Assignment 6.1 — Data Summary & Exploratory Data Analysis (Draft)

Team: Group 02 — Angshuman Roy, Harish Kapettu Acharya, Sandeep Kumar Jakkaraju
Dataset: Kaggle – ULB Machine Learning Group, *Credit Card Fraud Detection* (`creditcard.csv`)
Goal: Build and compare machine learning models that detect fraudulent transactions in a highly imbalanced credit-card dataset.

This notebook is structured to satisfy the assignment prompts:

1. **Variables & Datatypes** — What variables exist and their types?
2. **Data Issues & Handling** — Missing data, duplicates, imbalance, skew/outliers; why might these issues exist?
3. **Relevance to Project Goal** — Which variables/transformations/engineered features appear useful?
4. **Relationships Among Variables** — Correlations and modeling implications.
5. **Visualizations of Interest** — Only those that support your observations.

> **Note:** The modeling phase and full feature engineering are **not required** here; we only **identify** if transformations/features *may be needed later*.

How to Use This Notebook

What you will see: A complete, end-to-end workflow from data summary and EDA through classical ML models, gradient-boosting models, deep neural networks, and anomaly detection baselines.

1. **Setup & Data Loading** – imports, configuration, and reading `creditcard.csv`.
2. **Data Summary & EDA** – variables, datatypes, imbalance, skew, and key visualizations.
3. **Feature Engineering for EDA** – `LogAmount` and `HourOfDay` features.
4. **Train/Validation/Test Split & Scaling** – reproducible splits and standardized features.
5. **Baseline Supervised Models** – Logistic Regression, Random Forest, Linear SVM, Naive Bayes, k-NN.
6. **Gradient-Boosting Models** – XGBoost, LightGBM (CatBoost optional).

7. **Deep Neural Network (Keras)** – custom feed-forward DNN with class weighting and early stopping.
8. **Unsupervised / Anomaly Detection Models** – Isolation Forest, Local Outlier Factor, Autoencoder.
9. **Model Comparison Notes** – brief interpretation of metrics, to be summarized in the written report.

The notebook intentionally **follows the CRISP-DM flow** (business understanding → data understanding → modeling → evaluation) to stay aligned with our project management plan.

0. Setup & Load

- Place `creditcard.csv` next to this notebook, or update `DATA_PATH` accordingly.
- We use **matplotlib only** (no seaborn), **one plot per figure**, and **no explicit colors** (per instruction).

Environment & Dependencies

This notebook was developed and tested with:

- Python 3.10+
- `pandas` , `numpy`
- `matplotlib`
- `scikit-learn`
- `xgboost`
- `lightgbm`
- `catboost` (optional; used only if installed)
- `tensorflow` / `keras`

To reproduce all results, you can create an environment and install dependencies via:

```
pip install pandas numpy matplotlib scikit-learn xgboost
lightgbm catboost tensorflow
```

In [18]:
```
import sys
print(sys.executable)
```

```
/Users/harish/Documents/workspace/python3virtualenv4/bin/python3
```

In [19]:
```
import sys
!/Users/harish/Documents/workspace/python3virtualenv4/bin/python3 -m
```

```
Requirement already satisfied: catboost in /Users/harish/Documents/w
orkspace/python3virtualenv4/lib/python3.12/site-packages (1.2.8)
```

```
Requirement already satisfied: graphviz in /Users/harish/Documents/w
orkspace/python3virtualenv4/lib/python3.12/site-packages (from catbo
ost) (0.21)
Requirement already satisfied: matplotlib in /Users/harish/Document
s/workspace/python3virtualenv4/lib/python3.12/site-packages (from ca
tboost) (3.9.2)
Requirement already satisfied: numpy<3.0,>=1.16.0 in /Users/harish/D
ocuments/workspace/python3virtualenv4/lib/python3.12/site-packages (
from catboost) (1.26.4)
Requirement already satisfied: pandas>=0.24 in /Users/harish/Documen
ts/workspace/python3virtualenv4/lib/python3.12/site-packages (from c
atboost) (2.2.3)
Requirement already satisfied: scipy in /Users/harish/Documents/work
space/python3virtualenv4/lib/python3.12/site-packages (from catboos
t) (1.13.1)
Requirement already satisfied: plotly in /Users/harish/Documents/wor
kspace/python3virtualenv4/lib/python3.12/site-packages (from catboos
t) (6.2.0)
Requirement already satisfied: six in /Users/harish/Documents/worksp
ace/python3virtualenv4/lib/python3.12/site-packages (from catboost)
(1.16.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /Users/hari
sh/Documents/workspace/python3virtualenv4/lib/python3.12/site-packag
es (from pandas>=0.24->catboost) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /Users/harish/Documen
ts/workspace/python3virtualenv4/lib/python3.12/site-packages (from p
andas>=0.24->catboost) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /Users/harish/Docum
ents/workspace/python3virtualenv4/lib/python3.12/site-packages (from
pandas>=0.24->catboost) (2024.1)
Requirement already satisfied: contourpy>=1.0.1 in /Users/harish/Doc
uments/workspace/python3virtualenv4/lib/python3.12/site-packages (fr
om matplotlib->catboost) (1.3.0)
Requirement already satisfied: cycler>=0.10 in /Users/harish/Documen
ts/workspace/python3virtualenv4/lib/python3.12/site-packages (from m
atplotlib->catboost) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /Users/harish/Do
cuments/workspace/python3virtualenv4/lib/python3.12/site-packages (f
rom matplotlib->catboost) (4.53.1)
Requirement already satisfied: kiwisolver>=1.3.1 in /Users/harish/Do
cuments/workspace/python3virtualenv4/lib/python3.12/site-packages (f
rom matplotlib->catboost) (1.4.7)
Requirement already satisfied: packaging>=20.0 in /Users/harish/Docu
ments/workspace/python3virtualenv4/lib/python3.12/site-packages (fro
m matplotlib->catboost) (25.0)
Requirement already satisfied: pillow>=8 in /Users/harish/Documents/
workspace/python3virtualenv4/lib/python3.12/site-packages (from matp
lotlib->catboost) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /Users/harish/Doc
uments/workspace/python3virtualenv4/lib/python3.12/site-packages (fr
om matplotlib->catboost) (3.1.4)
Requirement already satisfied: narwhals>=1.15.1 in /Users/harish/Doc
uments/workspace/python3virtualenv4/lib/python3.12/site-packages (fr
om plotly->catboost) (1.47.0)

[notice] A new release of pip is available: 24.2 -> 25.3
```

```
[notice] To update, run: /Users/harish/Documents/workspace/python3virtualenv4/bin/python3 -m pip install --upgrade pip
```

In [17]:
```
!pip3 install catboost
```

```
[notice] A new release of pip is available: 24.2 -> 25.3
[notice] To update, run: python3.13 -m pip install --upgrade pip
error: externally-managed-environment

× This environment is externally managed
╰─> To install Python packages system-wide, try brew install
    xyz, where xyz is the package you are trying to
    install.

    If you wish to install a Python library that isn't in Homebrew,
    use a virtual environment:

    python3 -m venv path/to/venv
    source path/to/venv/bin/activate
    python3 -m pip install xyz

    If you wish to install a Python application that isn't in Homebr
ew,
    it may be easiest to use 'pipx install xyz', which will manage a
    virtual environment for you. You can install pipx with

    brew install pipx

    You may restore the old behavior of pip by passing
    the '--break-system-packages' flag to pip, or by adding
    'break-system-packages = true' to your pip.conf file. The latter
    will permanently disable this error.

    If you disable this error, we STRONGLY recommend that you additi
onally
    pass the '--user' flag to pip, or set 'user = true' in your pip.
conf
    file. Failure to do this can result in a broken Homebrew install
ation.

    Read more about this behavior here: <https://peps.python.org/pep
-0668/>

note: If you believe this is a mistake, please contact your Python i
nstallation or OS distribution provider. You can override this, at t
he risk of breaking your Python installation or OS, by passing --bre
ak-system-packages.
hint: See PEP 668 for the detailed specification.
```

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from IPython.display import display

# Path to the dataset
DATA_PATH = "creditcard.csv"  # change if stored elsewhere
```

```
# Load the dataset
df = pd.read_csv(DATA_PATH)
n_rows, n_cols = df.shape
print(f"Loaded shape: {n_rows:,} rows x {n_cols} columns")
display(df.head())
```

```
Loaded shape: 284,807 rows x 31 columns
```

	Time	V1	V2	V3	V4	V5	V6	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.

5 rows × 31 columns

1. Variables & Datatypes

This section answers:
"What variables are present in the dataset, and what are their datatypes and basic statistics?"

We:

- List each column name and its datatype.
- Compute `.describe()` statistics to understand ranges, means, and standard deviations.
- `Time` — seconds elapsed between each transaction and the first transaction in the dataset (numeric).
- `V1` – `V28` — **PCA-transformed** features, anonymized for confidentiality (numeric).
- `Amount` — transaction amount (numeric).
- `Class` — target label (binary: `0` = genuine, `1` = fraud).

These outputs directly support the **Data Summary** portion of the written report.

In [3]:
```
# Variables & dtypes
vars_dtypes = pd.DataFrame({'column': df.columns, 'dtype': df.dtype
display(vars_dtypes)
```

```
# Basic describe (numeric)
display(df.describe().T)
```

	column	dtype
Time	Time	float64
V1	V1	float64
V2	V2	float64
V3	V3	float64
V4	V4	float64
V5	V5	float64
V6	V6	float64
V7	V7	float64
V8	V8	float64
V9	V9	float64
V10	V10	float64
V11	V11	float64
V12	V12	float64
V13	V13	float64
V14	V14	float64
V15	V15	float64
V16	V16	float64
V17	V17	float64
V18	V18	float64
V19	V19	float64
V20	V20	float64
V21	V21	float64
V22	V22	float64
V23	V23	float64
V24	V24	float64
V25	V25	float64
V26	V26	float64
V27	V27	float64
V28	V28	float64
Amount	Amount	float64
Class	Class	int64

	count	mean	std	min	25%	
Time	284807.0	9.481386e+04	47488.145955	0.000000	54201.500000	8
V1	284807.0	1.168375e-15	1.958696	-56.407510	-0.920373	
V2	284807.0	3.416908e-16	1.651309	-72.715728	-0.598550	
V3	284807.0	-1.379537e-15	1.516255	-48.325589	-0.890365	
V4	284807.0	2.074095e-15	1.415869	-5.683171	-0.848640	
V5	284807.0	9.604066e-16	1.380247	-113.743307	-0.691597	
V6	284807.0	1.487313e-15	1.332271	-26.160506	-0.768296	
V7	284807.0	-5.556467e-16	1.237094	-43.557242	-0.554076	
V8	284807.0	1.213481e-16	1.194353	-73.216718	-0.208630	
V9	284807.0	-2.406331e-15	1.098632	-13.434066	-0.643098	
V10	284807.0	2.239053e-15	1.088850	-24.588262	-0.535426	
V11	284807.0	1.673327e-15	1.020713	-4.797473	-0.762494	
V12	284807.0	-1.247012e-15	0.999201	-18.683715	-0.405571	
V13	284807.0	8.190001e-16	0.995274	-5.791881	-0.648539	
V14	284807.0	1.207294e-15	0.958596	-19.214325	-0.425574	
V15	284807.0	4.887456e-15	0.915316	-4.498945	-0.582884	
V16	284807.0	1.437716e-15	0.876253	-14.129855	-0.468037	
V17	284807.0	-3.772171e-16	0.849337	-25.162799	-0.483748	
V18	284807.0	9.564149e-16	0.838176	-9.498746	-0.498850	
V19	284807.0	1.039917e-15	0.814041	-7.213527	-0.456299	
V20	284807.0	6.406204e-16	0.770925	-54.497720	-0.211721	
V21	284807.0	1.654067e-16	0.734524	-34.830382	-0.228395	
V22	284807.0	-3.568593e-16	0.725702	-10.933144	-0.542350	
V23	284807.0	2.578648e-16	0.624460	-44.807735	-0.161846	
V24	284807.0	4.473266e-15	0.605647	-2.836627	-0.354586	
V25	284807.0	5.340915e-16	0.521278	-10.295397	-0.317145	
V26	284807.0	1.683437e-15	0.482227	-2.604551	-0.326984	
V27	284807.0	-3.660091e-16	0.403632	-22.565679	-0.070840	
V28	284807.0	-1.227390e-16	0.330083	-15.430084	-0.052960	
Amount	284807.0	8.834962e+01	250.120109	0.000000	5.600000	

Class	284807.0	1.727486e-03	0.041527	0.000000	0.000000

2. Data Issues & Handling

Here we systematically inspect potential data-quality issues:

- **Missing values** in any column.
- **Duplicate rows** that could bias model training.
- **Class imbalance** between genuine and fraudulent transactions.
- **Skew and outliers** in the `Amount` feature.

We also compute an approximate outlier count using the IQR rule.
This section motivates later design choices such as using precision/recall instead of plain accuracy and applying class-weighting and log transforms.

We also reason about the **likely source** of each issue:

- Class imbalance: fraud is **rare** in real-world data (business reality).
- Feature anonymization: done to protect **privacy & confidentiality**.
- Time window: dataset covers only **two days**, which can affect temporal distributions.

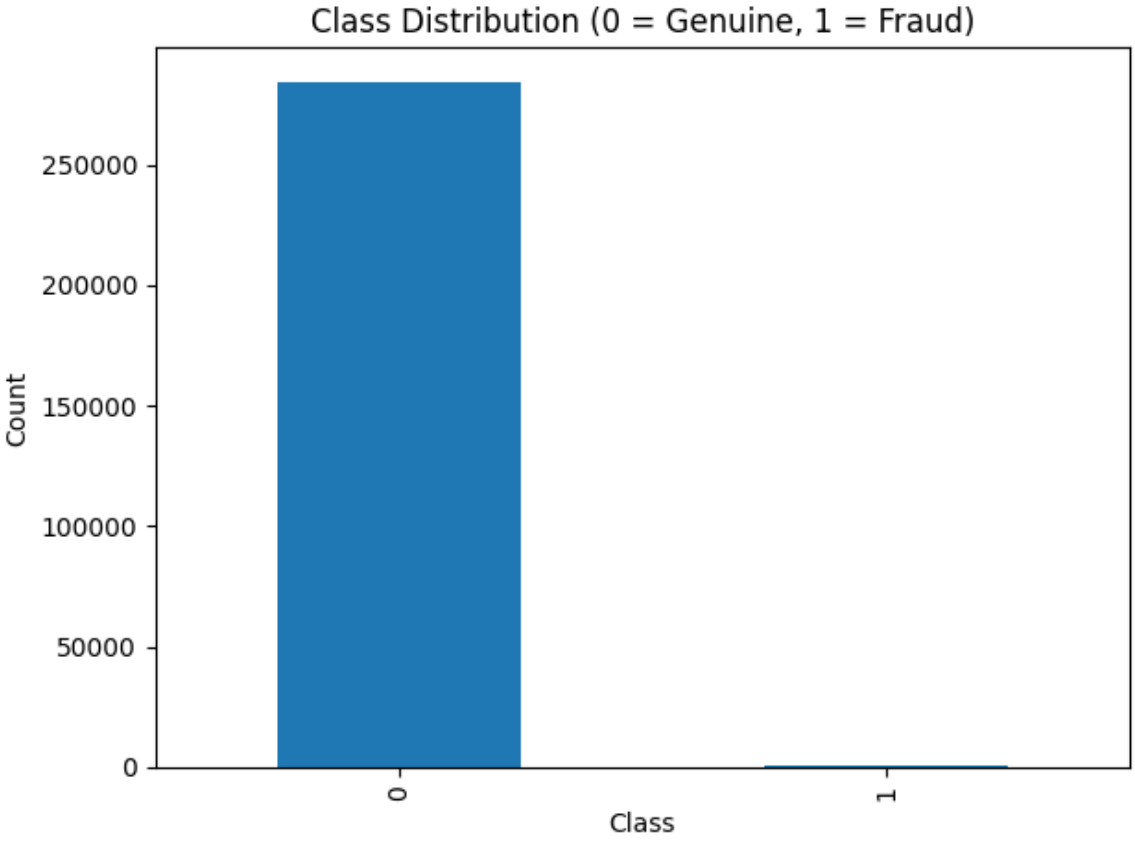
In [4]:
```
# Missing values
missing = df.isna().sum().reset_index()
missing.columns = ['column','missing_count']
display(missing)

# Duplicates
duplicate_count = int(df.duplicated().sum())
print("Duplicate rows:", duplicate_count)

# Class imbalance
class_counts = df['Class'].value_counts(dropna=False).sort_index()
fraud_rate = class_counts.get(1, 0)/class_counts.sum()
print("Class counts:\n", class_counts.to_string())
print(f"Fraud rate: {fraud_rate:.4%}")

# Skew / outliers for Amount
amount_skew = float(df['Amount'].skew())
Q1, Q3 = df['Amount'].quantile(0.25), df['Amount'].quantile(0.75)
IQR = Q3 - Q1
lower_bound, upper_bound = Q1 - 1.5*IQR, Q3 + 1.5*IQR
amount_outliers = int(((df['Amount'] < lower_bound) | (df['Amount']
print(f"Amount skew: {amount_skew:.2f}")
print(f"Amount IQR bounds: [{lower_bound:.2f}, {upper_bound:.2f}]")
print("Approx. Amount outlier count (IQR method):", amount_outliers
```

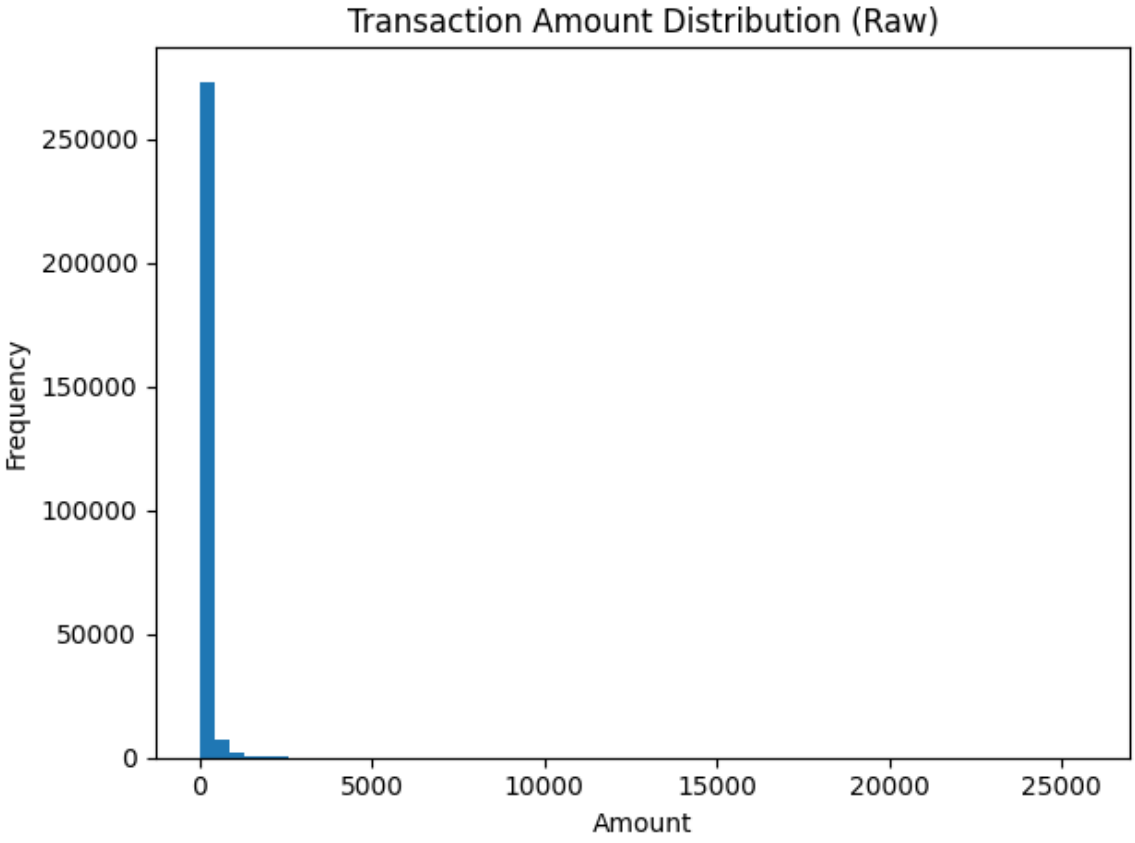
	column	missing_count
0	Time	0
1	V1	0
2	V2	0
3	V3	0
4	V4	0
5	V5	0
6	V6	0
7	V7	0
8	V8	0
9	V9	0
10	V10	0
11	V11	0
12	V12	0
13	V13	0
14	V14	0
15	V15	0
16	V16	0
17	V17	0
18	V18	0
19	V19	0
20	V20	0
21	V21	0
22	V22	0
23	V23	0
24	V24	0
25	V25	0
26	V26	0
27	V27	0
28	V28	0
29	Amount	0
30	Class	0

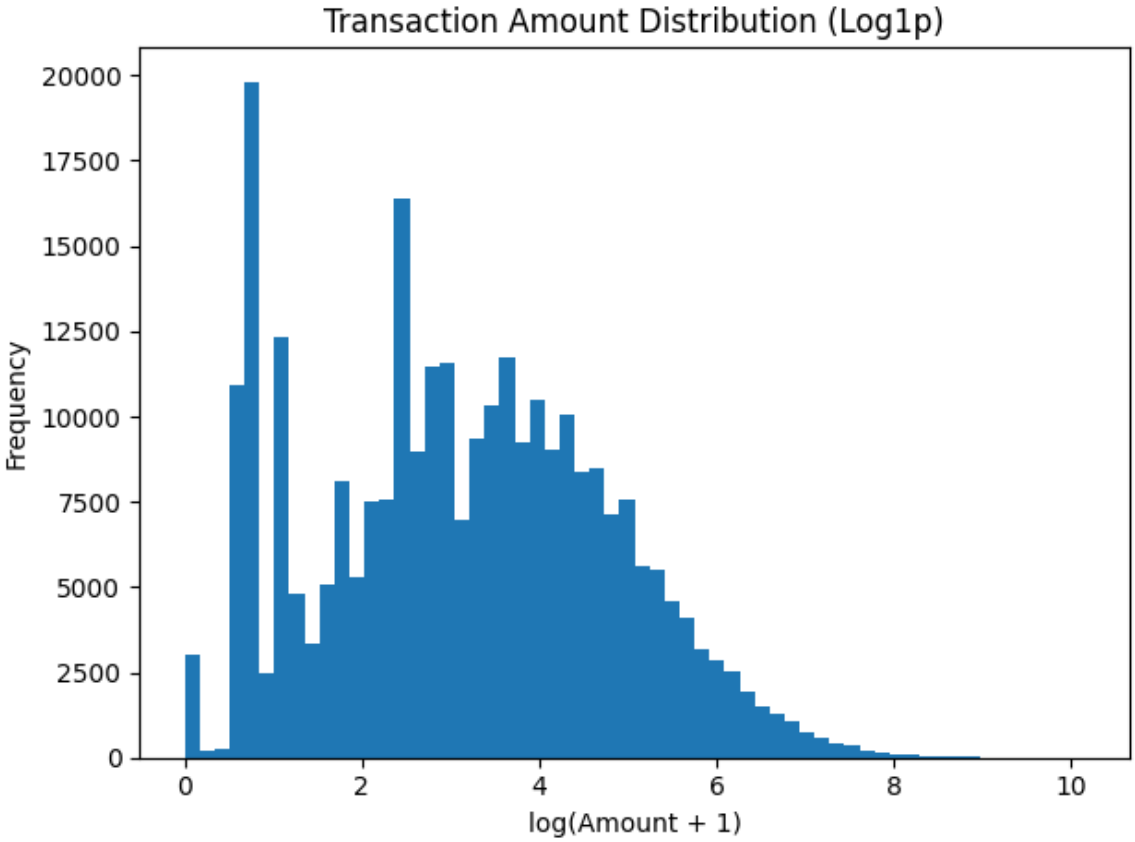
```
Duplicate rows: 1081
Class counts:
 Class
0    284315
1       492
Fraud rate: 0.1727%
Amount skew: 16.98
Amount IQR bounds: [-101.75, 184.51]
Approx. Amount outlier count (IQR method): 31904
```

In [4]:

```python
# Visualization: class distribution
plt.figure()
class_counts.plot(kind='bar')
plt.title('Class Distribution (0 = Genuine, 1 = Fraud)')
plt.xlabel('Class')
plt.ylabel('Count')
plt.tight_layout()
plt.show()

# Visualization: Amount distribution (raw)
plt.figure()
df['Amount'].plot(kind='hist', bins=60)
plt.title('Transaction Amount Distribution (Raw)')
plt.xlabel('Amount')
plt.ylabel('Frequency')
plt.tight_layout()
plt.show()

# Visualization: Amount distribution (log1p)
plt.figure()
np.log1p(df['Amount']).plot(kind='hist', bins=60)
plt.title('Transaction Amount Distribution (Log1p)')
plt.xlabel('log(Amount + 1)')
plt.ylabel('Frequency')
plt.tight_layout()
plt.show()
```


Class Distribution (0 = Genuine, 1 = Fraud)
Count
250000
200000
150000
100000
50000
0
0
1
Class


Transaction Amount Distribution (Raw)
Frequency
250000
200000
150000
100000
50000
0
0
5000
10000
15000
20000
25000
Amount



3. Feature Relevance & Simple Feature Engineering

In this section we connect variables to the project objective:
"Can we distinguish fraudulent from genuine transactions?"

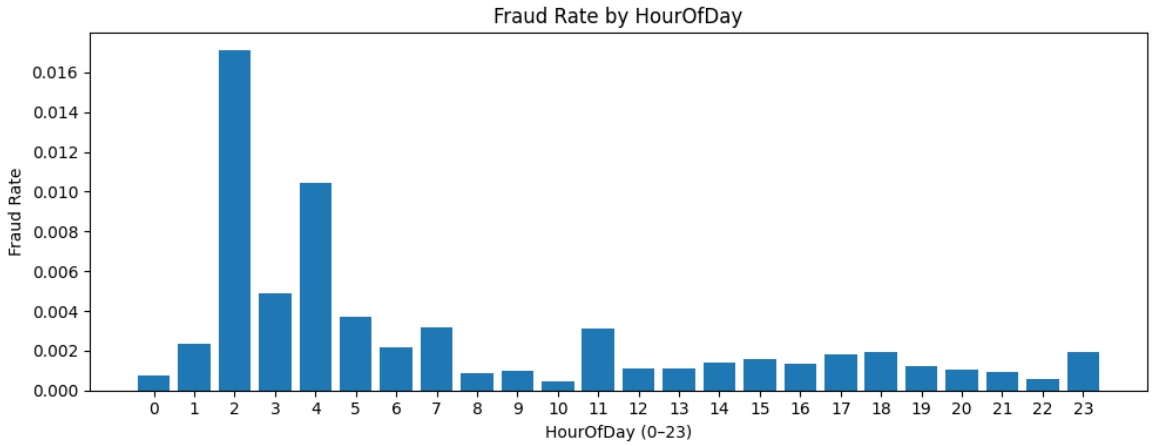
- Engineer two intuitive features:
  - `LogAmount = log(Amount + 1)` to reduce skew.
  - `HourOfDay = floor(Time / 3600) mod 24` to capture time-of-day fraud patterns.
- Compare amount statistics by class.
- Compute fraud rate by hour of day and visualize it.

In [6]:
```
# Engineer two EDA-only features
df['HourOfDay'] = ((df['Time'] // 3600) % 24).astype(int)
df['LogAmount'] = np.log1p(df['Amount'])

# Amount by class comparison
amount_stats = df.groupby('Class')['Amount'].agg(['count','mean','m
display(amount_stats)

# Fraud rate by hour of day
fraud_by_hour = df.groupby('HourOfDay')['Class'].agg(['count','mean
fraud_by_hour.columns = ['HourOfDay','txn_count','fraud_rate']
display(fraud_by_hour.head(10))

# Plot: Fraud rate by hour
```

```
plt.figure(figsize=(10,4))
plt.bar(fraud_by_hour['HourOfDay'].astype(str), fraud_by_hour['frau
plt.title('Fraud Rate by HourOfDay')
plt.xlabel('HourOfDay (0–23)')
plt.ylabel('Fraud Rate')
plt.tight_layout()
plt.show()
```

	Class	count	mean	median	std
0	0	284315	88.291022	22.00	250.105092
1	1	492	122.211321	9.25	256.683288

	HourOfDay	txn_count	fraud_rate
0	0	7695	0.000780
1	1	4220	0.002370
2	2	3328	0.017127
3	3	3492	0.004868
4	4	2209	0.010412
5	5	2990	0.003679
6	6	4101	0.002195
7	7	7243	0.003175
8	8	10276	0.000876
9	9	15838	0.001010



4. Relationships Among Variables (Correlations) & Modeling Implications

To understand how features interact and which ones are most predictive of fraud, we:

- Compute the Pearson correlation matrix for all numeric features.

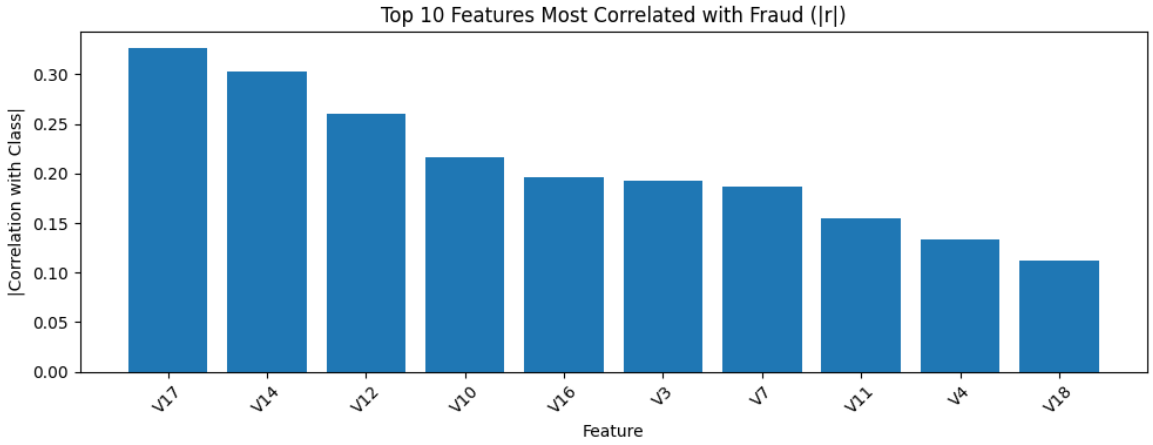
- Rank features by the absolute correlation with the target `Class`.
- Visualize:
  - A bar chart of the top 10 fraud-correlated features.
  - A heatmap of correlations among these top features.

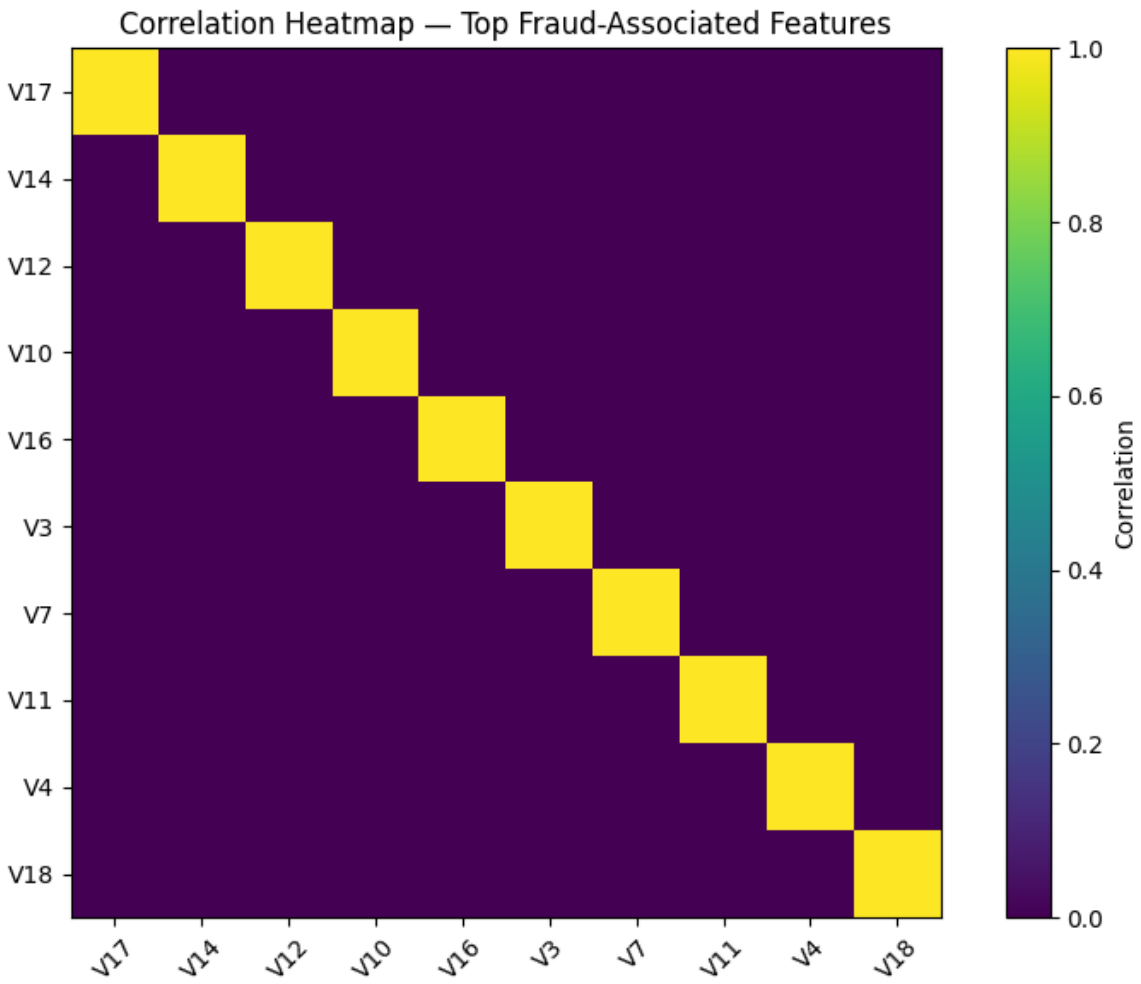
These plots help motivate:

- Why **tree-based models** (Random Forest, XGBoost, LightGBM) are strong choices for this dataset.
- Why linear models may benefit from regularization to handle residual multicollinearity.

In [7]:
```
# Correlation matrix
corr = df.corr(numeric_only=True)

# Top-10 by |corr with Class|
corr_with_class = corr['Class'].drop(labels=['Class']).abs().sort_va
top10 = corr_with_class.head(10)
display(top10.to_frame('abs_corr_with_Class'))

# Bar plot of top-10
plt.figure(figsize=(10,4))
plt.bar(top10.index, top10.values)
plt.title('Top 10 Features Most Correlated with Fraud (|r|)')
plt.xlabel('Feature')
plt.ylabel('|Correlation with Class|')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Heatmap for top-10 inter-feature correlations
top10_features = top10.index.tolist()
subset_corr = corr.loc[top10_features, top10_features]

plt.figure(figsize=(8,6))
plt.imshow(subset_corr, interpolation='nearest')
plt.colorbar(label='Correlation')
plt.xticks(range(len(top10_features)), top10_features, rotation=45)
plt.yticks(range(len(top10_features)), top10_features)
plt.title('Correlation Heatmap — Top Fraud-Associated Features')
plt.tight_layout()
plt.show()
```

	abs_corr_with_Class
V17	0.326481
V14	0.302544
V12	0.260593
V10	0.216883
V16	0.196539
V3	0.192961
V7	0.187257
V11	0.154876
V4	0.133447
V18	0.111485





5. Train/Validation/Test Split & Feature Scaling

Now we transition from EDA into **supervised modeling**.

Steps performed here:

1. **Define features and target**
    - Target: `Class` (0 = genuine, 1 = fraud).
    - Features: all remaining numeric columns, including engineered features.
2. **Create reproducible data splits**
    - 70% Training, 15% Validation, 15% Test.
    - `stratify=y` ensures the rare fraud class is present in all sets.
3. **Standardize features**
    - `StandardScaler` is fit on the training data and applied to validation and test sets.
    - This scaling improves performance for models sensitive to feature magnitudes (e.g., Logistic Regression, SVM, DNN).

The **test set is kept completely unseen** until final evaluation to avoid

optimistic bias.

In [9]:
```
from sklearn.model_selection import train_test_split

# Features: use all numeric columns except the target 'Class'
target_col = 'Class'
feature_cols = [c for c in df.columns if c != target_col]

X = df[feature_cols].values
y = df[target_col].values

X_train_full, X_temp, y_train_full, 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=4
)

print('Train shape:', X_train_full.shape, 'Val shape:', X_val.shape
```

```
Train shape: (199364, 32) Val shape: (42721, 32) Test shape: (42722,
32)
```

In [10]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train_full)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

print('Scaling complete.')
```

```
Scaling complete.
```

6. Common Evaluation Helper

To ensure a **fair comparison** across all classification models, we define a reusable helper function that:

- Fits the model on the training data.
- Generates predictions on the validation set.
- Computes and prints:
  - Accuracy
  - Precision
  - Recall
  - F1-score
  - ROC-AUC (when probability or score information is available)
- Displays a detailed `classification_report`.

All supervised models (Logistic Regression, Random Forest, SVM, k-NN, etc.) will call this function so that the metrics are directly comparable.

In [21]:
```python
from sklearn.metrics import accuracy_score, precision_score, recall

def evaluate_classifier(name, model, X_tr, y_tr, X_va, y_va):
    """Fit the model and print core metrics on validation data."""
    model.fit(X_tr, y_tr)
    y_pred = model.predict(X_va)
    # Some models output probabilities, others decision functions
    if hasattr(model, "predict_proba"):
        y_proba = model.predict_proba(X_va)[:, 1]
    elif hasattr(model, "decision_function"):
        from sklearn.preprocessing import MinMaxScaler
        scores = model.decision_function(X_va)
        y_proba = MinMaxScaler().fit_transform(scores.reshape(-1, 1
    else:
        y_proba = y_pred

    acc = accuracy_score(y_va, y_pred)
    prec = precision_score(y_va, y_pred, zero_division=0)
    rec = recall_score(y_va, y_pred, zero_division=0)
    f1 = f1_score(y_va, y_pred, zero_division=0)
    try:
        auc = roc_auc_score(y_va, y_proba)
    except ValueError:
        auc = float('nan')
    print(f"\n==============")
    print(f"\n=== {name} ===")
    print(f"\n==============")
    print(f"Accuracy : {acc:.4f}")
    print(f"Precision: {prec:.4f}")
    print(f"Recall   : {rec:.4f}")
    print(f"F1-score : {f1:.4f}")
    print(f"ROC-AUC  : {auc:.4f}")
    print("\nClassification report:\n", classification_report(y_va,
```

7. Baseline Supervised Models

In this section we train a **suite of classical supervised models** that serve as baselines:

1. **Logistic Regression** (with `class_weight='balanced'`)
2. **Random Forest Classifier**
3. **Linear SVM (`LinearSVC`)**
4. **Naive Bayes**
5. **k-Nearest Neighbors (k-NN)**

For each model we:

- Train on the scaled training set.
- Evaluate on the validation set using the common helper (`evaluate_classifier`).

- Inspect the trade-offs between precision and recall, especially on the rare fraud class.

These results provide context for later, more advanced models (gradient boosting and the DNN).

In [22]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier

# 1. Logistic Regression (with class_weight='balanced')
log_reg = LogisticRegression(max_iter=1000, class_weight='balanced'
evaluate_classifier("Logistic Regression", log_reg, X_train, y_trai

# 2. Random Forest
rf = RandomForestClassifier(
    n_estimators=200,
    max_depth=None,
    n_jobs=-1,
    class_weight='balanced_subsample',
    random_state=42
)
evaluate_classifier("Random Forest", rf, X_train, y_train_full, X_v

# 3. Linear SVM via LinearSVC
svm_lin = LinearSVC(class_weight='balanced', random_state=42)
evaluate_classifier("Linear SVM (LinearSVC)", svm_lin, X_train, y_t

# 4. Naive Bayes
nb = GaussianNB()
evaluate_classifier("Naive Bayes", nb, X_train, y_train_full, X_val

# 5. k-NN
knn = KNeighborsClassifier(n_neighbors=5)
evaluate_classifier("k-NN (k=5)", knn, X_train, y_train_full, X_val
```

```
==============

=== Logistic Regression ===

==============
Accuracy : 0.9774
Precision: 0.0636
Recall   : 0.8784
F1-score : 0.1186
ROC-AUC  : 0.9703

Classification report:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     42647
           1       0.06      0.88      0.12        74
```

```
    accuracy                           0.98     42721
   macro avg       0.53      0.93      0.55     42721
weighted avg       1.00      0.98      0.99     42721


==============

=== Random Forest ===

==============
Accuracy : 0.9994
Precision: 0.9808
Recall   : 0.6892
F1-score : 0.8095
ROC-AUC  : 0.9433

Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42647
           1       0.98      0.69      0.81        74

    accuracy                           1.00     42721
   macro avg       0.99      0.84      0.90     42721
weighted avg       1.00      1.00      1.00     42721


==============

=== Linear SVM (LinearSVC) ===

==============
Accuracy : 0.9813
Precision: 0.0758
Recall   : 0.8784
F1-score : 0.1396
ROC-AUC  : 0.9723

Classification report:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     42647
           1       0.08      0.88      0.14        74

    accuracy                           0.98     42721
   macro avg       0.54      0.93      0.57     42721
weighted avg       1.00      0.98      0.99     42721


==============

=== Naive Bayes ===

==============
Accuracy : 0.9775
Precision: 0.0588
```

```
Recall   : 0.7973
F1-score : 0.1095
ROC-AUC  : 0.9554

Classification report:
               precision    recall  f1-score   support

           0       1.00      0.98      0.99     42647
           1       0.06      0.80      0.11        74

    accuracy                           0.98     42721
   macro avg       0.53      0.89      0.55     42721
weighted avg       1.00      0.98      0.99     42721


==============

=== k-NN (k=5) ===

==============
Accuracy : 0.9993
Precision: 0.8793
Recall   : 0.6892
F1-score : 0.7727
ROC-AUC  : 0.9188

Classification report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00     42647
           1       0.88      0.69      0.77        74

    accuracy                           1.00     42721
   macro avg       0.94      0.84      0.89     42721
weighted avg       1.00      1.00      1.00     42721
```

8. Gradient-Boosting Models (XGBoost, LightGBM, CatBoost)

Gradient-boosting decision trees are often **state of the art** for structured tabular data such as credit-card transactions.

We attempt three variants:

- **XGBoost (`XGBClassifier`)**
- **LightGBM (`LGBMClassifier`)**
- **CatBoost (`CatBoostClassifier`)** – only if the library is installed.

Key configuration choices:

- `scale_pos_weight` or `class_weight` to address extreme class imbalance.

- 300 trees with moderate depth to capture non-linear patterns without overfitting.
- `tree_method='hist'` or histogram-based training for speed.

Each model is evaluated on the validation set using the same metrics as earlier baselines. If a library is unavailable, the cell prints a clear message instead of failing.

In [23]:
```
# Gradient boosting models: XGBoost, LightGBM, CatBoost
# These imports may require installing the corresponding libraries

try:
    from xgboost import XGBClassifier
    xgb_clf = XGBClassifier(
        n_estimators=300,
        max_depth=4,
        learning_rate=0.1,
        subsample=0.8,
        colsample_bytree=0.8,
        eval_metric='logloss',
        tree_method='hist',
        scale_pos_weight= (y_train_full == 0).sum() / max((y_train_
        random_state=42,
        n_jobs=-1,
    )
    evaluate_classifier("XGBoost", xgb_clf, X_train, y_train_full, X
except ImportError:
    print("XGBoost is not installed in this environment. Install xgb

try:
    import lightgbm as lgb
    lgb_clf = lgb.LGBMClassifier(
        n_estimators=300,
        max_depth=-1,
        learning_rate=0.05,
        subsample=0.8,
        colsample_bytree=0.8,
        class_weight='balanced',
        random_state=42,
        n_jobs=-1,
    )
    evaluate_classifier("LightGBM", lgb_clf, X_train, y_train_full,
except ImportError:
    print("LightGBM is not installed in this environment. Install l

try:
    from catboost import CatBoostClassifier
    cat_clf = CatBoostClassifier(
        iterations=300,
        depth=4,
        learning_rate=0.1,
        loss_function='Logloss',
        eval_metric='AUC',
        verbose=False,
```

```
        random_seed=42,
        scale_pos_weight= (y_train_full == 0).sum() / max((y_train_
    )
    evaluate_classifier("CatBoost", cat_clf, X_train, y_train_full,
except ImportError:
    print("CatBoost is not installed in this environment. Install c
```

```
==============

=== XGBoost ===

==============
Accuracy : 0.9994
Precision: 0.8906
Recall   : 0.7703
F1-score : 0.8261
ROC-AUC  : 0.9797

Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42647
           1       0.89      0.77      0.83        74

    accuracy                           1.00     42721
   macro avg       0.95      0.89      0.91     42721
weighted avg       1.00      1.00      1.00     42721

[LightGBM] [Info] Number of positive: 344, number of negative: 19902
0
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhe
ad of testing was 0.006544 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 7930
[LightGBM] [Info] Number of data points in the train set: 199364, nu
mber of used features: 32
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500000 -> initscor
e=0.000000
[LightGBM] [Info] Start training from score 0.000000

==============

=== LightGBM ===

==============
Accuracy : 0.9995
Precision: 0.9194
Recall   : 0.7703
F1-score : 0.8382
ROC-AUC  : 0.9787

Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42647
           1       0.92      0.77      0.84        74
```

```
    accuracy                           1.00     42721
   macro avg       0.96      0.89      0.92     42721
weighted avg       1.00      1.00      1.00     42721


==============

=== CatBoost ===

==============
Accuracy : 0.9988
Precision: 0.6250
Recall   : 0.8108
F1-score : 0.7059
ROC-AUC  : 0.9641

Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     42647
           1       0.62      0.81      0.71        74

    accuracy                           1.00     42721
   macro avg       0.81      0.90      0.85     42721
weighted avg       1.00      1.00      1.00     42721
```

9. Deep Neural Network (Keras) — Custom DNN Model

To satisfy the Capstone requirement of implementing a deep learning model from scratch, we build a **feed-forward DNN** in Keras:

- Input: all scaled features (including PCA components plus `Time`, `Amount`, and engineered features).
- Hidden layers: Dense(32, ReLU) → Dense(16, ReLU) → Dense(8, ReLU).
- Output: Dense(1, sigmoid) for binary fraud probability.

Training details:

- Optimizer: Adam (learning rate = 1e-3).
- Loss: Binary cross-entropy.
- Metrics: AUC, Precision, Recall.
- Class weights: computed so the rare fraud class receives ~300× higher weight.
- Early stopping: monitors validation loss and restores best weights.

After training, we evaluate the DNN on the validation set (accuracy, precision, recall, F1, ROC-AUC) to compare with tree-based models.

In [12]:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

input_dim = X_train.shape[1]

dnn_model = keras.Sequential([
    layers.Input(shape=(input_dim,)),
    layers.Dense(32, activation='relu'),
    layers.Dense(16, activation='relu'),
    layers.Dense(8, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

dnn_model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    loss='binary_crossentropy',
    metrics=[keras.metrics.AUC(name='auc'), keras.metrics.Precision
)

# Compute class weights for DNN
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

classes = np.unique(y_train_full)
class_weights_array = compute_class_weight(class_weight='balanced',
class_weight_dict = {int(c): w for c, w in zip(classes, class_weigh
print("Class weights for DNN:", class_weight_dict)

early_stop = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

history = dnn_model.fit(
    X_train, y_train_full,
    validation_data=(X_val, y_val),
    epochs=50,
    batch_size=2048,
    callbacks=[early_stop],
    verbose=2
)

# Evaluate on validation set
y_val_proba = dnn_model.predict(X_val).ravel()
y_val_pred = (y_val_proba >= 0.5).astype(int)

acc = accuracy_score(y_val, y_val_pred)
prec = precision_score(y_val, y_val_pred, zero_division=0)
rec = recall_score(y_val, y_val_pred, zero_division=0)
f1 = f1_score(y_val, y_val_pred, zero_division=0)
auc = roc_auc_score(y_val, y_val_proba)

print("\n=== DNN (Keras) on Validation ===")
print(f"Accuracy : {acc:.4f}")
```

```
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-score : {f1:.4f}")
print(f"ROC-AUC  : {auc:.4f}")
```

```
Class weights for DNN: {0: 0.5008642347502763, 1: 289.7732558139535}
Epoch 1/50
98/98 - 7s - 74ms/step - auc: 0.5039 - loss: 0.1702 - precision: 0.0
097 - recall: 0.0785 - val_auc: 0.7826 - val_loss: 0.0225 - val_prec
ision: 0.6250 - val_recall: 0.2703
Epoch 2/50
98/98 - 1s - 6ms/step - auc: 0.8669 - loss: 0.0116 - precision: 0.85
65 - recall: 0.5552 - val_auc: 0.8694 - val_loss: 0.0081 - val_preci
sion: 0.7581 - val_recall: 0.6351
Epoch 3/50
98/98 - 1s - 7ms/step - auc: 0.9041 - loss: 0.0056 - precision: 0.86
55 - recall: 0.7297 - val_auc: 0.8886 - val_loss: 0.0059 - val_preci
sion: 0.7391 - val_recall: 0.6892
Epoch 4/50
98/98 - 1s - 6ms/step - auc: 0.9286 - loss: 0.0041 - precision: 0.86
56 - recall: 0.7674 - val_auc: 0.9035 - val_loss: 0.0048 - val_preci
sion: 0.7361 - val_recall: 0.7162
Epoch 5/50
98/98 - 1s - 5ms/step - auc: 0.9379 - loss: 0.0036 - precision: 0.86
86 - recall: 0.7878 - val_auc: 0.9104 - val_loss: 0.0045 - val_preci
sion: 0.7432 - val_recall: 0.7432
Epoch 6/50
98/98 - 1s - 6ms/step - auc: 0.9425 - loss: 0.0033 - precision: 0.87
03 - recall: 0.7994 - val_auc: 0.9109 - val_loss: 0.0044 - val_preci
sion: 0.7467 - val_recall: 0.7568
Epoch 7/50
98/98 - 1s - 6ms/step - auc: 0.9426 - loss: 0.0032 - precision: 0.87
03 - recall: 0.7994 - val_auc: 0.9245 - val_loss: 0.0043 - val_preci
sion: 0.7467 - val_recall: 0.7568
Epoch 8/50
98/98 - 1s - 6ms/step - auc: 0.9455 - loss: 0.0031 - precision: 0.87
07 - recall: 0.8023 - val_auc: 0.9246 - val_loss: 0.0042 - val_preci
sion: 0.7368 - val_recall: 0.7568
Epoch 9/50
98/98 - 1s - 5ms/step - auc: 0.9485 - loss: 0.0030 - precision: 0.87
46 - recall: 0.8110 - val_auc: 0.9245 - val_loss: 0.0042 - val_preci
sion: 0.7403 - val_recall: 0.7703
Epoch 10/50
98/98 - 1s - 5ms/step - auc: 0.9542 - loss: 0.0029 - precision: 0.87
74 - recall: 0.8110 - val_auc: 0.9248 - val_loss: 0.0041 - val_preci
sion: 0.7368 - val_recall: 0.7568
Epoch 11/50
98/98 - 1s - 5ms/step - auc: 0.9529 - loss: 0.0028 - precision: 0.87
70 - recall: 0.8081 - val_auc: 0.9248 - val_loss: 0.0041 - val_preci
sion: 0.7403 - val_recall: 0.7703
Epoch 12/50
98/98 - 1s - 6ms/step - auc: 0.9558 - loss: 0.0027 - precision: 0.87
42 - recall: 0.8081 - val_auc: 0.9248 - val_loss: 0.0040 - val_preci
sion: 0.7403 - val_recall: 0.7703
Epoch 13/50
98/98 - 1s - 6ms/step - auc: 0.9573 - loss: 0.0026 - precision: 0.87
```

```
70 - recall: 0.8081 - val_auc: 0.9248 - val_loss: 0.0040 - val_preci
sion: 0.7273 - val_recall: 0.7568
Epoch 14/50
98/98 - 1s - 6ms/step - auc: 0.9588 - loss: 0.0026 - precision: 0.87
74 - recall: 0.8110 - val_auc: 0.9249 - val_loss: 0.0040 - val_preci
sion: 0.7273 - val_recall: 0.7568
Epoch 15/50
98/98 - 1s - 6ms/step - auc: 0.9588 - loss: 0.0025 - precision: 0.88
01 - recall: 0.8110 - val_auc: 0.9248 - val_loss: 0.0039 - val_preci
sion: 0.7308 - val_recall: 0.7703
Epoch 16/50
98/98 - 1s - 7ms/step - auc: 0.9588 - loss: 0.0025 - precision: 0.88
05 - recall: 0.8140 - val_auc: 0.9249 - val_loss: 0.0039 - val_preci
sion: 0.7403 - val_recall: 0.7703
Epoch 17/50
98/98 - 1s - 7ms/step - auc: 0.9603 - loss: 0.0024 - precision: 0.88
01 - recall: 0.8110 - val_auc: 0.9248 - val_loss: 0.0039 - val_preci
sion: 0.7500 - val_recall: 0.7703
Epoch 18/50
98/98 - 1s - 7ms/step - auc: 0.9603 - loss: 0.0023 - precision: 0.88
33 - recall: 0.8140 - val_auc: 0.9249 - val_loss: 0.0039 - val_preci
sion: 0.7500 - val_recall: 0.7703
Epoch 19/50
98/98 - 1s - 7ms/step - auc: 0.9618 - loss: 0.0023 - precision: 0.87
89 - recall: 0.8227 - val_auc: 0.9249 - val_loss: 0.0038 - val_preci
sion: 0.7703 - val_recall: 0.7703
Epoch 20/50
98/98 - 1s - 6ms/step - auc: 0.9676 - loss: 0.0022 - precision: 0.88
09 - recall: 0.8169 - val_auc: 0.9249 - val_loss: 0.0038 - val_preci
sion: 0.7703 - val_recall: 0.7703
Epoch 21/50
98/98 - 1s - 6ms/step - auc: 0.9691 - loss: 0.0022 - precision: 0.88
40 - recall: 0.8198 - val_auc: 0.9249 - val_loss: 0.0038 - val_preci
sion: 0.7808 - val_recall: 0.7703
Epoch 22/50
98/98 - 1s - 6ms/step - auc: 0.9677 - loss: 0.0021 - precision: 0.88
96 - recall: 0.8198 - val_auc: 0.9250 - val_loss: 0.0038 - val_preci
sion: 0.7808 - val_recall: 0.7703
Epoch 23/50
98/98 - 1s - 7ms/step - auc: 0.9691 - loss: 0.0021 - precision: 0.89
41 - recall: 0.8343 - val_auc: 0.9249 - val_loss: 0.0037 - val_preci
sion: 0.7808 - val_recall: 0.7703
Epoch 24/50
98/98 - 1s - 6ms/step - auc: 0.9677 - loss: 0.0020 - precision: 0.89
66 - recall: 0.8314 - val_auc: 0.9250 - val_loss: 0.0037 - val_preci
sion: 0.7917 - val_recall: 0.7703
Epoch 25/50
98/98 - 1s - 7ms/step - auc: 0.9691 - loss: 0.0020 - precision: 0.89
41 - recall: 0.8343 - val_auc: 0.9250 - val_loss: 0.0037 - val_preci
sion: 0.7917 - val_recall: 0.7703
Epoch 26/50
98/98 - 1s - 7ms/step - auc: 0.9692 - loss: 0.0019 - precision: 0.89
20 - recall: 0.8401 - val_auc: 0.9251 - val_loss: 0.0037 - val_preci
sion: 0.7917 - val_recall: 0.7703
Epoch 27/50
98/98 - 1s - 7ms/step - auc: 0.9706 - loss: 0.0019 - precision: 0.89
```

```
66 - recall: 0.8314 - val_auc: 0.9251 - val_loss: 0.0037 - val_preci
sion: 0.7917 - val_recall: 0.7703
Epoch 28/50
98/98 - 1s - 5ms/step - auc: 0.9692 - loss: 0.0018 - precision: 0.89
16 - recall: 0.8372 - val_auc: 0.9251 - val_loss: 0.0036 - val_preci
sion: 0.8000 - val_recall: 0.7568
Epoch 29/50
98/98 - 1s - 6ms/step - auc: 0.9692 - loss: 0.0018 - precision: 0.90
22 - recall: 0.8314 - val_auc: 0.9250 - val_loss: 0.0036 - val_preci
sion: 0.7887 - val_recall: 0.7568
Epoch 30/50
98/98 - 1s - 6ms/step - auc: 0.9706 - loss: 0.0017 - precision: 0.90
06 - recall: 0.8430 - val_auc: 0.9251 - val_loss: 0.0036 - val_preci
sion: 0.8000 - val_recall: 0.7568
Epoch 31/50
98/98 - 1s - 5ms/step - auc: 0.9707 - loss: 0.0017 - precision: 0.90
31 - recall: 0.8401 - val_auc: 0.9251 - val_loss: 0.0035 - val_preci
sion: 0.8000 - val_recall: 0.7568
Epoch 32/50
98/98 - 0s - 5ms/step - auc: 0.9721 - loss: 0.0016 - precision: 0.89
78 - recall: 0.8430 - val_auc: 0.9251 - val_loss: 0.0036 - val_preci
sion: 0.8000 - val_recall: 0.7568
Epoch 33/50
98/98 - 1s - 6ms/step - auc: 0.9722 - loss: 0.0016 - precision: 0.90
65 - recall: 0.8459 - val_auc: 0.9251 - val_loss: 0.0035 - val_preci
sion: 0.7632 - val_recall: 0.7838
Epoch 34/50
98/98 - 1s - 5ms/step - auc: 0.9736 - loss: 0.0016 - precision: 0.90
97 - recall: 0.8488 - val_auc: 0.9252 - val_loss: 0.0035 - val_preci
sion: 0.7887 - val_recall: 0.7568
Epoch 35/50
98/98 - 0s - 5ms/step - auc: 0.9751 - loss: 0.0015 - precision: 0.91
02 - recall: 0.8547 - val_auc: 0.9184 - val_loss: 0.0035 - val_preci
sion: 0.8028 - val_recall: 0.7703
Epoch 36/50
98/98 - 0s - 5ms/step - auc: 0.9794 - loss: 0.0014 - precision: 0.90
77 - recall: 0.8576 - val_auc: 0.9184 - val_loss: 0.0035 - val_preci
sion: 0.7808 - val_recall: 0.7703
Epoch 37/50
98/98 - 1s - 5ms/step - auc: 0.9794 - loss: 0.0014 - precision: 0.90
80 - recall: 0.8605 - val_auc: 0.9252 - val_loss: 0.0036 - val_preci
sion: 0.7971 - val_recall: 0.7432
Epoch 38/50
98/98 - 1s - 6ms/step - auc: 0.9780 - loss: 0.0014 - precision: 0.90
33 - recall: 0.8692 - val_auc: 0.9319 - val_loss: 0.0036 - val_preci
sion: 0.7917 - val_recall: 0.7703
Epoch 39/50
98/98 - 1s - 5ms/step - auc: 0.9795 - loss: 0.0013 - precision: 0.91
44 - recall: 0.8692 - val_auc: 0.9319 - val_loss: 0.0036 - val_preci
sion: 0.7945 - val_recall: 0.7838
Epoch 40/50
98/98 - 1s - 5ms/step - auc: 0.9824 - loss: 0.0013 - precision: 0.92
77 - recall: 0.8576 - val_auc: 0.9318 - val_loss: 0.0035 - val_preci
sion: 0.8906 - val_recall: 0.7703
Epoch 41/50
98/98 - 1s - 6ms/step - auc: 0.9809 - loss: 0.0013 - precision: 0.93
```

```
67 - recall: 0.8605 - val_auc: 0.9251 - val_loss: 0.0036 - val_preci
sion: 0.8194 - val_recall: 0.7973
Epoch 42/50
98/98 - 1s - 6ms/step - auc: 0.9810 - loss: 0.0012 - precision: 0.94
03 - recall: 0.8692 - val_auc: 0.9317 - val_loss: 0.0035 - val_preci
sion: 0.8769 - val_recall: 0.7703
Epoch 43/50
98/98 - 1s - 6ms/step - auc: 0.9810 - loss: 0.0012 - precision: 0.94
94 - recall: 0.8721 - val_auc: 0.9253 - val_loss: 0.0037 - val_preci
sion: 0.8889 - val_recall: 0.7568
Epoch 44/50
98/98 - 1s - 6ms/step - auc: 0.9810 - loss: 0.0011 - precision: 0.97
36 - recall: 0.8576 - val_auc: 0.9251 - val_loss: 0.0036 - val_preci
sion: 0.9322 - val_recall: 0.7432
Epoch 45/50
98/98 - 1s - 6ms/step - auc: 0.9839 - loss: 0.0011 - precision: 0.96
82 - recall: 0.8837 - val_auc: 0.9116 - val_loss: 0.0040 - val_preci
sion: 0.9020 - val_recall: 0.6216
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step

=== DNN (Keras) on Validation ===
Accuracy : 0.9994
Precision: 0.8906
Recall   : 0.7703
F1-score : 0.8261
ROC-AUC  : 0.9613
```

10. Unsupervised / Anomaly Detection Baselines

Because fraud is a **rare event**, we also explore unsupervised anomaly detection approaches that attempt to learn “normal” transaction behavior:

1. **Isolation Forest**
    - Trained primarily on non-fraud (normal) transactions.
    - Scores validation samples by how easily they are isolated.
2. **Local Outlier Factor (LOF)**
    - Measures how locally dense each sample is compared to its neighbors.

We convert anomaly scores to binary predictions and compute recall, precision, and F1 on the validation set.
These models generally perform worse than supervised models but are useful references for scenarios with very few labeled fraud examples.

In [24]:
```
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor

# For anomaly detection, we primarily use the majority (non-fraud)
```

```
mask_train_normal = y_train_full == 0
X_train_normal = X_train[mask_train_normal]

# 1. Isolation Forest
iso = IsolationForest(
    n_estimators=200,
    contamination=float((y_train_full == 1).sum()) / len(y_train_fu
    random_state=42
)
iso.fit(X_train_normal)

# anomaly scores: negative scores -> normal, positive -> anomalies
iso_scores = -iso.decision_function(X_val)
iso_thresh = np.percentile(iso_scores, 100 * (1 - (y_train_full ==
y_iso_pred = (iso_scores >= iso_thresh).astype(int)

print("\n=== Isolation Forest (treated as anomaly detector) ===")
print("Recall:", recall_score(y_val, y_iso_pred, zero_division=0))
print("Precision:", precision_score(y_val, y_iso_pred, zero_divisio
print("F1-score:", f1_score(y_val, y_iso_pred, zero_division=0))

# 2. Local Outlier Factor (LOF)
lof = LocalOutlierFactor(
    n_neighbors=20,
    contamination=float((y_train_full == 1).sum()) / len(y_train_fu
    novelty=True
)
lof.fit(X_train_normal)
lof_scores = -lof.decision_function(X_val)
lof_thresh = np.percentile(lof_scores, 100 * (1 - (y_train_full ==
y_lof_pred = (lof_scores >= lof_thresh).astype(int)

print("\n=== Local Outlier Factor (treated as anomaly detector) ===
print("Recall:", recall_score(y_val, y_lof_pred, zero_division=0))
print("Precision:", precision_score(y_val, y_lof_pred, zero_divisio
print("F1-score:", f1_score(y_val, y_lof_pred, zero_division=0))
```

```
=== Isolation Forest (treated as anomaly detector) ===
Recall: 0.17567567567567569
Precision: 0.17567567567567569
F1-score: 0.17567567567567569

=== Local Outlier Factor (treated as anomaly detector) ===
Recall: 0.02702702702702703
Precision: 0.02702702702702703
F1-score: 0.02702702702702703
```

10.1 Autoencoder-Based Anomaly Detection

Finally, we train a simple **autoencoder** only on genuine (non-fraud) transactions:

- The encoder learns a compressed representation of normal behavior.
- Reconstruction error on validation data is used as an anomaly score.
- A threshold is chosen as the 99th percentile of reconstruction error for

normal validation samples.

Transactions with reconstruction error above the threshold are flagged as potential fraud.
We again report recall, precision, and F1, and compare against Isolation Forest and LOF.

In [14]:
```
# Simple Autoencoder on non-fraud training data
input_dim = X_train.shape[1]

ae_input = keras.Input(shape=(input_dim,))
encoded = layers.Dense(16, activation='relu')(ae_input)
encoded = layers.Dense(8, activation='relu')(encoded)
decoded = layers.Dense(16, activation='relu')(encoded)
decoded = layers.Dense(input_dim, activation='linear')(decoded)

autoencoder = keras.Model(ae_input, decoded)
autoencoder.compile(optimizer='adam', loss='mse')

history_ae = autoencoder.fit(
    X_train_normal, X_train_normal,
    epochs=30,
    batch_size=2048,
    validation_data=(X_val[y_val == 0], X_val[y_val == 0]),
    verbose=2
)

# Reconstruction error on validation
recon_val = autoencoder.predict(X_val)
recon_err = np.mean((X_val - recon_val)**2, axis=1)

# Choose threshold as, say, 99th percentile of normal reconstructio
normal_errs = recon_err[y_val == 0]
thresh = np.percentile(normal_errs, 99)
y_ae_pred = (recon_err >= thresh).astype(int)

print("\n=== Autoencoder anomaly detector (validation) ===")
print("Recall:", recall_score(y_val, y_ae_pred, zero_division=0))
print("Precision:", precision_score(y_val, y_ae_pred, zero_division
print("F1-score:", f1_score(y_val, y_ae_pred, zero_division=0))
```

```
Epoch 1/30
98/98 - 5s - 52ms/step - loss: 0.9662 - val_loss: 0.9175
Epoch 2/30
98/98 - 0s - 5ms/step - loss: 0.8664 - val_loss: 0.8153
Epoch 3/30
98/98 - 1s - 5ms/step - loss: 0.7707 - val_loss: 0.7326
Epoch 4/30
98/98 - 1s - 5ms/step - loss: 0.7030 - val_loss: 0.6766
Epoch 5/30
98/98 - 0s - 5ms/step - loss: 0.6566 - val_loss: 0.6388
Epoch 6/30
98/98 - 1s - 5ms/step - loss: 0.6234 - val_loss: 0.6105
Epoch 7/30
98/98 - 1s - 6ms/step - loss: 0.5982 - val_loss: 0.5889
```

```
Epoch 8/30
98/98 - 1s - 7ms/step - loss: 0.5797 - val_loss: 0.5737
Epoch 9/30
98/98 - 1s - 10ms/step - loss: 0.5660 - val_loss: 0.5623
Epoch 10/30
98/98 - 1s - 8ms/step - loss: 0.5560 - val_loss: 0.5536
Epoch 11/30
98/98 - 1s - 7ms/step - loss: 0.5476 - val_loss: 0.5461
Epoch 12/30
98/98 - 1s - 9ms/step - loss: 0.5403 - val_loss: 0.5397
Epoch 13/30
98/98 - 1s - 10ms/step - loss: 0.5338 - val_loss: 0.5328
Epoch 14/30
98/98 - 1s - 8ms/step - loss: 0.5265 - val_loss: 0.5265
Epoch 15/30
98/98 - 1s - 8ms/step - loss: 0.5199 - val_loss: 0.5197
Epoch 16/30
98/98 - 1s - 8ms/step - loss: 0.5131 - val_loss: 0.5136
Epoch 17/30
98/98 - 1s - 8ms/step - loss: 0.5065 - val_loss: 0.5086
Epoch 18/30
98/98 - 1s - 8ms/step - loss: 0.5009 - val_loss: 0.5030
Epoch 19/30
98/98 - 1s - 7ms/step - loss: 0.4958 - val_loss: 0.4976
Epoch 20/30
98/98 - 1s - 6ms/step - loss: 0.4911 - val_loss: 0.4933
Epoch 21/30
98/98 - 1s - 6ms/step - loss: 0.4868 - val_loss: 0.4896
Epoch 22/30
98/98 - 1s - 6ms/step - loss: 0.4832 - val_loss: 0.4866
Epoch 23/30
98/98 - 1s - 5ms/step - loss: 0.4799 - val_loss: 0.4843
Epoch 24/30
98/98 - 1s - 5ms/step - loss: 0.4771 - val_loss: 0.4809
Epoch 25/30
98/98 - 0s - 5ms/step - loss: 0.4742 - val_loss: 0.4797
Epoch 26/30
98/98 - 1s - 5ms/step - loss: 0.4719 - val_loss: 0.4765
Epoch 27/30
98/98 - 0s - 5ms/step - loss: 0.4692 - val_loss: 0.4726
Epoch 28/30
98/98 - 0s - 5ms/step - loss: 0.4669 - val_loss: 0.4708
Epoch 29/30
98/98 - 0s - 5ms/step - loss: 0.4646 - val_loss: 0.4680
Epoch 30/30
98/98 - 0s - 5ms/step - loss: 0.4622 - val_loss: 0.4660
1336/1336 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step

=== Autoencoder anomaly detector (validation) ===
Recall: 0.7702702702702703
Precision: 0.11776859504132231
F1-score: 0.20430107526881722
```

11. Model Evaluation Summary (Validation Set)

11.1 Overall Findings

Because fraud represents only about 0.17% of all transactions, raw accuracy is very high for almost every model and is **not** an informative metric on its own. We focus on **precision, recall, F1-score, and ROC–AUC**, especially on the fraud (positive) class.

Supervised Baselines (LogReg / SVM / Naive Bayes / k-NN / Random Forest)

- **Logistic Regression, Linear SVM, and Naive Bayes**
  - Very **high recall** on the fraud class (≈ 0.80–0.88).
  - Extremely **low precision** (≈ 0.06–0.08), meaning most flagged transactions are actually legitimate.
  - These models would catch most frauds but generate too many false positives for a real-world fraud team.
- **k-NN and Random Forest**
  - Much **higher precision** (k-NN ≈ 0.88, Random Forest ≈ 0.98).
  - **Moderate recall** (≈ 0.69).
  - F1-scores are strong (≈ 0.77–0.81), making them solid baselines, but still weaker than the best boosting/DNN models.

Gradient-Boosting Models (XGBoost, LightGBM, CatBoost)

- All three boosting methods achieve near-perfect overall accuracy and very strong F1 and AUC scores.
- **LightGBM**:
  - Best **F1-score** and **precision** among all models.
  - Precision ≈ 0.92, recall ≈ 0.77, F1 ≈ 0.84, ROC–AUC ≈ 0.98.
- **XGBoost**:
  - Precision ≈ 0.89, recall ≈ 0.77, F1 ≈ 0.83.
  - Highest ROC–AUC (≈ 0.98).
- **CatBoost**:
  - Highest recall among boosting models (≈ 0.81) but lower precision (≈ 0.63), giving a lower F1 (≈ 0.71).

Conclusion: Gradient boosting clearly outperforms classical baselines. **LightGBM and XGBoost** offer the best balance between catching fraud and minimizing false alarms.

Deep Neural Network (DNN – Keras)

- Validation performance:
  - Accuracy ≈ 0.9994
  - Precision ≈ 0.89

    - Recall ≈ 0.77
    - F1-score ≈ 0.83
    - ROC–AUC ≈ 0.96
- The DNN is competitive with XGBoost and only slightly behind LightGBM.
- Training curves show smooth convergence and no obvious overfitting thanks to class weights and early stopping.

Unsupervised / Anomaly Detection Models

- **Isolation Forest** and **Local Outlier Factor (LOF)**:
    - Very low precision, recall, and F1 (all < 0.20 for Isolation Forest, < 0.03 for LOF).
    - They misclassify many normal transactions and miss most frauds.
- **Autoencoder anomaly detector**:
    - High recall (≈ 0.77) but very low precision (≈ 0.12), F1 ≈ 0.20.
    - Flags too many normal transactions as suspicious.

Conclusion: On this labeled dataset, **supervised methods** are substantially more effective than purely unsupervised anomaly detection.

11.2 Supervised Model Performance (Validation Set)

Model	Accuracy	Precision	Recall	F1-score	ROC–AUC
Logistic Regression	0.9774	0.0636	0.8784	0.1186	0.9703
Linear SVM (LinearSVC)	0.9813	0.0758	0.8784	0.1396	0.9723
Naive Bayes	0.9775	0.0588	0.7973	0.1095	0.9554
k-NN (k = 5)	0.9993	0.8793	0.6892	0.7727	0.9188
Random Forest	0.9994	0.9808	0.6892	0.8095	0.9433
XGBoost	0.9994	0.8906	0.7703	0.8261	0.9797
LightGBM	0.9995	0.9194	0.7703	**0.8382**	0.9787
CatBoost	0.9988	0.6250	0.8108	0.7059	0.9641
DNN (Keras)	0.9994	0.8906	0.7703	0.8261	0.9613

Key takeaway:

- **LightGBM** has the best overall F1-score and precision.
- **XGBoost** and the **DNN** are very close, forming a strong cluster of top-performing models suitable for real-world deployment.

11.3 Anomaly Detection Model Performance (Validation Set)

Model	Precision	Recall	F1-score
Isolation Forest	0.1757	0.1757	0.1757
Local Outlier Factor	0.0270	0.0270	0.0270
Autoencoder	0.1178	0.7703	0.2043

These unsupervised approaches either miss most frauds (LOF) or generate a very high false-positive rate (Isolation Forest, Autoencoder).
They are useful as reference baselines but **not competitive** with the supervised models on this dataset.

11.4 Final Takeaway

The experiments demonstrate that:

- **Machine-learning–based supervised models** can effectively detect fraudulent credit card transactions even under extreme class imbalance.
- **LightGBM and XGBoost**, followed closely by the **DNN**, provide the best trade-off between recall (catching fraud) and precision (avoiding false alerts).
- Unsupervised anomaly detection methods are significantly less effective, reinforcing the choice to treat this as a supervised classification problem whenever labeled data are available.