In [19]:
```python
# Define a threshold value for anomalies
threshold_rnn = np.percentile(mse_rnn, 95)
# Predict anomalies
y_pred_rnn = mse_rnn > threshold_rnn
# Ensure y_pred_rnn has the same length as y_test
y_pred_rnn = y_pred_rnn[:len(y_test)]
# Flatten predictions for evaluation
y_pred_rnn = y_pred_rnn.flatten()
# Evaluate the performance
print("RNN Model Performance")
print(classification_report(y_test, y_pred_rnn))
```

```
RNN Model Performance
              precision    recall  f1-score   support

           0       1.00      0.94      0.97    284315
           1       0.00      0.08      0.00       492

    accuracy                           0.94    284807
   macro avg       0.50      0.51      0.49    284807
weighted avg       1.00      0.94      0.97    284807
```

Autoencoders

- **Paradigm**: Unsupervised anomaly detection
- **Distinctive Features**: – Reconstructive learning methodology
  - Latent space representation – Anomaly highlighting capabilities

Building the Autoencoder Model Architecture

Layer Breakdown

Encoder

- **Input Layer** (Input_Layer): This is the input layer. The shape is defined by the number of features in your data (input_dim). This layer takes in the raw data.
- **First Encoding Layer** (Encoding_Layer_1): The first encoding layer reduces the input dimensions to 32. The relu activation function introduces non-linearity, helping the model learn complex patterns.
- **Second Encoding Layer** (Encoding_Layer_2): This layer further reduces the dimensions to 16, continuing to compress the information while maintaining important features.
- **Third Encoding Layer** (Encoding_Layer_3): The last encoding layer compresses the data to an 8-dimensional space. This is the bottleneck layer, capturing the most compressed representation of the data. The small dimension helps isolate anomalies since anomalies will differ more significantly after being compressed and reconstructed.

Decoder

- **First Decoding Layer** (Decoding_Layer_1): This layer begins to reconstruct the data by expanding the dimensions back to 16. The goal is to mirror the encoding layers to regain the original data shape.
- **Second Decoding Layer** (Decoding_Layer_2): Here, the layer further expands the dimensions to 32, following the symmetric structure of the autoencoder.
- **Output Layer** (Output_Layer): The final layer reconstructs the data back to the original input dimension. The sigmoid activation function is used to ensure the output ranges between 0 and 1, which is beneficial if you're dealing with normalized data.
- **Layer Sizes**: The layer sizes progressively decrease and then increase to form a bottleneck, which helps capture key features and isolate anomalies.
- **Activation Functions**: ReLU is used in the hidden layers to introduce non-linearity and improve learning capacity. Sigmoid in the output layer ensures that the reconstructed outputs remain in the same range as the input data.

This architecture was chosen to effectively compress the input data to a lower-dimensional space (encoder) and then reconstruct it back to its original form (decoder). By training the autoencoder to minimize reconstruction error, the model learns to distinguish between normal data and anomalies (which have higher reconstruction errors).

In [20]:
```
#STEP 1: Import libraries
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.callbacks import EarlyStopping
```

```python
import numpy as np

#STEP 2: Preprocess the Data
# nothing to do, we reuse the same data as for LSTM model also for the RNN model

#STEP3: Build the Autoencoder Model
# Define the input dimension
input_dim = X_train.shape[1]

# Define the Autoencoder model

#Input layer
input_layer = Input(shape=(input_dim,), name='Input_Layer')

#Encoder
encoded = Dense(32, activation='relu', name='Encoding_Layer_1')(input_layer)
encoded = Dense(16, activation='relu', name='Encoding_Layer_2')(encoded)
encoded = Dense(8, activation='relu', name='Encoding_Layer_3')(encoded)

#Decoder
decoded = Dense(16, activation='relu', name='Decoding_Layer_1')(encoded)
decoded = Dense(32, activation='relu', name='Decoding_Layer_2')(decoded)

#Output layer
output_layer = Dense(input_dim, activation='sigmoid', name='Output_Layer')(decoded)

# Create the Autoencoder model
autoencoder = Model(inputs=input_layer, outputs=output_layer, name='Autoencoder_Fraud_Detection')
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
print(autoencoder.summary())

#STEP4: Train the Model
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
history = autoencoder.fit(X_train, X_train, epochs=20, batch_size=64, validation_data=(X_test, X_test), callbacks=[early_stopping

#STEP5: Evaluate the Model
# Calculate reconstruction error on the test set
reconstructed = autoencoder.predict(X_test)
mse = np.mean(np.power(X_test- reconstructed, 2), axis=1)

# Define a threshold value for anomalies
```

```
threshold = np.percentile(mse, 95)

# Predict anomalies
y_pred = mse > threshold
```

Model: "Autoencoder_Fraud_Detection"

Layer (type)	Output Shape	Param #
Input_Layer (InputLayer)	(None, 30)	0
Encoding_Layer_1 (Dense)	(None, 32)	992
Encoding_Layer_2 (Dense)	(None, 16)	528
Encoding_Layer_3 (Dense)	(None, 8)	136
Decoding_Layer_1 (Dense)	(None, 16)	144
Decoding_Layer_2 (Dense)	(None, 32)	544
Output_Layer (Dense)	(None, 30)	990

Total params: 3,334 (13.02 KB)

Trainable params: 3,334 (13.02 KB)

Non-trainable params: 0 (0.00 B)

```
None
Epoch 1/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 16s 3ms/step - loss: 0.8809 - val_loss: 0.8214
Epoch 2/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - loss: 0.7819 - val_loss: 0.7967
Epoch 3/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 12s 3ms/step - loss: 0.7576 - val_loss: 0.7875
Epoch 4/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - loss: 0.7431 - val_loss: 0.7745
Epoch 5/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - loss: 0.7341 - val_loss: 0.7712
Epoch 6/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 15s 3ms/step - loss: 0.7248 - val_loss: 0.7685
Epoch 7/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 18s 4ms/step - loss: 0.7311 - val_loss: 0.7668
Epoch 8/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 23s 5ms/step - loss: 0.7206 - val_loss: 0.7644
Epoch 9/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 15s 3ms/step - loss: 0.7324 - val_loss: 0.7624
Epoch 10/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 20s 5ms/step - loss: 0.7267 - val_loss: 0.7613
Epoch 11/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 18s 4ms/step - loss: 0.7188 - val_loss: 0.7594
Epoch 12/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 16s 4ms/step - loss: 0.7209 - val_loss: 0.7583
Epoch 13/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 21s 5ms/step - loss: 0.7252 - val_loss: 0.7573
Epoch 14/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 14s 3ms/step - loss: 0.7146 - val_loss: 0.7559
Epoch 15/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - loss: 0.7240 - val_loss: 0.7568
Epoch 16/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 15s 3ms/step - loss: 0.7319 - val_loss: 0.7555
Epoch 17/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 22s 4ms/step - loss: 0.7138 - val_loss: 0.7542
Epoch 18/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 16s 4ms/step - loss: 0.7203 - val_loss: 0.7539
Epoch 19/20
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 17s 4ms/step - loss: 0.7105 - val_loss: 0.7535
Epoch 20/20
```

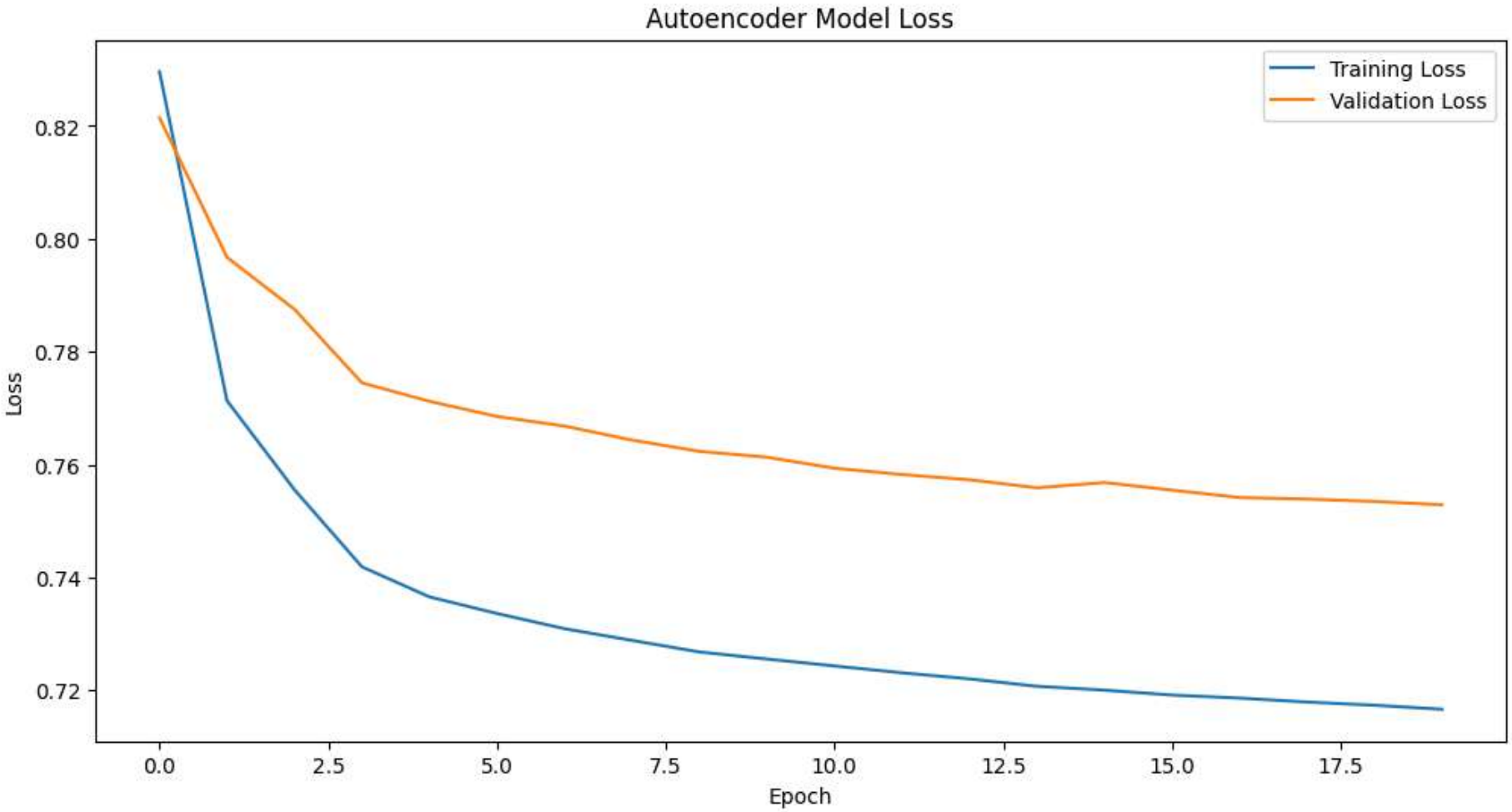
```
4443/4443 ━━━━━━━━━━━━━━━━━━━━ 15s 3ms/step - loss: 0.7125 - val_loss: 0.7529
8901/8901 ━━━━━━━━━━━━━━━━━━━━ 14s 2ms/step
```

In [21]:
```python
#STEP6: Evaluate the Model
# Evaluate the performance
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
plt.figure(figsize=(12, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Autoencoder Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
              precision    recall  f1-score   support

           0       1.00      0.95      0.97    284315
           1       0.03      0.87      0.06       492

    accuracy                           0.95    284807
   macro avg       0.51      0.91      0.52    284807
weighted avg       1.00      0.95      0.97    284807
```



Performance Evaluation

- **Comprehensive Metrics**:
  - Accuracy
  - Precision
  - Recall
  - F1-Score

- Computational Efficiency

Autoencoders Model:

- **Accuracy**: 0.95
- **Macro Avg Precision/Recall/F1-Score**: 0.52/0.91/0.52
- **Weighted Avg Precision/Recall/F1-Score**: 1.00/0.95/0.97

RNN Model:

- **To detect Class 0**:
  - **Precision**: 1.00
  - **Recall**: 0.94
  - **F1-Score**: 0.97
- **To detect Class 1**:
  - Precision: 0.00
  - Recall: 0.08
  - F1-Score: 0.00
- **Accuracy**: 0.94
- **Macro Avg Precision/Recall/F1-Score**: 0.50/0.51/0.49
- **Weighted Avg Precision/Recall/F1-Score**: 1.00/0.94/0.97

LSTM Model:

- **Similar performance to RNN**:
  - **Class 0**:
    - **Precision**: 1.00
    - **Recall**: 0.94
    - **F1-Score**: 0.97

    - **Class 1**:
        - **Precision**: 0.00
        - **Recall**: 0.08
        - **F1-Score**: 0.00
- **Accuracy**: 0.94
- **Macro Avg Precision/Recall/F1-Score**: 0.50/0.51/0.49
- **Weighted Avg Precision/Recall/F1-Score**: 1.00/0.94/0.97

Comparison Highlights:

- **Autoencoders**: Exhibits slightly higher accuracy and stronger performance in macro avg precision and recall and are computationally much more efficient.
- **RNN** & **LSTM**: Both show similar efficiencies and higher weighted avg metrics but struggle with imbalance in detection of Class 1. It takes much longer time to tran and evauate them.

Results and Insights

Model Performance Ranking:

- **LSTM** & **RNN**: – Both showed nearly identical performance, so saying that the LSTM is "superior" might be an overstatement. Both models had challenges with detecting Class 1 (fraud), reflected by a precision of 0.00 and recall of 0.08. – Their strength lies in detecting Class 0 with a precision of 1.00 and an overall accuracy of 0.94.
- **Autoencoder**: – Indeed performed with a slightly higher accuracy (0.95) compared to RNN and LSTM (0.94). – Generates higher false positives but offers advantages in anomaly detection and visualization. – The weighted averages are quite high, but there are general challenges with imbalanced classification, evident in macro avg.

Refined Conclusion:

Model Performance Ranking

- 1. **Autoencoder**:

    - Marginally better overall accuracy.
    - Suitable for anomaly detection and visualization, despite higher false positives.
- 2. **RNN** & **LSTM** (tied):
    - Both models show robust performance in sequential learning.
    - Strong in detecting non-fraudulent cases, less effective in fraud detection.
    - Accurate yet struggle with class imbalance, evidenced by low precision and recall for fraud cases.

Recommendations

Immediate Implementation

- Shift focus to an Autoencoder-based fraud detection system considering its marginally better performance and higher accuracy.
- Develop robust model monitoring infrastructure to identify performance drifts and anomalies.
- Implement continuous learning mechanisms to adapt to new patterns and maintain performance.

Future Research Directions

- Integrate additional contextual features that might improve model sensitivity and specificity.
- Explore hybrid model architectures that combine the strengths of different models for more robust detection.
- Develop real-time fraud detection capabilities to immediately flag and address suspicious activities.
- Investigate advanced sampling techniques to better manage imbalanced datasets and improve the detection of rare fraud cases.

Analysis Summary:

In real-world scenarios, credit card fraud detection often deals with highly imbalanced data. Upon comparison of different models, we determined that the Autoencoder model performed the best given the current data.

Possible Flaws in the Autoencoder Model:

- 1. **Class Imbalance Bias**: Even though the Autoencoder performed well, it could still be biased towards the majority class, leading to higher false positives.
- 2. **Overfitting Risks**: Given its complexity, the Autoencoder might overfit the training data, making it less effective on unseen data.
- 3. **Feature Limitations**: The current features might not fully capture the complexities of fraud patterns, limiting the Autoencoder's efficacy.
- 4. **Scalability Challenges**: The computational cost of running the Autoencoder in real-time can be significant, posing scalability issues.
- 5. **Metric Dependence**: The reliance on accuracy as a primary metric can be misleading, especially in imbalanced datasets, requiring more comprehensive metric use.

Plan for future Action:

- 1. **Data Augmentation**:
  - **Collect Additional Data**: Gather more diverse datasets including various types of fraudulent transactions for improved representation.
  - **Synthetic Oversampling**: Apply techniques like SMOTE to artificially balance the dataset.
- 2. **Exploring Alternative Models**:
  - **Hybrid Models**: Combine the autoencoder's anomaly detection strengths with classification models to improve overall effectiveness.
  - **Advanced Anomaly Detection Techniques**: Experiment with **Isolation Forests**, **One-Class SVMs**, and **GANs** (Generative Adversarial Networks) to compare performance.
- 3. **Feature Engineering**:
  - **Contextual Features**: Introduce additional features such as geolocation, transaction sequences, and user behavior patterns.
  - **Temporal Patterns**: Extract and utilize features that capture transaction timestamps and sequences to detect temporal fraud patterns better.
- 4. **Comprehensive Evaluation Metrics**:
  - **Additional Metrics**: Focus on metrics like Precision, Recall, F1-Score, and ROC-AUC to get a balanced view of the model's performance.
- 5. **Continuous Model Improvement**:
  - **Active Learning**: Continuously update the model with new data to adapt to evolving fraud patterns.
  - **Robust Monitoring**: Implement a real-time monitoring system to detect and address model performance drifts.
- 6. **Scalability and Efficiency**:

- **Optimization**: Optimize the autoencoder for lower computational cost while maintaining high performance.
- **Resource Allocation**: Utilize distributed computing techniques to manage large-scale data operations efficiently.

Concluding Observations

This research underscores the transformative potential of deep learning in financial fraud detection. It highlights the critical role of sophisticated machine learning techniques, such as **Autoencoders**, **RNN**, and **LSTM**, in safeguarding financial ecosystems, and underscores the importance of continuous learning and advanced methodologies to adapt to emerging threats.