the encoder to get the latent space representation
latent_representations = encoder.predict(X_train)

# Visualize latent representations
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
latent_pca = pca.fit_transform(latent_representations)

plt.figure(figsize=(10, 6))
plt.scatter(latent_pca[:, 0], latent_pca[:, 1], c=y_train, cmap='viridis')
plt.colorbar()
plt.title("Latent Space Representation PCA")
plt.show()
```

```
 1/63 ━━━━━━━━━━━━━━━━━━━━ 6s 110ms/step
c:\Users\USER\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\layers\rnn\rnn.py:204:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
```



Key Questions

1. How well does the LSTM autoencoder reconstruct the sequences?

The LSTM autoencoder will aim to reconstruct the stock price sequences based on the latent representation. By comparing the reconstructed sequences to the original ones using MSE, you can evaluate how well the model learns the temporal dependencies and reconstructs the stock price data. A lower MSE value indicates better reconstruction quality.

2. How does the choice of latent space dimensionality affect reconstruction quality and compression?

The number of LSTM units in the encoder and decoder determines the latent space dimensionality. If the latent space is too small, the reconstruction may lose important information, resulting in higher MSE and a less accurate reconstruction. A larger latent space retains more information but may reduce the compression efficiency. You can experiment with different LSTM unit sizes to find a balance between reconstruction quality and dimensionality.