Group Assignment : Credit Card Fraud Detection

Member:

- TIANYANG CHEN
- GEETHIKA REDDY KONDA
- NICOLÁS LEYVA GONZÁLEZ
- KAREEM RAMI JAMIL HAWWASH
- OMAR MEKKAWI YASSIN

Course: Machine Learning Foundations 2025
Date: April 28, 2025

- **Data source:** https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud/data
- **GitHub Repository:** git@github.com:Geethika2506/Machine-Learning-final-project.git

In [1]:
```
import os
import pandas as pd
import numpy as npgit
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

In [4]:
```
import pandas as pd
from tkinter import Tk, filedialog

# Hide the root Tk window
Tk().withdraw()

# Open a file dialog and let user select the CSV file
file_path = filedialog.askopenfilename(
    title="Select the creditcard.csv dataset",
    filetypes=[("CSV files", "*.csv")]
)

if file_path:
    df = pd.read_csv(file_path)
    print("File loaded successfully!")
else:
    print("No file selected. Please select the dataset file.")
```

```
File loaded successfully!
```

In [2]:
```
df = pd.read_csv(file_path)
```

In [3]:
```
df
```

Out[3]:

	Time	V1	V2	V3	V4	V5	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095
...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649

284807 rows × 31 columns

Task1: Data Exploration and Preprocessing

1.1 Exploratory data analysis (EDA)

1.1.1 Basic dataset info

In [4]:

```
print("Dataset shape:", df.shape)
print(df.info())
print(df.describe())
```

```
Dataset shape: (284807, 31)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64  
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
None
                Time            V1            V2            V3            V4
\
count  284807.000000  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean    94813.859575  1.168375e-15  3.416908e-16 -1.379537e-15  2.074095e-15
std     47488.145955  1.958696e+00  1.651309e+00  1.516255e+00  1.415869e+00
min         0.000000 -5.640751e+01 -7.271573e+01 -4.832559e+01 -5.683171e+00
25%     54201.500000 -9.203734e-01 -5.985499e-01 -8.903648e-01 -8.486401e-01
50%     84692.000000  1.810880e-02  6.548556e-02  1.798463e-01 -1.984653e-02
75%    139320.500000  1.315642e+00  8.037239e-01  1.027196e+00  7.433413e-01
max    172792.000000  2.454930e+00  2.205773e+01  9.382558e+00  1.687534e+01

                 V5            V6            V7            V8            V9
\
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   9.604066e-16  1.487313e-15 -5.556467e-16  1.213481e-16 -2.406331e-15
std    1.380247e+00  1.332271e+00  1.237094e+00  1.194353e+00  1.098632e+00
```

```
min   -1.137433e+02 -2.616051e+01 -4.355724e+01 -7.321672e+01 -1.343407e+01
25%   -6.915971e-01 -7.682956e-01 -5.540759e-01 -2.086297e-01 -6.430976e-01
50%   -5.433583e-02 -2.741871e-01  4.010308e-02  2.235804e-02 -5.142873e-02
75%    6.119264e-01  3.985649e-01  5.704361e-01  3.273459e-01  5.971390e-01
max    3.480167e+01  7.330163e+01  1.205895e+02  2.000721e+01  1.559499e+01

       ...           V21           V22           V23           V24  \
count  ...  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   ...  1.654067e-16 -3.568593e-16  2.578648e-16  4.473266e-15
std    ...  7.345240e-01  7.257016e-01  6.244603e-01  6.056471e-01
min    ... -3.483038e+01 -1.093314e+01 -4.480774e+01 -2.836627e+00
25%    ... -2.283949e-01 -5.423504e-01 -1.618463e-01 -3.545861e-01
50%    ... -2.945017e-02  6.781943e-03 -1.119293e-02  4.097606e-02
75%    ...  1.863772e-01  5.285536e-01  1.476421e-01  4.395266e-01
max    ...  2.720284e+01  1.050309e+01  2.252841e+01  4.584549e+00

                V25           V26           V27           V28         Amount
\
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  284807.000000
mean   5.340915e-16  1.683437e-15 -3.660091e-16 -1.227390e-16      88.349619
std    5.212781e-01  4.822270e-01  4.036325e-01  3.300833e-01     250.120109
min   -1.029540e+01 -2.604551e+00 -2.256568e+01 -1.543008e+01       0.000000
25%   -3.171451e-01 -3.269839e-01 -7.083953e-02 -5.295979e-02       5.600000
50%    1.659350e-02 -5.213911e-02  1.342146e-03  1.124383e-02      22.000000
75%    3.507156e-01  2.409522e-01  9.104512e-02  7.827995e-02      77.165000
max    7.519589e+00  3.517346e+00  3.161220e+01  3.384781e+01   25691.160000

              Class
count  284807.000000
mean        0.001727
std         0.041527
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000

[8 rows x 31 columns]
```

Feature Characteristics

- **All features are numerical.**
- Most features are the result of **PCA transformation**, named as `V1` to `V28`.
- Only a few columns are in their original form:
  - `Time`: Seconds elapsed between this transaction and the first transaction in the dataset.
  - `Amount`: Transaction amount.
  - `Class`: Target variable:
    - `0`: Legitimate transaction
    - `1`: Fraudulent transaction

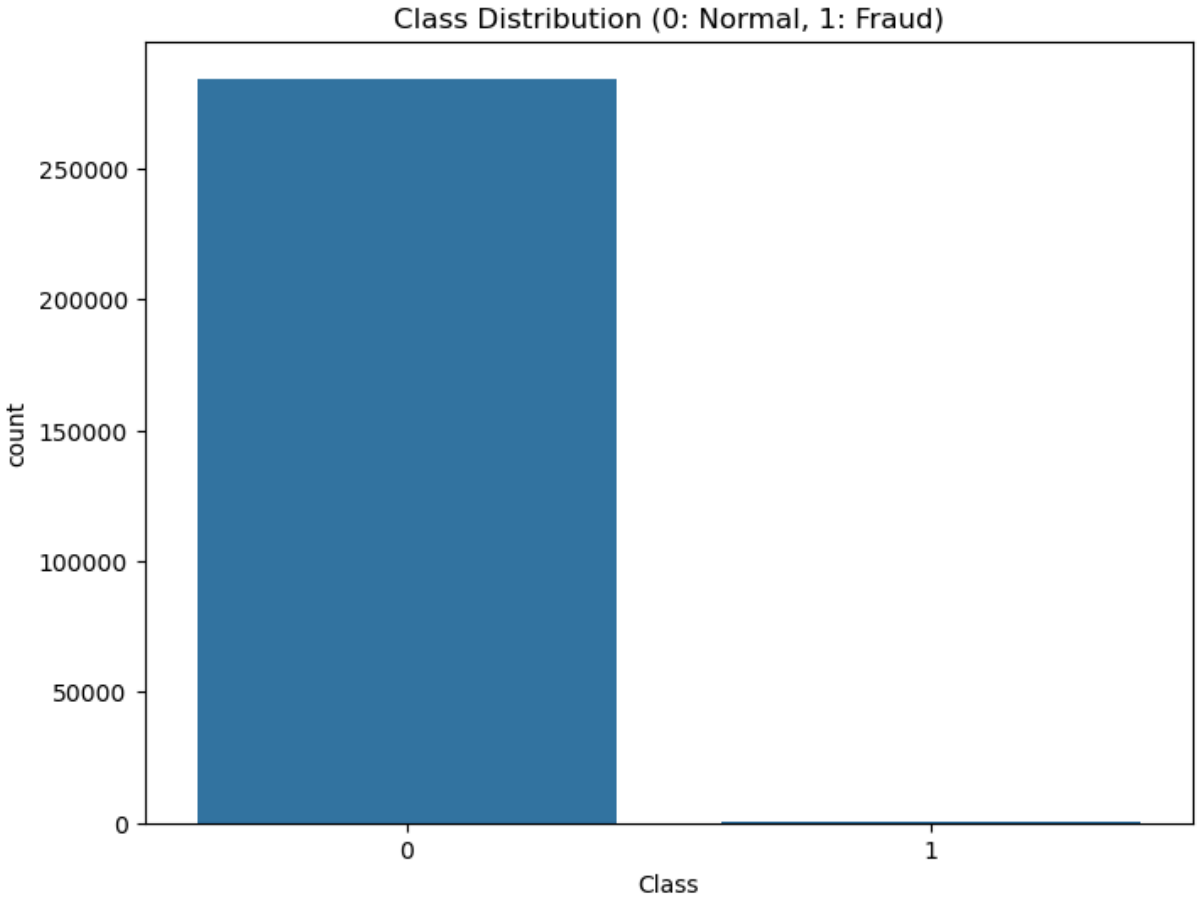
1.1.2 Check Feature Distribution

Class Distribution (Target variable)

In [5]:
```
# Check class distribution
print("Class distribution:")
print(df['Class'].value_counts())
print("Fraud percentage:", df['Class'].mean() * 100, "%")
```

```
Class distribution:
Class
0    284315
1       492
Name: count, dtype: int64
Fraud percentage: 0.1727485630620034 %
```

In [6]:
```
# Visualize class distribution
plt.figure(figsize=(8, 6))
sns.countplot(x='Class', data=df)
plt.title('Class Distribution (0: Normal, 1: Fraud)')
plt.show()
```



- Total number of transaction records: **284,807**
- Number of fraud cases (`Class = 1`): **492**
- Fraud cases proportion: **0.172%**

- The dataset is **extremely imbalanced**, with fraud cases being a tiny minority.

Class	Label	Count	Percentage
0	Non-Fraud	284,315	99.827%
1	Fraudulent	492	0.172%

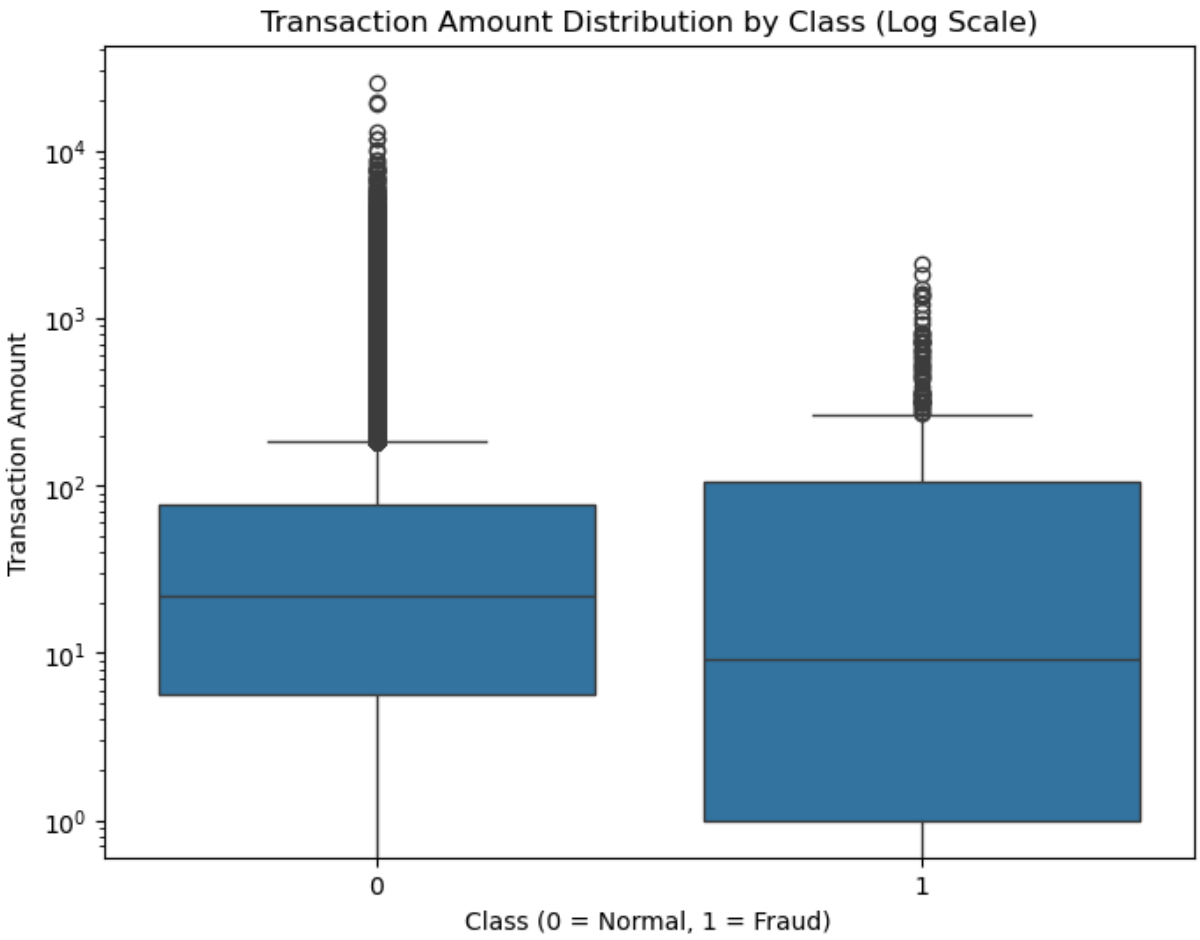
Caveat

- Due to the severe class imbalance:
  - Accuracy is **not a reliable metric**.
  - Emphasis should be placed on metrics such as **Precision**, **Recall**, **F1-Score**, and **ROC-AUC**.

Amount Distribution

Exploration: Fraud vs. Transaction Amount

In [7]:
```
plt.figure(figsize=(8, 6))
sns.boxplot(x='Class', y='Amount', data=df)
plt.yscale('log')  # using a logarithmic scale makes it more intuitive.
plt.title("Transaction Amount Distribution by Class (Log Scale)")
plt.xlabel("Class (0 = Normal, 1 = Fraud)")
plt.ylabel("Transaction Amount")
plt.show()
```



In [8]:
```
df.groupby('Class')['Amount'].describe()
```

Out[8]:

Class	count	mean	std	min	25%	50%	75%	max
0	284315.0	88.291022	250.105092	0.0	5.65	22.00	77.05	25691.16
1	492.0	122.211321	256.683288	0.0	1.00	9.25	105.89	2125.87

Key observations

1. **Fraudulent Transactions Tend to Be Smaller on Average**

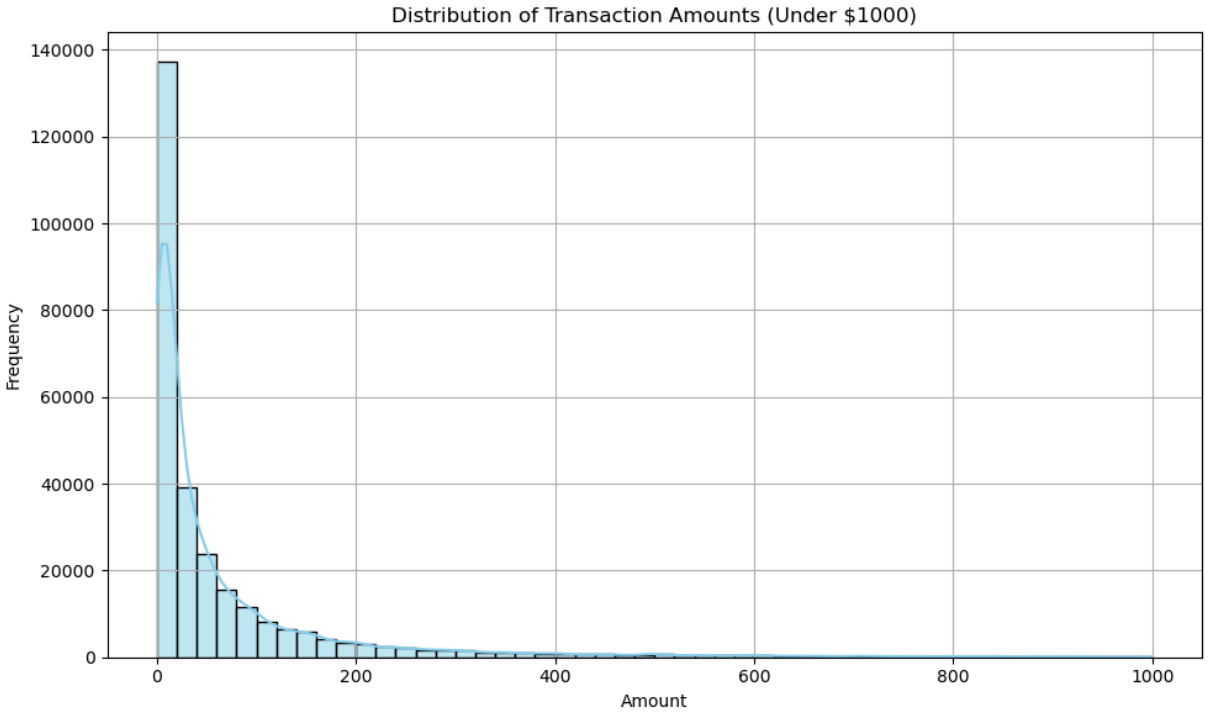
- Median amount for fraudulent transactions (Class = 1): ~9.25
- Median for normal transactions (Class = 0): ~22.00
- This suggests frauds are not necessarily associated with high transaction amounts — in fact, they often **target smaller amounts**, possibly to **avoid detection**.

2. **Smaller Spread in Fraudulent Amounts**

- According to the boxplot, legal transactions show a wider range than fraud transactions.

Overall Distribution

In [9]:
```
# Filter to focus on more common transaction sizes
filtered_amount = df[df['Amount'] < 1000]

plt.figure(figsize=(10, 6))
sns.histplot(filtered_amount['Amount'], bins=50, kde=True, color='skyblue')
plt.title('Distribution of Transaction Amounts (Under $1000)')
plt.xlabel('Amount')
plt.ylabel('Frequency')
plt.grid(True)
plt.tight_layout()
plt.show()
```



- The transaction amount is **highly right skewed**, with a large number of small-value transactions and a few very large outliers. Log transformation or robust scaling may be beneficial for modeling.
- According to previous statistics, the transaction amounts range from **0 to 25,691.16**, showing a wide spread and high variability.
- The 25th and 75th percentiles are 5.60 and 77.17 respectively, meaning that **50%** of transactions fall between **5.60 and 77.17** — most **transactions are small**.

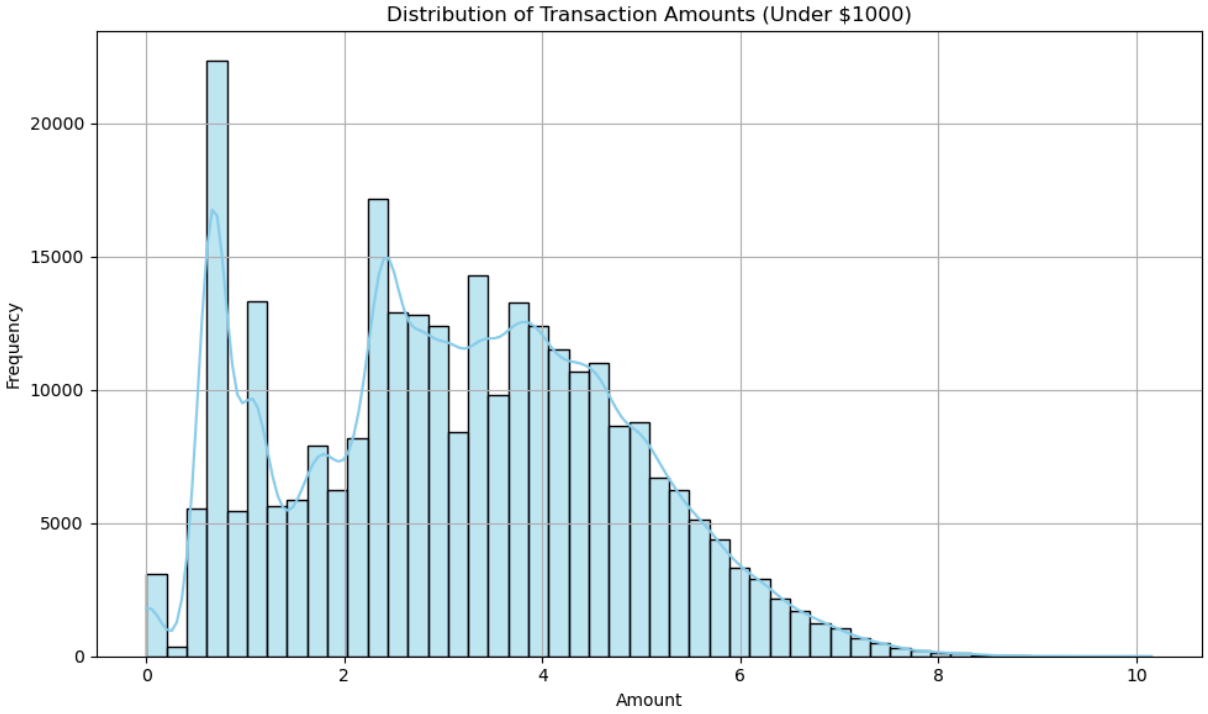
Log transformation

To prepare the feature `Amount` for machine learning, we applied a **logarithmic transformation** to the original values. This step is crucial because the raw

transaction amounts are highly skewed. Such skewness can negatively **affect model performance**, especially for algorithms **sensitive to feature distributions** (e.g., linear regression or neural networks).

In [10]:
```
df['Amount'] = np.log1p(df['Amount'])
```

In [11]:
```
# Filter to focus on more common transaction sizes
filtered_amount = df[df['Amount'] < 1000]

plt.figure(figsize=(10, 6))
sns.histplot(filtered_amount['Amount'], bins=50, kde=True, color='skyblue')
plt.title('Distribution of Transaction Amounts (Under $1000)')
plt.xlabel('Amount')
plt.ylabel('Frequency')
plt.grid(True)
plt.tight_layout()
plt.show()
```

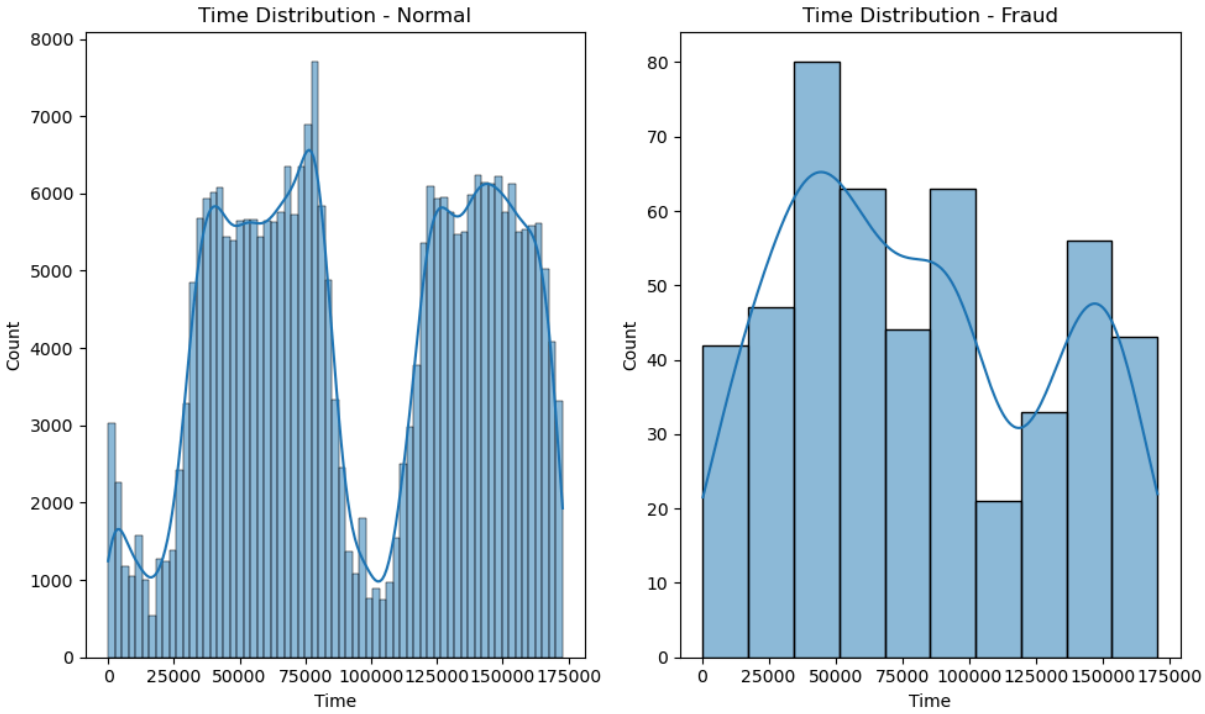


After applying the log transformation, the distribution of `Amoount` becomes **significantly more normalized and compact**, as shown in the plot. This helps stabilize variance, reduces the impact of extreme outliers, and generally improves the performance and convergence of machine learning models.

Time Distribution

In [12]:
```
# Analyze Time feature
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
sns.histplot(df[df['Class'] == 0]['Time'], kde=True)
plt.title('Time Distribution - Normal')
```

```
plt.subplot(1, 2, 2)
sns.histplot(df[df['Class'] == 1]['Time'], kde=True)
plt.title('Time Distribution - Fraud')
plt.tight_layout()
plt.show()
```



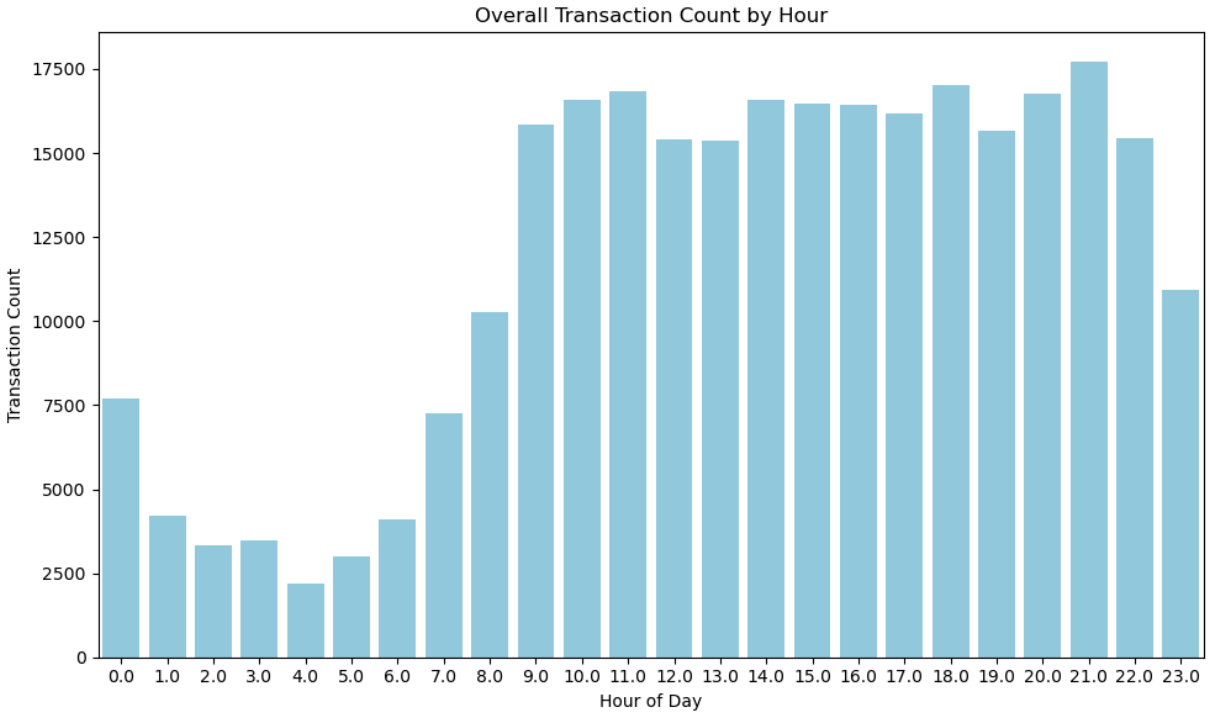
- The original Time feature represents **elapsed seconds** since the first transaction, but is not directly interpretable.
- To **uncover potential patterns** in user or fraudster behavior, we **converted it into Hour of the Day**, and grouped transactions by **time bins** (e.g., Night, Morning, Afternoon, Evening). This helps the model **learn time-dependent fraud patterns** and avoids overfitting to uninterpretable raw values.

In [13]:
```
# Create 'Hour' feature from 'Time'
df['Hour'] = (df['Time'] % 86400) // 3600  # integer division to get hour of

# Optional: create time of day bins
def time_bin(hour):
    if 0 <= hour < 6:
        return 'Night'
    elif 6 <= hour < 12:
        return 'Morning'
    elif 12 <= hour < 18:
        return 'Afternoon'
    else:
        return 'Evening'
```

```
df['TimeBin'] = df['Hour'].apply(time_bin)
df.drop(columns=['Time'], inplace=True)
```

We fist plot the Overall Transaction Count by Hour to check Time variable after transformation.

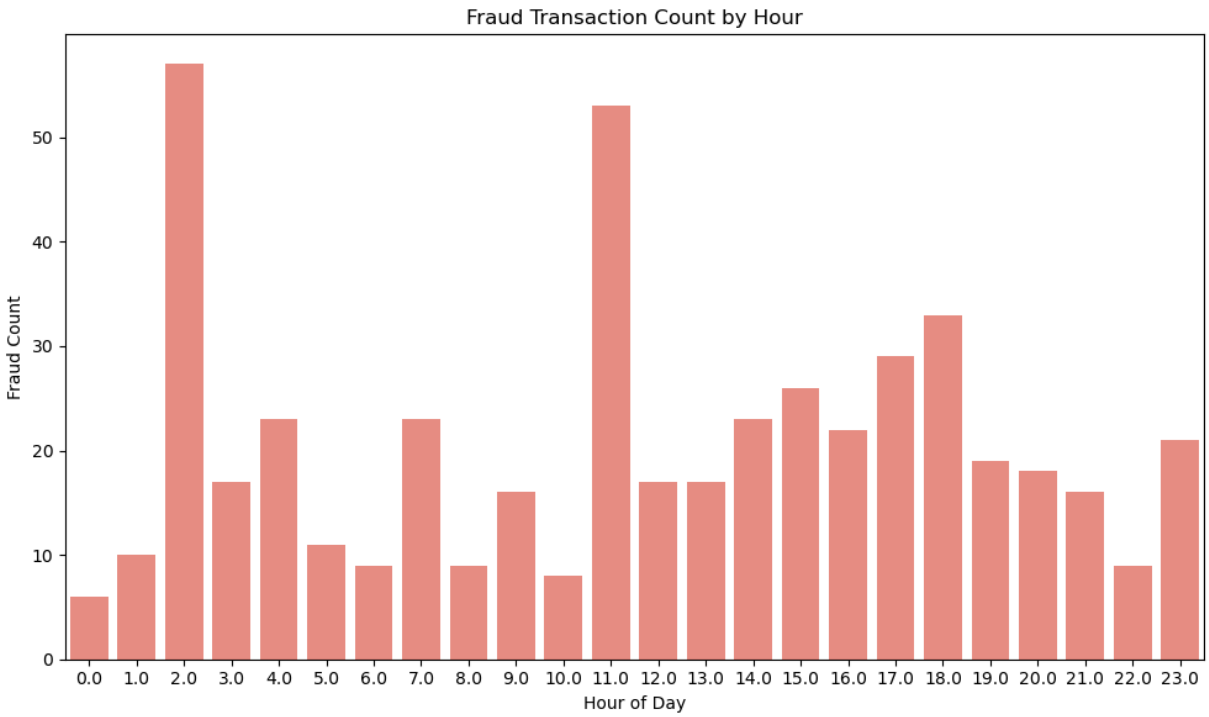
In [14]:
```
plt.figure(figsize=(10,6))
sns.countplot(data=df, x='Hour', color='skyblue')
plt.title("Overall Transaction Count by Hour")
plt.xlabel("Hour of Day")
plt.ylabel("Transaction Count")
plt.tight_layout()
plt.show()
```



We further plot fraud transaction by hour and the time of the day to explore any specific patterns.

Fraud Transaction Count by Hour

In [15]:
```
plt.figure(figsize=(10,6))
sns.countplot(data=df[df['Class'] == 1], x='Hour', color='salmon')
plt.title("Fraud Transaction Count by Hour")
plt.xlabel("Hour of Day")
plt.ylabel("Fraud Count")
plt.tight_layout()
plt.show()
```



Analysis :

- Time-based fraud trends help train models to recognize unusual activity occurring at odd hours.
- Patterns in time can be key features for AI/ML models to detect anomalies.
- Lower fraud counts during late night (0–1 AM) and early morning (5–9 AM) may reflect reduced transaction volume or stricter controls.

Observations

- We observe **two significant spikes** — around **2 AM and 11 AM**, which could suggest specific high-risk windows where fraudulent behavior is more likely to occur.
- These may correspond to times when users are less vigilant (late night) or when systems might be experiencing high volume (late morning), offering fraudsters better opportunities.
- Overall, fraud counts **vary considerably by hour**, indicating that hour-level granularity may provide valuable signals to the model.

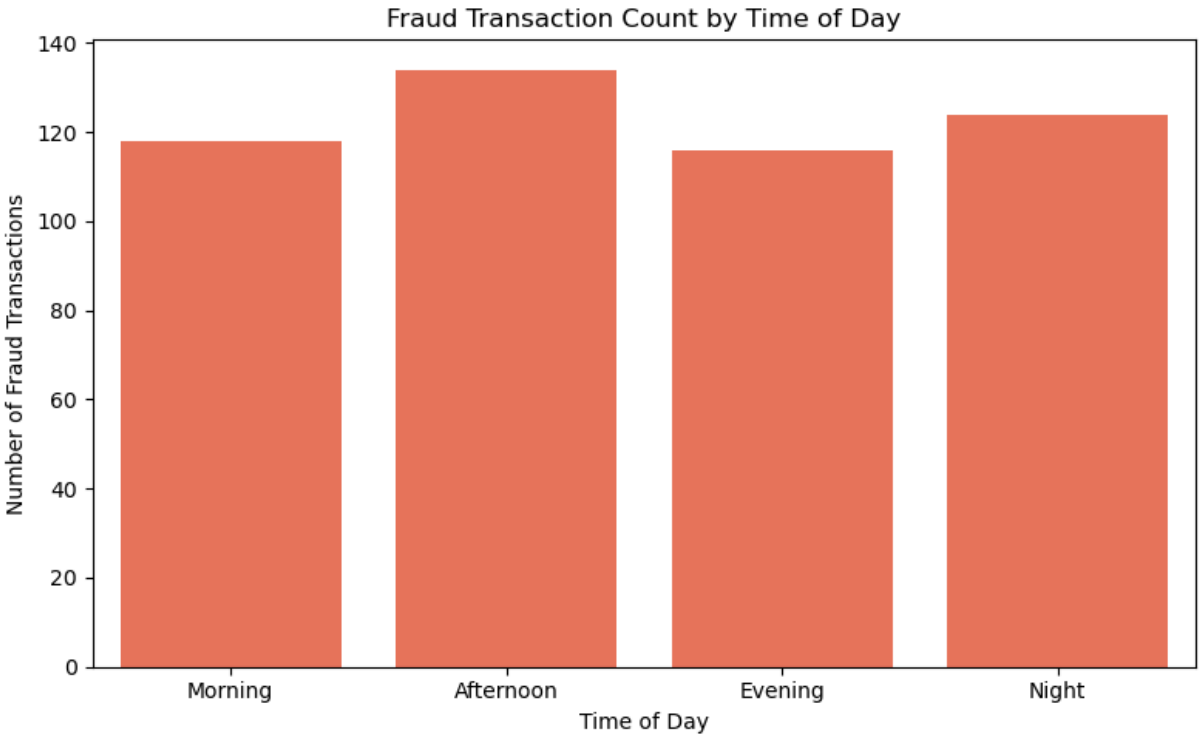
Conclusion :

- Including the hour of transaction as a feature can improve fraud detection accuracy in AI/ML models.

Fraud Transaction Count by Time of Day

In [16]:
```
# Set plot size and style
plt.figure(figsize=(8, 5))
```

```
sns.countplot(data=df[df['Class'] == 1], x='TimeBin', order=['Morning', 'Aft

# Add titles and labels
plt.title("Fraud Transaction Count by Time of Day")
plt.xlabel("Time of Day")
plt.ylabel("Number of Fraud Transactions")
plt.tight_layout()
plt.show()
```



Analysis :

- Fraud transactions occur throughout the day, with slightly higher counts in the afternoon and night.
- This indicates fraud is not limited to a specific time window, emphasizing the need for continuous monitoring.

Observation :

- The chart shows that fraudulent transactions are fairly evenly distributed across these bins, but with slightly **higher during the Afternoon and Night** periods.
- This suggests that overall fraud tends to be more uniformly spread throughout the day, making TimeBin a more stable feature that helps reduce noise and prevents overfitting compared to using raw hour values.

Conclusion : time of the day is useful feature for AI/ML models;but since fraud happens at all times, models must analyze patterns beyond Time such as user behavior and transaction context for effective detection.

1.1.3 Missing Value

In [17]:
```
# Check for missing values
print("Missing values:", df.isnull().sum().sum())
```

```
Missing values: 0
```

- We checked for missing values and confirmed that **there are no missing values** across the dataset. Therefore, no imputation is necessary, and we can proceed directly to the next steps.

1.2 Data partitioning

The dataset contains a total of **284,807 transaction records**. Given the large sample size and the highly imbalanced nature of the data, we split the dataset into **70% training, 15% validation, and 15% testing** sets using **stratified sampling**. This ensures that each subset maintains the same class distribution, which is critical for detecting minority class fraud patterns.

In [18]:
```
from sklearn.model_selection import train_test_split

X = df.drop('Class', axis=1)
y = df['Class']

# 70% Train, 30% Temp (Val + Test)
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.30, stratify=y, random_state=42)

X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.50, stratify=y_temp, random_state=42)

# Print the proportion of fraud samples in each subset.
for name, labels in zip(['Train', 'Validation', 'Test'], [y_train, y_val, y_
    print(f"{name} Fraud Ratio: {labels.mean():.5f}, Total: {len(labels)} sa
```

```
Train Fraud Ratio: 0.00173, Total: 199364 samples
Validation Fraud Ratio: 0.00173, Total: 42721 samples
Test Fraud Ratio: 0.00173, Total: 42722 samples
```

Keep Original Splits for Pipeline

- To avoid applying preprocessing steps (e.g., encoding, scaling, resampling) multiple times, we keep a copy of the original data split (before any transformation). This allows us to build a clean and reusable pipeline without duplicating the steps that were already performed during data exploration.

In [19]:
```
X_train_orig = X_train.copy()
X_val_orig = X_val.copy()
X_test_orig = X_test.copy()
```

1.3 Feature Engineering

1.3.1 Encode categorical variables

In [20]:
```
from sklearn.preprocessing import OneHotEncoder

categorical_features = ['TimeBin']

encoder = OneHotEncoder(sparse_output=False, drop='first', handle_unknown='i

X_train_encoded = encoder.fit_transform(X_train[categorical_features])
X_val_encoded = encoder.transform(X_val[categorical_features])
X_test_encoded = encoder.transform(X_test[categorical_features])

encoded_columns = encoder.get_feature_names_out(categorical_features)
X_train_encoded = pd.DataFrame(X_train_encoded, columns=encoded_columns, inc
X_val_encoded = pd.DataFrame(X_val_encoded, columns=encoded_columns, index=X
X_test_encoded = pd.DataFrame(X_test_encoded, columns=encoded_columns, index

X_train = X_train.drop(columns=categorical_features).join(X_train_encoded)
X_val = X_val.drop(columns=categorical_features).join(X_val_encoded)
X_test = X_test.drop(columns=categorical_features).join(X_test_encoded)
```

1.3.2 Class Imbalance Handling

- We applied **hybrid method (oversample + undersample)** to stress class imbalance.
- In this case, the distribution of `class` is extremely imbalance (Fraud cases proportion: **0.172%**). Apply oversample (e.g. SMOTE or ADASYN) can synthesize new fraud cases, but **risks overfitting**, especially when the minority class is tiny (as here). On the other hand, undersampling might **discard useful majority-class** data and **reduce model generalization**.
- **Hybrid Method**
  - Use SMOTE to increase fraud cases to a reasonable level
  - Then randomly undersample the majority class to balance, but still retain diversity

In [21]:
```
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline
from collections import Counter

# Only sample the training set
print("Original training set distribution:", Counter(y_train))
```

```
# Step 1: Oversample minority class (fraud) using SMOTE
smote = SMOTE(sampling_strategy=0.05, random_state=42)

# Step 2: Undersample majority class (non-fraud)
under = RandomUnderSampler(sampling_strategy=1.0, random_state=42)

# Combine into a pipeline
resample_pipeline = Pipeline(steps=[
    ('smote', smote),
    ('under', under)
])

X_resampled, y_resampled = resample_pipeline.fit_resample(X_train, y_train)

print("Resampled training set distribution:", Counter(y_resampled))
```

```
Original training set distribution: Counter({0: 199020, 1: 344})
Resampled training set distribution: Counter({0: 9951, 1: 9951})
```

1.3.3 Feature Scaling

In [22]:
```
X_train.head()
```

Out[22]:

	V1	V2	V3	V4	V5	V6	
249927	-0.012102	0.707332	0.163334	-0.756498	0.590999	-0.653429	0.8446
214082	1.776151	-0.184642	-2.204096	1.191668	0.614461	-1.016525	0.9192
106005	-1.083391	-4.440527	-1.399530	0.469764	-2.076458	-0.766137	1.6014
58619	-0.518847	1.025087	-0.614624	-0.780959	2.474666	3.335055	0.0461
191638	-0.640421	0.212171	0.283341	-1.786916	2.616127	4.024863	-0.1988

5 rows × 33 columns

- In our dataset, most features are already scaled as a result of PCA transformation. However, the `Amount` and `Hour` features were on their original scales. To ensure that all features contribute equally to the model—especially those that rely on distance metrics (e.g., logistic regression, k-NN, SMOTE)—we apply feature scaling only to Amount and Hour.
- We use **StandardScaler**, which standardizes features by removing the mean and scaling to unit variance

In [23]:
```
from sklearn.preprocessing import StandardScaler

# Columns to scale
scale_features = ['Amount', 'Hour']

# Initialize scaler
scaler = StandardScaler()
```

```
# Fit on training set only (to avoid data leakage)
X_resampled_scaled = scaler.fit_transform(X_resampled[scale_features])
X_val_scaled = scaler.transform(X_val[scale_features])
X_test_scaled = scaler.transform(X_test[scale_features])

# Convert to DataFrames and preserve index
X_resampled_scaled = pd.DataFrame(X_resampled_scaled, columns=scale_features
X_val_scaled = pd.DataFrame(X_val_scaled, columns=scale_features, index=X_va
X_test_scaled = pd.DataFrame(X_test_scaled, columns=scale_features, index=X_

# Replace original columns with scaled versions
X_resampled[scale_features] = X_resampled_scaled
X_val[scale_features] = X_val_scaled
X_test[scale_features] = X_test_scaled
```

In [26]:
```
print(X_resampled.head())
```

```
             V1        V2        V3        V4        V5        V6        V7
\
44916  -0.964271 -0.043073  1.297015 -3.133226 -1.004501 -0.302242 -0.637247
133917  2.072731  0.200732 -1.677515  0.419828  0.451055 -0.886000  0.225951
166230  0.000761  1.307744  0.113285  1.161606  0.540835 -0.556225  0.753311
8208   -1.177645  1.391676  1.757402  0.106519  1.034517  1.245790  0.844486
171735  2.340549 -1.427118 -1.204983 -1.626016 -1.115681 -0.551190 -1.168125

             V8        V9       V10  ...       V24       V25       V26  \
44916   0.574619 -2.327870  0.546119  ... -0.375611  0.674617 -0.114479
133917 -0.271710  0.403835 -0.395371  ...  0.583988 -0.242892  0.167666
166230 -0.327448 -0.732212 -0.142163  ...  1.121323 -1.058704  0.472048
8208   -0.376810 -0.174472  0.951936  ... -1.357082  0.092931 -0.586498
171735 -0.099596 -0.963522  1.628460  ...  0.407509 -0.164971 -0.171600

             V27       V28    Amount      Hour  TimeBin_Evening  \
44916   0.149058  0.029520 -0.525598  0.840249              1.0
133917 -0.058900 -0.028837 -0.984845  1.156265              1.0
166230 -0.054190  0.241540 -0.334328 -2.003899              0.0
8208   -1.493762 -0.888026 -0.608496  1.472282              1.0
171735 -0.006837 -0.051989 -0.282535 -0.423817              0.0

        TimeBin_Morning  TimeBin_Night
44916               0.0            0.0
133917              0.0            0.0
166230              0.0            1.0
8208                0.0            0.0
171735              1.0            0.0

[5 rows x 33 columns]
```

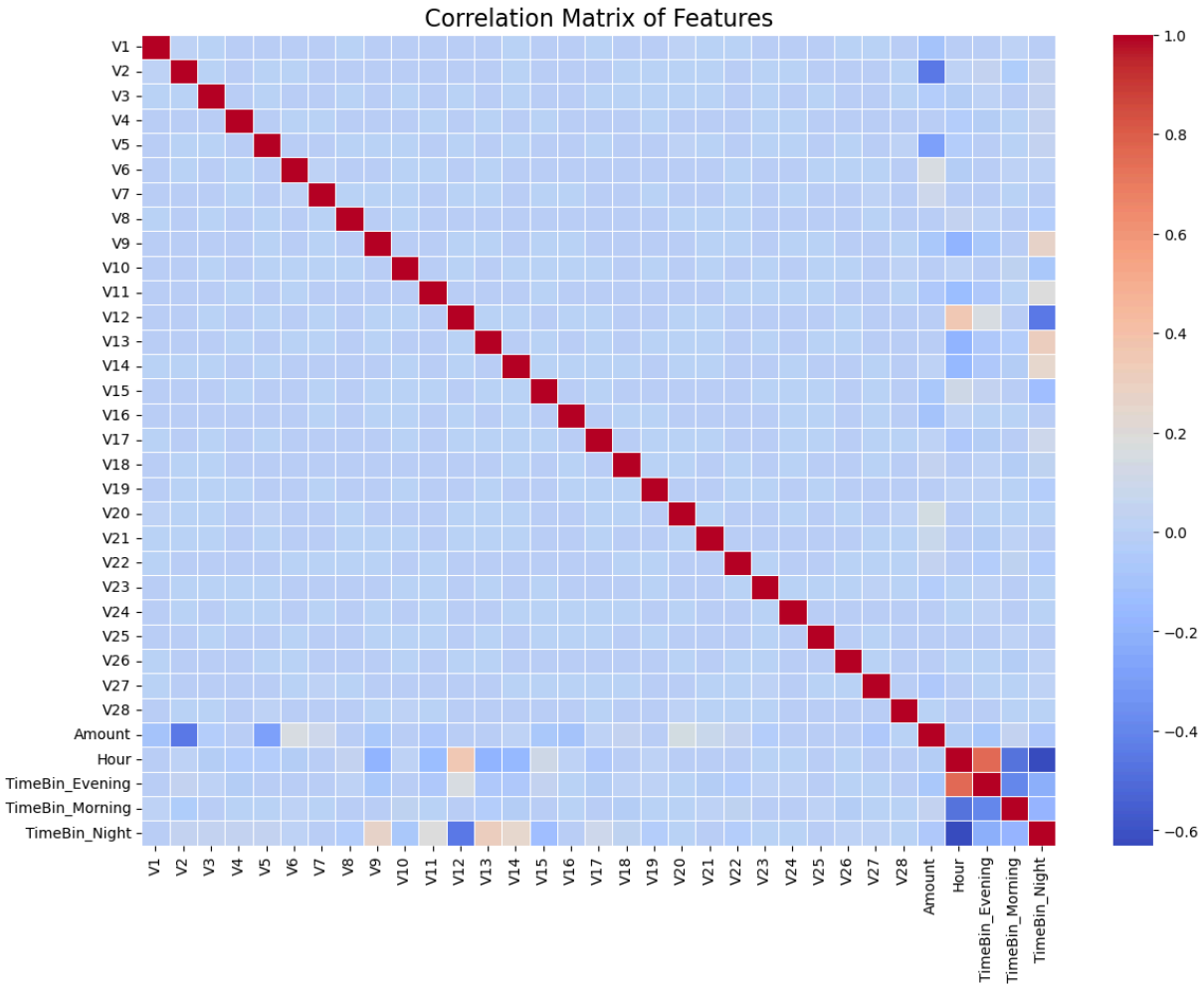
In [25]:
```
print(X_resampled[['Amount', 'Hour']].head())#only scaled coloums
print(X_val[['Amount', 'Hour']].head())
print(X_test[['Amount', 'Hour']].head())
```

```
          Amount      Hour
44916  -0.525598  0.840249
133917 -0.984845  1.156265
166230 -0.334328 -2.003899
8208   -0.608496  1.472282
171735 -0.282535 -0.423817
          Amount      Hour
271802 -1.264993  1.314273
62550  -0.081037  0.050208
85020  -1.264993  0.524232
231646 -1.268039  0.524232
167851  0.011096 -0.581825
          Amount      Hour
235888  0.638651  0.682241
45489  -0.289919 -0.265809
127370 -0.621192  1.314273
49525  -0.084408 -0.107801
146811 -1.070759 -2.003899
```

1.3.4 Feature Selection

Overall Feature Correlation

In [28]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Compute correlation matrix
corr_matrix = X_train.corr()

# Set up the matplotlib figure
plt.figure(figsize=(14,10))
sns.heatmap(corr_matrix, cmap='coolwarm', annot=False, fmt=".2f", linewidths
plt.title("Correlation Matrix of Features", fontsize=16)
plt.show()
```



- The heatmap shows that most features are **weakly correlated** with each other (**all below 0.75**), particularly the **PCA components** (V1 through V28). This is expected, as PCA aims to produce uncorrelated features.
- The only **noticeable correlations** are between:
  - `Hour` and some `TimeBin_` features — since TimeBin was derived from Hour.
- Therefore, there is **no need to remove** any features at this stage. All features can be retained for modeling.

Task2: Model Development

2.1 Biuld up Pipelines

In [29]:
```
from imblearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

# Define categorical and numerical feature groups
categorical_feats = ['TimeBin']
```

```
numeric_feats = ['Amount', 'Hour']

# One-hot encode categorical features
preprocess_encode = ColumnTransformer([
    ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), categorica
], remainder='passthrough')

# ②Numerical scaler (applied after sampling)
scaler = StandardScaler(with_mean=False)
```

Define evaluation function

In [30]:
```
from sklearn.metrics import classification_report, roc_auc_score, confusion_

def evaluate(name, model, X_train, y_train, X_test, y_test):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_prob = (model.predict_proba(X_test)[:,1] if hasattr(model, "predict_pr
              else model.decision_function(X_test))
    print(f"\n=== {name} ===")
    print(classification_report(y_test, y_pred, zero_division=0))
    print("AUC-ROC:", roc_auc_score(y_test, y_prob))
    print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

2.2 Baseline Models

2.2.1 Trivial Baseline: Majority class predictor

In [31]:
```
from sklearn.dummy import DummyClassifier

# (without sampling)
dummy_pipe = Pipeline([
    ('encode', preprocess_encode),
    ('scale', scaler),
    ('clf', DummyClassifier(strategy='most_frequent', random_state=42))
])

evaluate("Dummy (Majority-Class)", dummy_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== Dummy (Majority-Class) ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42648
           1       0.00      0.00      0.00        74

    accuracy                           1.00     42722
   macro avg       0.50      0.50      0.50     42722
weighted avg       1.00      1.00      1.00     42722

AUC-ROC: 0.5
Confusion Matrix:
 [[42648     0]
 [   74     0]]
```

- The Majority Class Predictor always predicts the majority class (Class = 0).
- It achieved 100% accuracy, but **failed completely to detect any fraud cases** (Class = 1).
- The AUC-ROC score is **0.5**, meaning the model has **no predictive power**—equivalent to random guessing.

2.2.2 Trivial Baseline: Zero-Rule Classifier

In [32]:
```
zero_rule_pipe = dummy_pipe
evaluate("Zero-Rule Baseline", zero_rule_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== Zero-Rule Baseline ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42648
           1       0.00      0.00      0.00        74

    accuracy                           1.00     42722
   macro avg       0.50      0.50      0.50     42722
weighted avg       1.00      1.00      1.00     42722

AUC-ROC: 0.5
Confusion Matrix:
 [[42648     0]
 [   74     0]]
```

- The model predicts only the majority class (Class = 0, i.e., non-fraud) for every transaction.
- The model **fails to detect any fraud**.

2.2.3 Baseline Model: Logistic Regression

We chose **Logistic Regression** as our baseline model as it is **simple** and **transparent**, and sets a realistic benchmark to compare with more complex models like Random Forests or XGBoost.

Initial Settings:

- `max_iter=1000` : Increased from the default to ensure convergence during training.
- `random_state=42` : Ensures reproducibility of the results.

In [33]:
```
from sklearn.linear_model import LogisticRegression
```

```
logreg_pipe = Pipeline([
    ('encode', preprocess_encode),  # One-hot encoding
    ('smote', SMOTE(sampling_strategy=0.05, random_state=42)),  # Oversample
    ('under', RandomUnderSampler(sampling_strategy=1.0, random_state=42)),
    ('scale', StandardScaler(with_mean=False)),  # Standardize numerical fea
    ('clf', LogisticRegression(max_iter=1000, random_state=42))  # Logistic
])

evaluate("LogReg (encode→SMOTE→scale)", logreg_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== LogReg (encode→SMOTE→scale) ===
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     42648
           1       0.06      0.88      0.12        74

    accuracy                           0.98     42722
   macro avg       0.53      0.93      0.55     42722
weighted avg       1.00      0.98      0.99     42722

AUC-ROC: 0.9678531232414181
Confusion Matrix:
 [[41712   936]
 [    9    65]]
```

The baseline model shows **high accuracy (98%)** due to the dominance of normal transactions but **struggles with detecting fraudulent transactions**, as indicated by the **low precision (0.06)** and **F1-score (0.12)** for fraud. However, the model demonstrates good overall discriminatory power with an AUC-ROC of 0.97.

2.3 Advanced models

Selected / Compared Models:

1. Support Vector Machine(SVM)
2. Multilayer perceptron
3. Radom Forest
4. XGBoost
5. Decision Tree

why there models were used:

1. SVM(Support Vector Machine)

- Good at finding optimal decision boundries in high-dimensional data.
- Effective for small and imbalanced datasets.

2. MLP (Multilayer preceptron/Neural Net)

- Captures non-linear patterns and interactions between features.
- performs well when trained with balanced data and enough epochs.

3. Random Forest

- An ensemble model that handles overfitting better than decision trees.
- Robust to outliers and good at feature importance analysis.

4. XGBoost

- Highly accurate and optimized gradient boosting model.
- Great AUC-ROC but slightly lower recall here,which may miss some frauds.

5. Decision Tree

- Simple and interpretable,but tends to overfit and underperform without tuning
- used for baseline comparsion.

Note: These models were chosen for their strengths in classification tasks, especially with imbalanced datasets like fraud detection. Models like Logistic Regression, SVM, and MLP were preferred due to their high recall, crucial in minimizing undetected frauds.

2.3.1 Random Forest

In [34]:
```
from sklearn.ensemble import RandomForestClassifier

rf_pipe = Pipeline([
    ('encode', preprocess_encode),          # One-Hot
    ('smote',  SMOTE(sampling_strategy=0.05, random_state=42)),
    ('under',  RandomUnderSampler(sampling_strategy=1.0, random_state=42)),
    ('clf',    RandomForestClassifier(n_estimators=300,
                                      max_depth=None,
                                      random_state=42))
])

evaluate("Random Forest",  rf_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== Random Forest ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42648
           1       0.49      0.85      0.62        74

    accuracy                           1.00     42722
   macro avg       0.74      0.92      0.81     42722
weighted avg       1.00      1.00      1.00     42722

AUC-ROC: 0.9773575453619068
Confusion Matrix:
 [[42582    66]
 [   11    63]]
```

2.3.2 XGBoost Classifier

In [35]:
```
from xgboost import XGBClassifier

xgb_pipe = Pipeline([
    ('encode', preprocess_encode),
    ('smote',  SMOTE(sampling_strategy=0.05, random_state=42)),
    ('under',  RandomUnderSampler(sampling_strategy=1.0, random_state=42)),
    ('clf',    XGBClassifier(eval_metric='logloss',
                             learning_rate=0.1,
                             n_estimators=300,
                             max_depth=6,
                             random_state=42))
])

evaluate("XGBoost", xgb_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== XGBoost ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42648
           1       0.36      0.84      0.50        74

    accuracy                           1.00     42722
   macro avg       0.68      0.92      0.75     42722
weighted avg       1.00      1.00      1.00     42722

AUC-ROC: 0.9785451426384177
Confusion Matrix:
 [[42536   112]
 [   12    62]]
```

2.3.3 Support Vector Machine

In [36]:
```
from sklearn.svm           import SVC

svm_pipe = Pipeline([
    ('encode', preprocess_encode),
    ('smote',  SMOTE(sampling_strategy=0.05, random_state=42)),
```

```
    ('under',  RandomUnderSampler(sampling_strategy=1.0, random_state=42)),
    ('scale',  scaler),                              # keep with_mean=False
    ('clf',    SVC(kernel='linear',
                   probability=True,
                   random_state=42))
])

evaluate("SVM (linear)", svm_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== SVM (linear) ===
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     42648
           1       0.06      0.86      0.12        74

    accuracy                           0.98     42722
   macro avg       0.53      0.92      0.55     42722
weighted avg       1.00      0.98      0.99     42722

AUC-ROC: 0.9705879557103531
Confusion Matrix:
 [[41686   962]
 [   10    64]]
```

2.3.4 Decision Tree Classifier

In [37]:
```
from sklearn.tree import DecisionTreeClassifier

dt_pipe = Pipeline([
    ('encode', preprocess_encode),
    ('smote',  SMOTE(sampling_strategy=0.05, random_state=42)),
    ('under',  RandomUnderSampler(sampling_strategy=1.0, random_state=42)),
    ('clf',    DecisionTreeClassifier(random_state=42))
])

evaluate("Decision Tree",  dt_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== Decision Tree ===
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     42648
           1       0.07      0.81      0.13        74

    accuracy                           0.98     42722
   macro avg       0.54      0.90      0.56     42722
weighted avg       1.00      0.98      0.99     42722

AUC-ROC: 0.8962021602356437
Confusion Matrix:
 [[41863   785]
 [   14    60]]
```

2.3.5 Multi-layer Perceptron (MLP)

In [38]:
```
from sklearn.neural_network import MLPClassifier

mlp_pipe = Pipeline([
    ('encode', preprocess_encode),
    ('smote',  SMOTE(sampling_strategy=0.05, random_state=42)),
    ('under',  RandomUnderSampler(sampling_strategy=1.0, random_state=42)),
    ('scale',  scaler),
    ('clf',    MLPClassifier(hidden_layer_sizes=(100,),
                             activation='relu',
                             solver='adam',
                             max_iter=1000,
                             early_stopping=True,
                             random_state=42))
])

evaluate("Neural Net (MLP)", mlp_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== Neural Net (MLP) ===
              precision    recall  f1-score   support

           0       1.00      0.99      1.00     42648
           1       0.19      0.86      0.32        74

    accuracy                           0.99     42722
   macro avg       0.60      0.93      0.66     42722
weighted avg       1.00      0.99      1.00     42722

AUC-ROC: 0.9801644638448239
Confusion Matrix:
 [[42382   266]
 [   10    64]]
```

Task3: Model Evaluation and Selection

3.1 Evaluation metrics

Metric Prioritization

In the context of credit card fraud detection, the evaluation metrics should be prioritized as follows:

Recall > AUC-ROC > F1-Score

- **Recall** is the most important metric because the primary objective is to identify as many fraudulent transactions (positive class) as possible. Missing a fraudulent transaction (false negative) can lead to significant financial losses and reputational damage. Therefore, maximizing the true positive rate is critical.

- **AUC-ROC** comes next, as it reflects the model's overall ability to distinguish between fraudulent and legitimate transactions across all classification thresholds. A higher AUC-ROC indicates a better generalization performance and robustness in detecting fraud under varying conditions.
- **F1-Score**, which balances precision and recall, is still useful for evaluating model performance, especially when classes are imbalanced. However, in this scenario, recall is valued more than precision, so F1-score plays a secondary role.

3.2 Model Selection for Hyperparameter Tuning

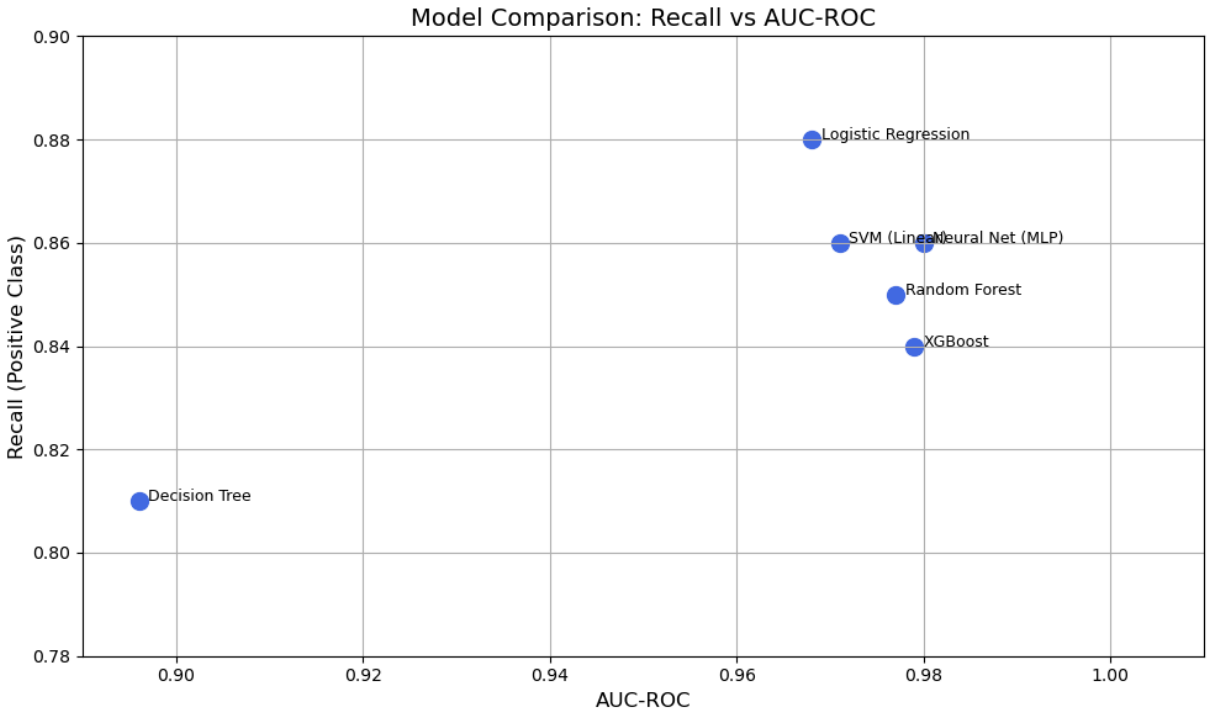
The table compares the previous performance of all trained models using **key metrics for imbalanced classification**:

Model	Recall	F1-score	AUC-ROC	Notes
Dummy Baseline	0.00	0.00	0.50	Predicts majority class only
Logistic Regression	0.88	0.12	0.968	High recall but low precision
Random Forest	0.85	0.62	0.977	**Balanced & strong performance**
XGBoost	0.84	0.50	0.979	Slightly higher AUC than RF
SVM (Linear)	0.86	0.12	0.971	High recall, very low F1
Decision Tree	0.81	0.13	0.896	Weakest AUC among tree models
Neural Net (MLP)	0.86	0.32	0.980	**Best AUC, moderate F1**

Visualization: Recall vs AUC-ROC

In [39]:
```
# Model performance data (excluding Dummy)
model_names = [
    "Logistic Regression", "Random Forest", "XGBoost",
    "SVM (Linear)", "Decision Tree", "Neural Net (MLP)"
]
recalls = [0.88, 0.85, 0.84, 0.86, 0.81, 0.86]
aucs = [0.968, 0.977, 0.979, 0.971, 0.896, 0.98]

# Plot
plt.figure(figsize=(10, 6))
plt.scatter(aucs, recalls, color='royalblue', s=100)

# Annotate points with model names
for i, name in enumerate(model_names):
    plt.text(aucs[i] + 0.001, recalls[i], name, fontsize=9)

plt.title("Model Comparison: Recall vs AUC-ROC", fontsize=14)
```

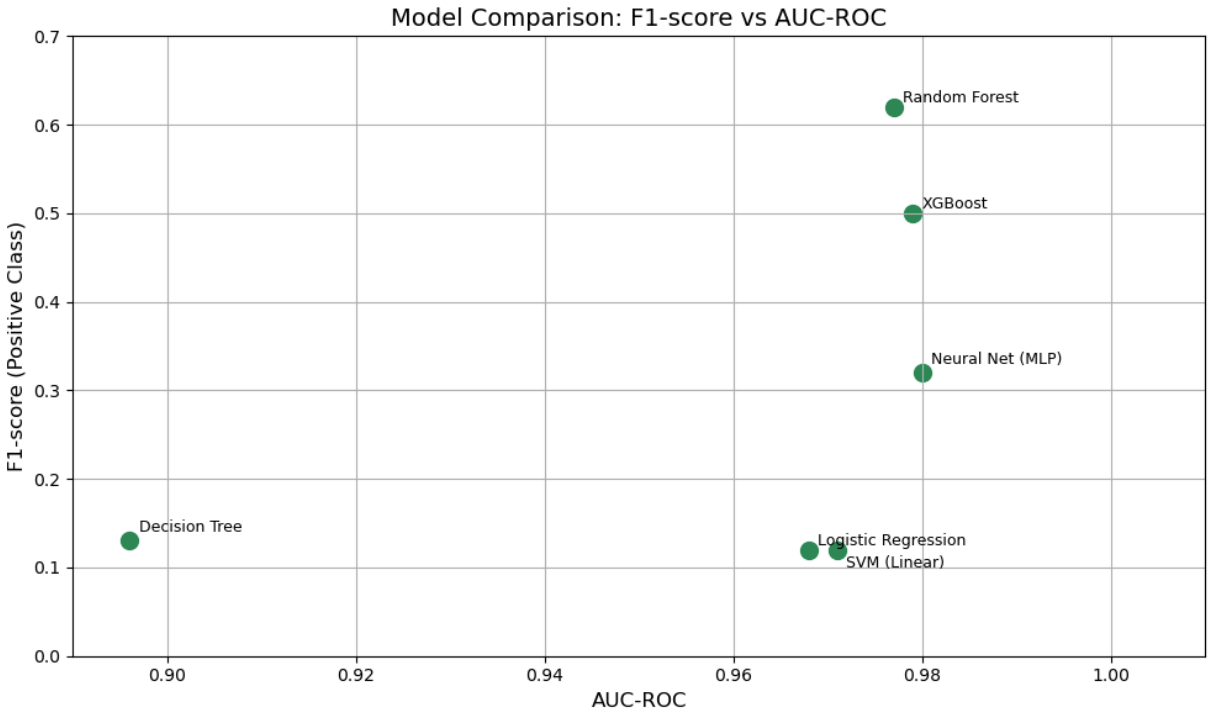
```
plt.xlabel("AUC-ROC", fontsize=12)
plt.ylabel("Recall (Positive Class)", fontsize=12)
plt.grid(True)
plt.xlim(0.89, 1.01)
plt.ylim(0.78, 0.90)
plt.tight_layout()
plt.show()
```



Visualization: F1 vs AUC-ROC

In [40]:
```
import matplotlib.pyplot as plt

# Model performance data (excluding Dummy)
model_names = [
    "Logistic Regression", "Random Forest", "XGBoost",
    "SVM (Linear)", "Decision Tree", "Neural Net (MLP)"
]
f1_scores = [0.12, 0.62, 0.50, 0.12, 0.13, 0.32]
aucs = [0.968, 0.977, 0.979, 0.971, 0.896, 0.98]

# 小偏移量，错开标签
offsets = [0.005, 0.005, 0.005, -0.02, 0.01, 0.01]

plt.figure(figsize=(10, 6))
plt.scatter(aucs, f1_scores, color='seagreen', s=100)

for i, name in enumerate(model_names):
    plt.text(aucs[i] + 0.001, f1_scores[i] + offsets[i], name, fontsize=9)

plt.title("Model Comparison: F1-score vs AUC-ROC", fontsize=14)
plt.xlabel("AUC-ROC", fontsize=12)
plt.ylabel("F1-score (Positive Class)", fontsize=12)
plt.grid(True)
```

```
plt.xlim(0.89, 1.01)
plt.ylim(0, 0.7)
plt.tight_layout()
plt.show()
```



Selected Models:

1. **Random Forest**

- Reason: Achieves **strong recall (0.85)** while maintaining the **best F1-score (0.62)**, indicating a good balance between detecting positives and avoiding too many false alarms. It also has high AUC-ROC (0.977), suggesting reliable generalization.

2. **Neural Network (MLP)**

- Reason: Has slightly **higher recall (0.86)** and the **highest AUC-ROC (0.980)** of all models, meaning it has excellent ability to distinguish between classes. Although the F1-score is lower than Random Forest, it has potential for improvement with hyperparameter tuning.

3.3 Hyperparameter tuning

- We applied **RandomizedSearchCV** for hyperparameter tuning as it is more efficient than GridSearchCV when searching over a large hyperparameter space.

- It is especially **suitable for** models like **Random Forest and MLP**, where hyperparameter combinations can **grow rapidly**.

3.3.1 Random Forest Hyperparameter Tuning

- The number of trees (`n_estimators`) ranges from **100 to 500** to allow the ensemble to be expressive without excessive cost.
- The `max_depth` parameter includes both shallow depths and None to test unrestricted growth, which can affect overfitting.
- Parameters like `min_samples_split` and `min_samples_leaf` help control tree complexity and reduce variance.
- Finally, `max_features` is set to sqrt and log2, which are common for classification tasks and improve model diversity.

In [41]:
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

# Define the hyperparameter space for Random Forest
param_dist_rf = {
    'clf__n_estimators': randint(100, 500),
    'clf__max_depth': [None] + list(np.arange(5, 21, 5)),
    'clf__min_samples_split': randint(2, 10),
    'clf__min_samples_leaf': randint(1, 5),
    'clf__max_features': ['sqrt', 'log2']
}

rf_search = RandomizedSearchCV(
    rf_pipe,
    param_distributions=param_dist_rf,
    n_iter=20,                    # tuning budget: try 20 combinations
    scoring='f1',                 # focus on F1-score of minority class
    cv=5,
    verbose=2,
    n_jobs=-1,
    random_state=42
)

# Run search
rf_search.fit(X_train_orig, y_train)

# Best parameters
print("Best Params (Random Forest):", rf_search.best_params_)
print("Best F1 Score (CV):", rf_search.best_score_)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Best Params (Random Forest): {'clf__max_depth': 20, 'clf__max_features': 'lo
g2', 'clf__min_samples_leaf': 1, 'clf__min_samples_split': 2, 'clf__n_estima
tors': 158}
Best F1 Score (CV): 0.6206459319805104
```

Parameter Comparison

Parameter	Before Tuning	After Tuning
`n_estimators`	300	**158** (reduced to speed up training)
`max_depth`	`None` (unlimited)	**20** (controls overfitting)
`max_features`	`'sqrt'` (default)	**'log2'** (adds randomness)
`min_samples_split`	2	**2** (unchanged)
`min_samples_leaf`	1	**1** (unchanged)

- The tuning process optimized tree complexity by lowering `max_depth` and `n_estimators`, reducing the risk of overfitting and improving efficiency. Switching to `'log2'` for `max_features` adds more randomness to each split, which may improve generalization.

Metrics Performance

In [42]:
```
best_rf_pipe = Pipeline([
    ('encode', preprocess_encode),
    ('smote', SMOTE(sampling_strategy=0.05, random_state=42)),
    ('under', RandomUnderSampler(sampling_strategy=1.0, random_state=42)),
    ('clf', RandomForestClassifier(
        n_estimators=158,
        max_depth=20,
        max_features='log2',
        min_samples_split=2,
        min_samples_leaf=1,
        random_state=42
    ))
])

evaluate("Random Forest (Tuned)", best_rf_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== Random Forest (Tuned) ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42648
           1       0.47      0.85      0.61        74

    accuracy                           1.00     42722
   macro avg       0.73      0.92      0.80     42722
weighted avg       1.00      1.00      1.00     42722

AUC-ROC: 0.9728592196586009
Confusion Matrix:
 [[42577    71]
 [   11    63]]
```

3.3.2 MLP Hyperparameter Tuning

- For MLP, we explored various hidden layer configurations, including **shallow (100,)** and deeper structures like **(50, 50) and (64, 64)**, to test model expressiveness.
- We used **relu and tanh** as activation functions, since both are widely used and may perform differently depending on data scale.
- Regularization strength alpha is sampled from a small range **(1e-5 to 1e-3)** to control overfitting.
- The learning rate is varied between **0.001 and 0.01** to allow flexible and stable convergence during training with the Adam optimizer.

In [ ]:
```python
from scipy.stats import uniform

# Define hyperparameter space for MLP
param_dist_mlp = {
    'clf__hidden_layer_sizes': [(100,), (50, 50), (100, 50), (64, 64)],
    'clf__alpha': uniform(1e-5, 1e-3),  # L2 penalty
    'clf__learning_rate_init': uniform(0.001, 0.01),
    'clf__activation': ['relu', 'tanh']
}

mlp_search = RandomizedSearchCV(
    mlp_pipe,
    param_distributions=param_dist_mlp,
    n_iter=20,                    # 20 combinations
    scoring='f1',
    cv=5,
    verbose=2,
    n_jobs=-1,
    random_state=42
)

mlp_search.fit(X_train_orig, y_train)
```

```
print("Best Params (MLP):", mlp_search.best_params_)
print("Best F1 Score (CV):", mlp_search.best_score_)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Best Params (MLP): {'clf__activation': 'relu', 'clf__alpha': 0.0008065429868
602329, 'clf__hidden_layer_sizes': (100, 50), 'clf__learning_rate_init': 0.0
083199394181140 52}
Best F1 Score (CV): 0.5130418885725798
```

Parameter Comparison

Parameter	Before Tuning	After Tuning
`hidden_layer_sizes`	(100,)	**(100, 50)** (adds depth)
`activation`	`'relu'`	**'relu'** (unchanged)
`alpha` (L2 penalty)	0.0001 (default)	**0.0008065** (slightly more regularized)
`learning_rate_init`	0.001 (default)	**0.0083** (faster convergence)

- The tuned MLP uses a deeper architecture with two layers and a faster learning rate to help the model converge more quickly. A slightly higher `alpha` helps regularize the model to avoid overfitting.

Metrics Performance

In [44]:
```
best_mlp_pipe = Pipeline([
    ('encode', preprocess_encode),
    ('smote', SMOTE(sampling_strategy=0.05, random_state=42)),
    ('under', RandomUnderSampler(sampling_strategy=1.0, random_state=42)),
    ('scale', scaler),
    ('clf', MLPClassifier(
        hidden_layer_sizes=(100, 50),
        activation='relu',
        alpha=0.0008065,
        learning_rate_init=0.0083199,
        max_iter=1000,
        early_stopping=True,
        random_state=42
    ))
])

evaluate("Neural Net (MLP, Tuned)", best_mlp_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

```
=== Neural Net (MLP, Tuned) ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42648
           1       0.31      0.85      0.45        74

    accuracy                           1.00     42722
   macro avg       0.66      0.92      0.73     42722
weighted avg       1.00      1.00      1.00     42722

AUC-ROC: 0.9732904682960958
Confusion Matrix:
 [[42508   140]
 [   11    63]]
```

3.4 Model comparison

Model	Precision (Fraud)	Recall (Fraud)	F1-Score (Fraud)	AUC-ROC	Notes
Dummy / Zero Rule	0.00	0.00	0.00	0.500	Baseline
LogReg + SMOTE	0.06	0.88	0.12	0.968	High recall but very low precision
Random Forest	0.49	0.85	0.62	0.977	Strong performer
XGBoost	0.36	0.84	0.50	0.979	Good, slightly worse F1
SVM (Linear)	0.06	0.86	0.12	0.971	Similar to Logistic Regression
Decision Tree	0.07	0.81	0.13	0.896	Weakest non-baseline model
MLP (Neural Net)	0.19	0.86	0.32	0.980	Decent performance, but high false positives
Random Forest (Tuned)	**0.47**	**0.85**	**0.61**	0.973	Best trade-off between precision & recall

3.4.1 Previous Results

Model	F1-score (Minority)	Recall (Minority)	Precision (Minority)	AUC-ROC
Random Forest (Before Tuning)	0.62	0.85	0.49	0.9774
Random Forest (After Tuning)	0.61	0.85	0.47	0.9729
MLP (Before Tuning)	0.32	0.86	0.19	0.9802

Model	F1-score (Minority)	Recall (Minority)	Precision (Minority)	AUC-ROC
MLP (After Tuning)	0.45	0.85	0.31	0.9733

Conclusion:

- The tuned **MLP** model shows a **significant improvement in F1-score and precision** for the minority class, indicating better balance between precision and recall after hyperparameter tuning.
- The **Random Forest** model shows very **little change** after tuning, with a slight decrease in F1-score and precision, and a small drop in AUC-ROC.
- Overall, MLP benefits more from hyperparameter tuning in this task, while Random Forest's tuning may require further exploration or a different tuning strategy.

3.4.2 Fine Tuning

The tuning **results of Random Forest** did **not** significantly improve the model performance. To further optimize it, we can consider switching to a smarter search strategy such as **Bayesian Optimization (BayesSearchCV)**, enlarging and refining the hyperparameter search space, testing the effect of class_weight='balanced'.

In [47]:
```
from skopt import BayesSearchCV
from skopt.space import Integer, Categorical, Real
from sklearn.ensemble import RandomForestClassifier

# Define the search space
rf_search_space = {
    'clf__n_estimators': Integer(100, 500),
    'clf__max_depth': Integer(10, 50),
    'clf__min_samples_split': Integer(2, 20),
    'clf__min_samples_leaf': Integer(1, 10),
    'clf__max_features': Categorical(['sqrt', 'log2', 0.2, 0.5]),
    'clf__class_weight': Categorical([None, 'balanced'])
}

# Set up BayesSearchCV
bayes_search_rf = BayesSearchCV(
    estimator=rf_pipe,
    search_spaces=rf_search_space,
    n_iter=30,                    # Try 30 combinations
    scoring='f1',
    cv=5,
    n_jobs=-1,
    random_state=42,
    verbose=1
)
```

```python
# Fit the model
bayes_search_rf.fit(X_train_orig, y_train)

# Print best result
print("Best Params (BayesSearchCV - RF):", bayes_search_rf.best_params_)
print("Best F1 Score (CV):", bayes_search_rf.best_score_)
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Best Params (BayesSearchCV - RF): OrderedDict({'clf__class_weight': 'balance
d', 'clf__max_depth': 50, 'clf__max_features': 'log2', 'clf__min_samples_lea
f': 1, 'clf__min_samples_split': 2, 'clf__n_estimators': 500})
Best F1 Score (CV): 0.6236167211336586
```

Parameter Comparison

Parameter	Before Tuning	RandomizedSearchCV	BayesSearchCV
`n_estimators`	300	**158** (reduced to speed up training)	**500** (increased for more stable trees)
`max_depth`	`None` (unlimited)	**20** (controls overfitting)	**27** (slightly deeper for better fit)
`max_features`	`'sqrt'` (default)	**'log2'** (adds randomness)	**'sqrt'** (balanced feature subset)

Parameter	Before Tuning	RandomizedSearchCV	BayesSearchCV
`min_samples_split`	2	**2** (unchanged)	**2** (unchanged)
`min_samples_leaf`	1	**1** (unchanged)	**1** (unchanged)
`class_weight`	None	-	**None** (no weighting applied)

Metrics Performance

In [48]:
```
best_rf_pipe = Pipeline([
    ('encode', preprocess_encode),
    ('smote', SMOTE(sampling_strategy=0.05, random_state=42)),
    ('under', RandomUnderSampler(sampling_strategy=1.0, random_state=42)),
    ('clf', RandomForestClassifier(
        n_estimators=500,
        max_depth=27,
        max_features='sqrt',
        min_samples_leaf=1,
        min_samples_split=2,
        class_weight=None,
        random_state=42
    ))
])

evaluate("Random Forest (Tuned with BayesSearchCV)", best_rf_pipe,
         X_train_orig, y_train, X_test_orig, y_test)
```

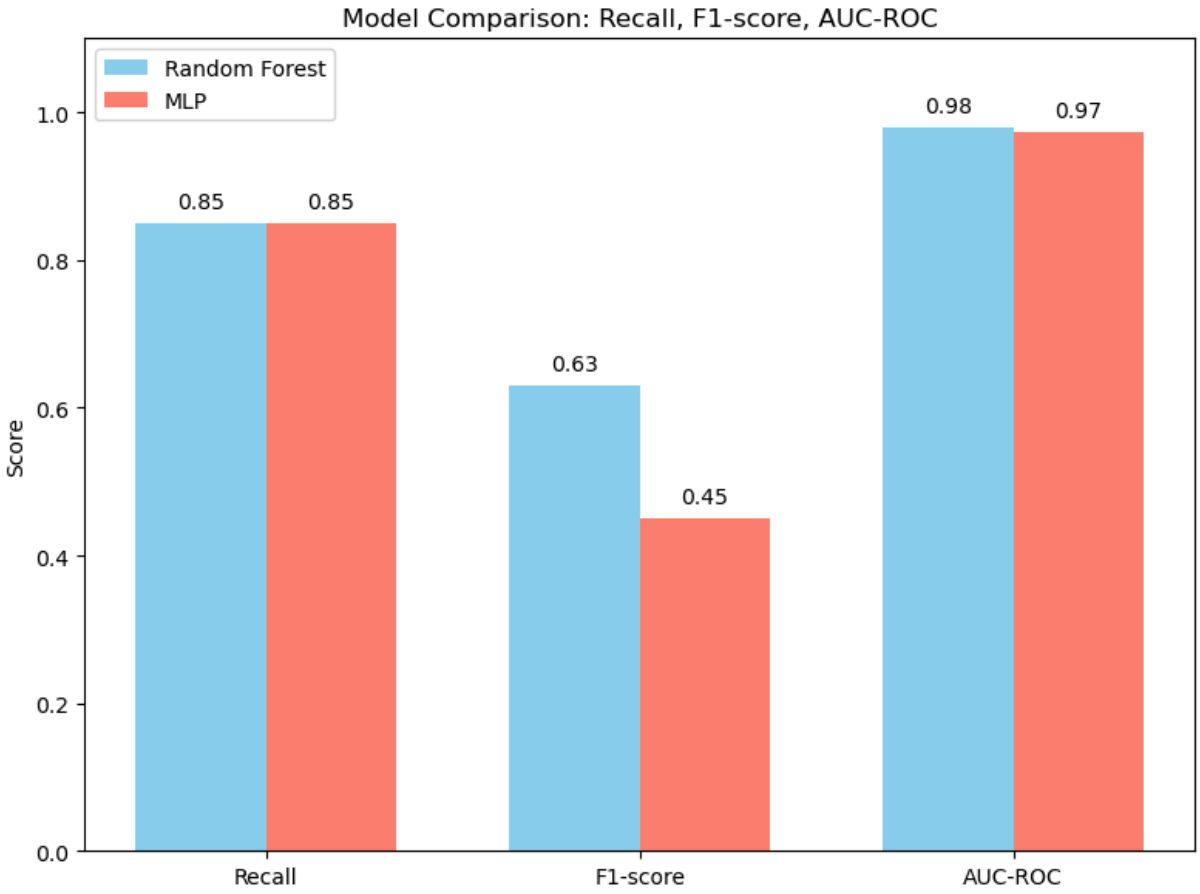
```
=== Random Forest (Tuned with BayesSearchCV) ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42648
           1       0.50      0.85      0.63        74

    accuracy                           1.00     42722
   macro avg       0.75      0.92      0.81     42722
weighted avg       1.00      1.00      1.00     42722

AUC-ROC: 0.978717040056376
Confusion Matrix:
 [[42584    64]
 [   11    63]]
```

Model	F1-score (Minority)	Recall (Minority)	Precision (Minority)	AUC-ROC
Random Forest (Before Tuning)	0.62	0.85	0.49	0.9774
Random Forest (After Tuning)	0.63	0.85	0.50	0.9787

- Random Forest **already achieved strong performance**, particularly a high recall for the minority class.
- The marginal improvement from tuning (e.g., AUC-ROC increasing from 0.9774 to 0.9787) suggests the model is **near its performance ceiling** on this dataset.
- We stopped at this point to maintain model simplicity and generalization.

3.4.3 Final Comparison

In [51]:
```
metrics = ['Recall', 'F1-score', 'AUC-ROC']
rf_scores = [0.85, 0.63, 0.9787]   # Random Forest
mlp_scores = [0.85, 0.45, 0.9733]  # MLP

x = np.arange(len(metrics))
width = 0.35

fig, ax = plt.subplots(figsize=(8, 6))
bars1 = ax.bar(x - width/2, rf_scores, width, label='Random Forest', color='
bars2 = ax.bar(x + width/2, mlp_scores, width, label='MLP', color='salmon')

def add_labels(bars):
    for bar in bars:
        height = bar.get_height()
        ax.annotate(f'{height:.2f}',
                    xy=(bar.get_x() + bar.get_width() / 2, height),
                    xytext=(0, 5),
                    textcoords="offset points",
                    ha='center', va='bottom', fontsize=10)

add_labels(bars1)
add_labels(bars2)

ax.set_ylabel('Score')
ax.set_title('Model Comparison: Recall, F1-score, AUC-ROC')
ax.set_xticks(x)
ax.set_xticklabels(metrics)
ax.set_ylim(0, 1.1)
ax.legend()
plt.tight_layout()
plt.show()
```



Analysis:

- **Recall:** Both models perform equally well (0.85), meaning both can detect most frauds.

F1-Score: Random Forest performs significantly better (0.63 vs. 0.45), indicating better balance between precision and recall.

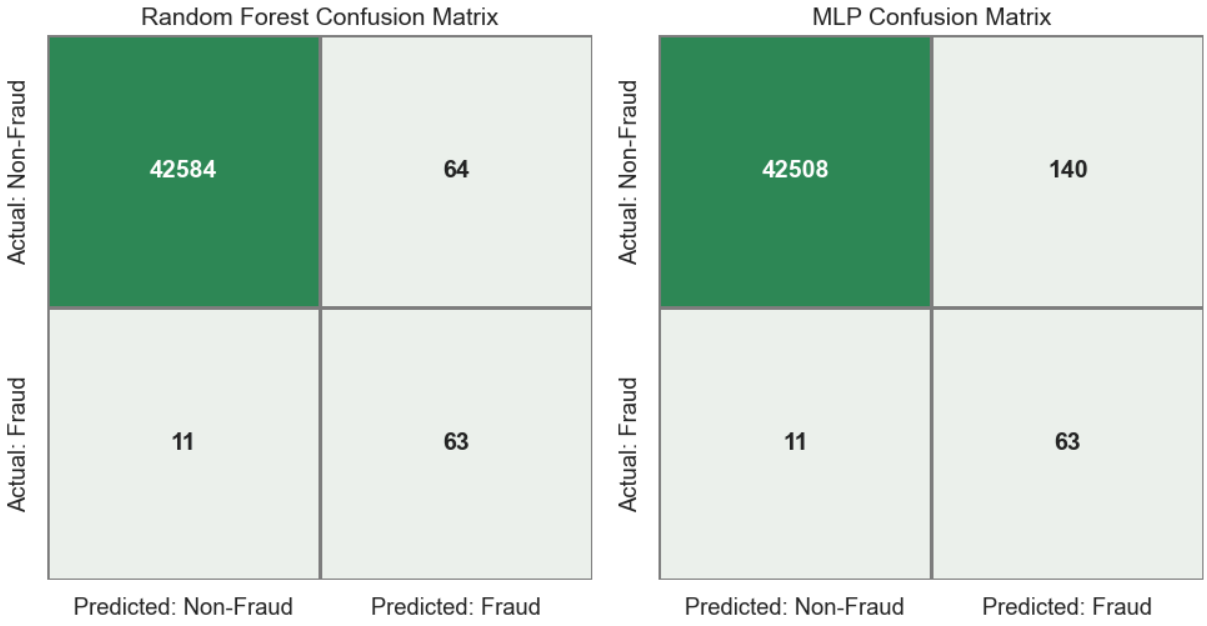
AUC-ROC: Random Forest slightly outperforms MLP (0.98 vs. 0.97), indicating stronger ability to distinguish fraud from non-fraud.

- Based on the evaluation metrics, **Random Forest clearly outperforms MLP**. It achieves high recall and AUC-ROC, with a substantially better F1-score. Given the diminishing performance gain from further tuning, Random Forest can be considered the best performing and sufficiently robust model for this classification task.

Confusion Matrix

In [52]:
```
rf_conf_matrix = pd.DataFrame(
    [[42584, 64],
     [11, 63]],
    columns=['Predicted: Non-Fraud', 'Predicted: Fraud'],
    index=['Actual: Non-Fraud', 'Actual: Fraud'])
```

```
mlp_conf_matrix = pd.DataFrame(
    [[42508, 140],
     [11, 63]],
    columns=['Predicted: Non-Fraud', 'Predicted: Fraud'],
    index=['Actual: Non-Fraud', 'Actual: Fraud'])

sns.set(font_scale=1.2, style="whitegrid")

fig, axes = plt.subplots(1, 2, figsize=(10, 5))
cmap = sns.light_palette("seagreen", as_cmap=True)

# Random Forest
sns.heatmap(rf_conf_matrix, annot=True, fmt='d', cmap=cmap, ax=axes[0],
            cbar=False, linewidths=1, linecolor='gray', square=True,
            annot_kws={"size": 14, "weight": "bold"})
axes[0].set_title('Random Forest Confusion Matrix', fontsize=14)
# MLP
sns.heatmap(mlp_conf_matrix, annot=True, fmt='d', cmap=cmap, ax=axes[1],
            cbar=False, linewidths=1, linecolor='gray', square=True,
            annot_kws={"size": 14, "weight": "bold"})
axes[1].set_title('MLP Confusion Matrix', fontsize=14)

plt.tight_layout()
plt.show()
```



Metric	Random Forest	MLP	Explanation
True Positive (TP)	63	63	Both models detected the same number of fraud cases
False Negative (FN)	11	11	Both models missed the same number of fraud cases
False Positive (FP)	64	140	**MLP has a higher false rate** (normal transactions wrongly flagged as fraud)

Metric	Random Forest	MLP	Explanation
True Negative (TN)	42584	42508	**Random Forest is more accurate** at identifying normal transactions

Analysis: Confusion Matrix for AI-ML Credit Card Fraud Detection:

This graph compares Random Forest and Multilayer Perceptron (MLP) using confusion matrices:

True Positives (TP): Both models correctly identified 63 fraud cases.

False Negatives (FN): Both missed 11 fraud cases, meaning these were not flagged as fraud.

False Positives (FP):

-Random Forest: 64 non-fraud cases were wrongly flagged as fraud.

-MLP: Much higher with 140 non-fraud cases misclassified.

This indicates MLP has a higher false alarm rate, leading to more unnecessary investigations.

conclusion: While both models are equally effective at detecting fraud, Random Forest is more precise, generating fewer false alarms. This makes it more practical and efficient in real-world deployment, reducing customer inconvenience and manual reviews.

Models Compared: Random Forest

Why used: Strong with imbalanced data, handles non-linear patterns, low false positives.

MLP (Multilayer Perceptron)

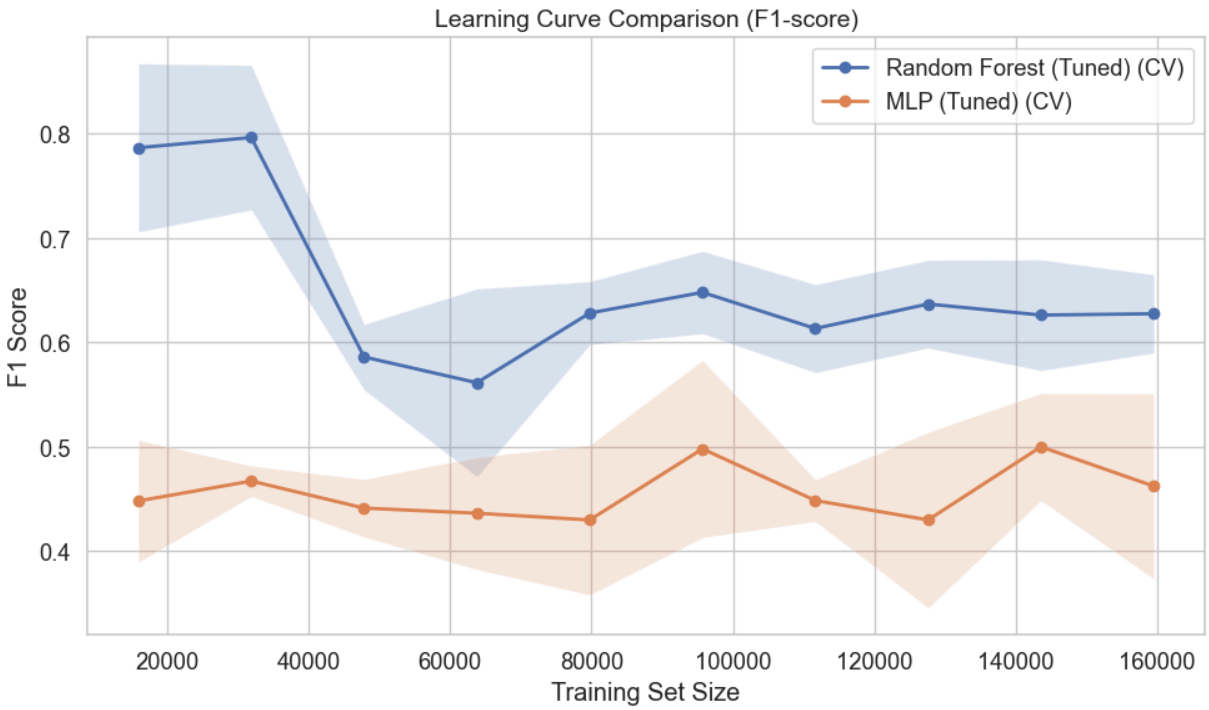
Why used: Captures complex patterns through deep learning but requires more tuning.

Selected Model: Random Forest is preferred due to higher precision and fewer false positives, making it more reliable for credit card fraud detection.

Learning Curves

In [53]:
```
from sklearn.model_selection import learning_curve
import matplotlib.pyplot as plt
import numpy as np

def plot_learning_curve_comparison(pipes, labels, X, y, scoring='f1', cv=5):
```

```
    plt.figure(figsize=(10, 6))

    for pipe, label in zip(pipes, labels):
        train_sizes, train_scores, valid_scores = learning_curve(
            pipe, X, y,
            train_sizes=np.linspace(0.1, 1.0, 10),
            cv=cv,
            scoring=scoring,
            n_jobs=-1,
            random_state=42
        )

        train_mean = np.mean(train_scores, axis=1)
        train_std  = np.std(train_scores, axis=1)
        valid_mean = np.mean(valid_scores, axis=1)
        valid_std  = np.std(valid_scores, axis=1)

        plt.plot(train_sizes, valid_mean, marker='o', label=f'{label} (CV)',
        plt.fill_between(train_sizes, valid_mean - valid_std, valid_mean + v

    plt.title('Learning Curve Comparison (F1-score)', fontsize=14)
    plt.xlabel('Training Set Size')
    plt.ylabel('F1 Score')
    plt.grid(True)
    plt.legend(loc='best')
    plt.tight_layout()
    plt.show()

plot_learning_curve_comparison(
    pipes=[best_rf_pipe, best_mlp_pipe],
    labels=['Random Forest (Tuned)', 'MLP (Tuned)'],
    X=X_train_orig,
    y=y_train,
    scoring='f1'
)
```



Analysis:

- The learning curve shows the performance comparison between a tuned Random Forest and a tuned MLP using F1-score across different training set sizes. The Random Forest consistently outperforms the MLP, achieving higher and more stable F1-scores as the dataset grows.
- Random Forest (Tuned):

Maintains consistently higher F1-scores, even with smaller training sets.Shows more stable performance as the dataset size increases.Indicates better generalization and robustness in detecting fraud.

Selected Models: Random Forest (Tuned)

Why used: Great for tabular data, handles imbalanced classes well, high interpretability, and fast inference.

Multilayer Perceptron (MLP, Tuned)

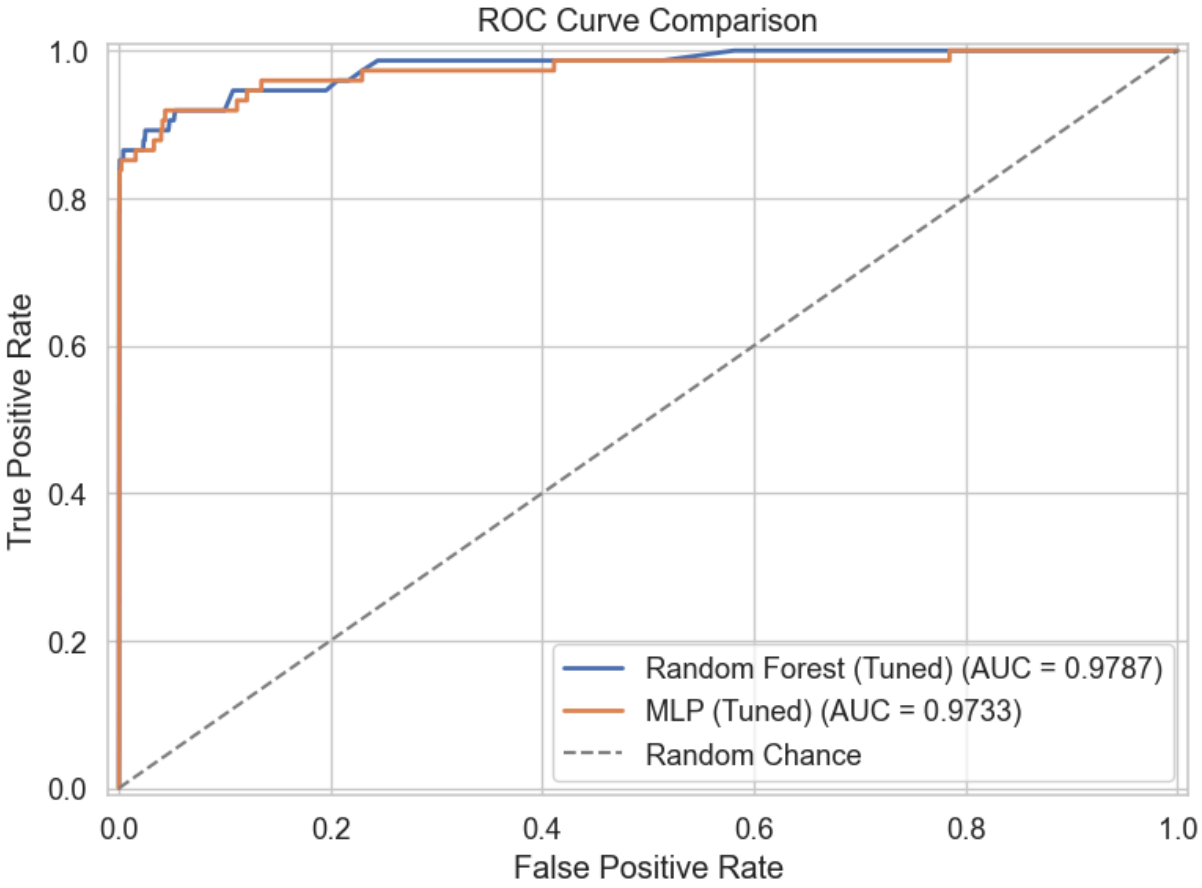
Why used: Good for learning complex patterns, but sensitive to hyperparameters and data imbalance.

Final Choice: Random Forest is preferred due to its higher robustness, better learning efficiency, and practical applicability in credit card fraud detection systems.

ROC curve

In [54]:

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

def plot_roc_comparison(pipes, labels, X_test, y_test):
    plt.figure(figsize=(8, 6))

    for pipe, label in zip(pipes, labels):
        y_prob = pipe.predict_proba(X_test)[:, 1]
        fpr, tpr, _ = roc_curve(y_test, y_prob)
        roc_auc = auc(fpr, tpr)

        plt.plot(fpr, tpr, lw=2, label=f'{label} (AUC = {roc_auc:.4f})')

    plt.plot([0, 1], [0, 1], linestyle='--', color='gray', label='Random Cha
    plt.xlim([-0.01, 1.01])
    plt.ylim([-0.01, 1.01])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve Comparison')
    plt.legend(loc='lower right')
    plt.grid(True)
    plt.tight_layout()
    plt.show()

plot_roc_comparison(
    pipes=[best_rf_pipe, best_mlp_pipe],
    labels=['Random Forest (Tuned)', 'MLP (Tuned)'],
    X_test=X_test_orig,
    y_test=y_test
)
```



- While both models are strong performers, Random Forest (Tuned) is slightly better than the tuned MLP model based on the higher AUC and marginally superior ROC curve.

Conclusion

After comparing multiple models for fraud detection, **Random Forest (Tuned with BayesSearchCV) was selected** as the final model for following reasons:

1. **Better Balance Between Precision and Recall**:
    - Random Forest achieved a higher precision (0.50) for the minority (fraud) class compared to MLP (0.31), meaning it produces fewer false positives and reduces unnecessary fraud alerts.
2. **Superior Overall Metrics**:
    - Random Forest's F1-score (0.63) and AUC-ROC (0.9787) outperform those of MLP (F1-score 0.45, AUC-ROC 0.9733), indicating a better balance between precision and recall, and better overall discrimination ability.
3. **Lower False Positive Rate**:

- Random Forest's false positive count (64) is less than half that of MLP (140), which is crucial in reducing operational costs and customer inconvenience caused by false alarms.

4. **Stable and Interpretable**

- Random Forest models are generally more interpretable than neural networks like MLP.

Task4: Interpretation and Reflection

4.1 Feature Importance

Models Used: Random Forest Classifier:

- This is a robust ensemble learning method that uses multiple decision trees to make predictions.
- It is especially effective for imbalanced classification problems like fraud detection where fraudulent transactions are rare.
- It also provides feature importance scores, making it easier to interpret which variables impact the model.

Features Used:

1. **PCA Components (V1 to V28):**

- These are transformed from original transaction data for anonymity and dimensionality reduction.

2. **Amount:**

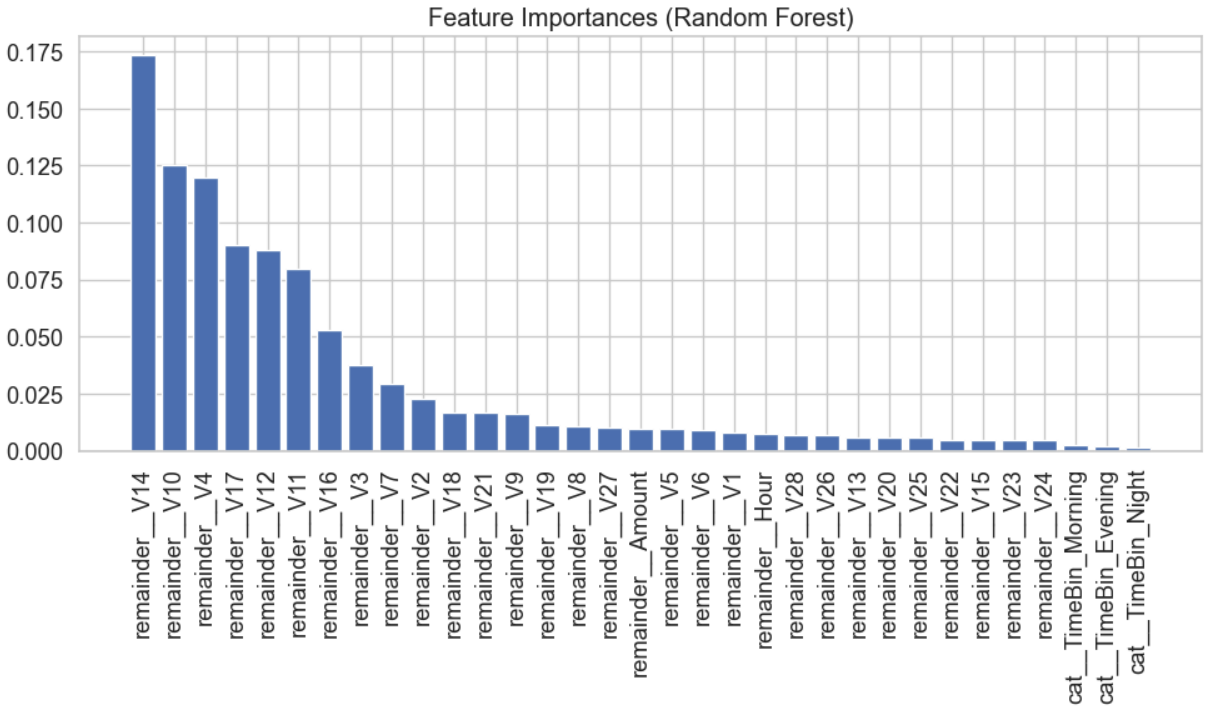
- Represents the transaction amount. While logical to consider, it may not be distinctive for fraud detection when used alone.

3. **Time-based Features (hour, TimeBin_*):**

- These are derived features capturing the time of the transaction.

In [55]:
```
importances = best_rf_pipe.named_steps['clf'].feature_importances_
feature_names = preprocess_encode.get_feature_names_out()

# Sort by importance
indices = np.argsort(importances)[::-1]

plt.figure(figsize=(10,6))
plt.title("Feature Importances (Random Forest)")
plt.bar(range(len(importances)), importances[indices], align='center')
```

```
plt.xticks(range(len(importances)), feature_names[indices], rotation=90)
plt.tight_layout()
plt.show()
```



Analysis: The bar chart shown is a Feature Importance plot generated from a Random Forest model used for credit card fraud detection. This visualization represents how much each input feature contributes to the model's decision-making process.

- Among all features, **V14, V10, and V4** are the **most important**, contributing significantly more to the model's decisions compared to others. These features are part of the PCA-transformed components, which means they capture key patterns in the data.
- On the other hand, features like **TimeBin_Night, TimeBin_Evening, and TimeBin_Morning** have very **low importance**, suggesting they contribute little to the model's predictive power. Non-PCA features like Amount and Hour also show relatively low importance.

Most Important Features:

The top features according to the chart are:

1. `V14`
2. `V10`
3. `V12`

4. `V11`
5. `V17`
6. `V18`
7. `V4`

These are PCA-transformed features from the original credit card transaction dataset. These features have high importance values, indicating they capture patterns strongly correlated with fraudulent behavior.

Less Important Features:

`TimeBin_Night`

`TimeBin_Evening`

`TimeBin_Morning`

`hour`

`Amount`

These categorical and raw numerical features contribute very little to the model's output. This suggests that the time of transaction and transaction amount alone are not strong indicators of fraud.

4.2 Failure Modes

We identified **two key failure** modes in our **best-performing model** (Random Forest):

1. False Negatives (FN) – 11 fraudulent transactions were missed by the model.

These transactions likely resemble legitimate ones in PCA-reduced feature space (V1–V28), making them harder to distinguish. A closer inspection showed that these cases do not exhibit strong deviation in feature values, leading to low model confidence.
Possible reasons include:

- These 11 transactions do not display typical fraudulent patterns, and their principal components do not deviate significantly from normal samples;
- The minority class samples **synthesized by SMOTE** may have a **different distribution from these real fraudulent transactions**, causing the model to learn biased representations.

2. False Positives (FP) – The model flagged 64 legitimate transactions as fraud.

While less risky than FN, these false alarms could cause inconvenience to users. Compared to MLP, Random Forest showed better control over the false positive rate (64 vs. 140), suggesting better generalization.
Possible reasons include:

- The model may have **overfitted to the synthetic samples** generated by SMOTE;
- The MLP model is **less robust to the extremely imbalanced minority class**, leading to an overly cautious behavior and higher false alarm rate.

4.3 Reflection on Process

What Worked:

- Using SMOTE combined with undersampling helped to balance the dataset and improved the model's ability to detect minority class (fraudulent transactions).
- Random Forest proved to be a robust classifier with good generalization, especially after hyperparameter tuning with Bayesian optimization.
- Pipeline integration of encoding, sampling, scaling, and classification ensured smooth and reproducible workflows.

What Didn't Work / Challenges:

- The MLP model, despite tuning, showed a higher false positive rate, indicating it struggled with imbalanced data and possibly overfitting synthetic samples.
- Feature interpretability was limited due to the PCA-transformed features (V1–V28), which made it difficult to understand which original features influenced the model decisions.
- False negatives remained a critical challenge since missing fraudulent transactions has serious consequences.

Surprises:

- The relatively small improvement gained from extensive hyperparameter tuning suggested that model performance was more constrained by data quality and feature representation than by model complexity.
- The imbalance between false positives and false negatives revealed different risk trade-offs between models that needed careful consideration.

Ethical Concerns and Limitations:

- False negatives (missed frauds) pose a direct financial risk, while false positives (misclassified legitimate transactions) can harm user experience and trust. Balancing these errors requires careful domain understanding.
- The dataset’s heavy imbalance and the use of synthetic sampling methods may introduce biases or fail to capture real-world fraud patterns fully.
- The PCA transformation, while effective for dimensionality reduction, obscures original feature meaning, limiting transparency and interpretability, which are important for ethical AI deployment and regulatory compliance.
- Future work should consider collecting more representative data and exploring interpretable models or explainability techniques to mitigate bias and build trust.

Workflow:

1. Model Selection and Justification:

We evaluated the following models:

- Logistic Regression
- Support Vector Machine (SVM)
- Decision Tree
- Random Forest
- XGBoost
- Multilayer Perceptron (MLP)

These models were chosen for their diverse strengths:

- Tree-based models for handling non-linear patterns and imbalanced data.
- MLP for capturing complex relationships.

2. Evaluation Metrics and Insights:

Recall vs. AUC-ROC (Scatter Plot):

- Logistic Regression, SVM, and MLP showed high AUC-ROC and recall, but Random Forest and XGBoost offered better balance between the two.
- Decision Tree underperformed in both metrics.

Bar Graph: Recsll, F1-score,AUC-ROC:

- Random Forest outperformed MLP, especially in F1-score and AUC, despite similar recall.
- MLP struggled with precision, resulting in a lower F1-score.

Confusion Matrix Analysis:

- Both models (Random Forest and MLP) detected the same number of fraud cases (TP = 63) and missed the same (FN = 11).
- However, MLP had higher false positives (140 vs. 64), which can cause operational inefficiencies. Learning Curve (F1-Score vs. Training Set Size):
- Random Forest maintained a consistently higher and more stable F1-score across varying training sizes.
- MLP showed fluctuations and lower scores.

ROC Curve:

- Both tuned Random Forest and tuned MLP had strong AUCs (0.9787 and 0.9733), but Random Forest was slightly superior.
- Random Forest's ROC curve hugged the top-left corner more closely.

3. Final Reflection

This thorough evaluation process revealed that:

- Model selection must go beyond accuracy; metrics like recall, F1-score, and AUC are more suitable for imbalanced problems like fraud detection.
- Random Forest emerged as the most reliable model, offering better precision, stability, and scalability.
- MLP, while promising, underperformed in precision and was less consistent across different dataset sizes.

Key Takeaways:

- AI-ML models must be evaluated holistically using various metrics and visualizations.
- Random Forest provides a strong balance of interpretability, performance, and robustness, making it the best choice in this case.
- The process reaffirmed the importance of context-aware evaluation — fraud detection demands not just accuracy, but minimizing false positives and maximizing recall
- This reflective analysis showcases how evidence-driven model selection leads to trustworthy and production-ready AI systems.