```python
import pandas as pd
import numpy as np
import gc

print("Libraries loaded.")
```

```
Libraries loaded.
```

```python
# STEP 1 — Setup

!pip install pandas numpy matplotlib seaborn tqdm scikit-learn
```

```
Requirement already satisfied: pandas in
/usr/local/lib/python3.12/dist-packages (2.3.3)
Requirement already satisfied: numpy in
/usr/local/lib/python3.12/dist-packages (2.0.2)
Requirement already satisfied: matplotlib in
/usr/local/lib/python3.12/dist-packages (3.10.0)
Requirement already satisfied: seaborn in
/usr/local/lib/python3.12/dist-packages (0.13.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-
packages (4.67.1)
Requirement already satisfied: scikit-learn in
/usr/local/lib/python3.12/dist-packages (1.6.1)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.12/dist-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.12/dist-packages (from pandas) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in
/usr/local/lib/python3.12/dist-packages (from pandas) (2025.3)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.12/dist-packages (from matplotlib) (1.3.3)
Requirement already satisfied: cycler>=0.10 in
/usr/local/lib/python3.12/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.12/dist-packages (from matplotlib) (4.61.1)
Requirement already satisfied: kiwisolver>=1.3.1 in
/usr/local/lib/python3.12/dist-packages (from matplotlib) (1.4.9)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.12/dist-packages (from matplotlib) (25.0)
Requirement already satisfied: pillow>=8 in
/usr/local/lib/python3.12/dist-packages (from matplotlib) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.12/dist-packages (from matplotlib) (3.3.1)
Requirement already satisfied: scipy>=1.6.0 in
/usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.16.3)
Requirement already satisfied: joblib>=1.2.0 in
/usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.5.3)
Requirement already satisfied: threadpoolctl>=3.1.0 in
/usr/local/lib/python3.12/dist-packages (from scikit-learn) (3.6.0)
```

```
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.8.2-
>pandas) (1.17.0)
```

```
df = pd.read_csv('/kaggle/input/datasets/nirajrock/inductive-graph-
neural-network-framework-d/PS_20174392719_1491204439457_log.csv')
print("Shape:", df.shape)
df.head()
```

```
Shape: (6362620, 11)
```

```
   step      type    amount     nameOrig  oldbalanceOrg
newbalanceOrig  \
0     1   PAYMENT   9839.64  C1231006815       170136.0
160296.36
1     1   PAYMENT   1864.28  C1666544295        21249.0
19384.72
2     1  TRANSFER    181.00  C1305486145          181.0
0.00
3     1  CASH_OUT    181.00   C840083671          181.0
0.00
4     1   PAYMENT  11668.14  C2048537720        41554.0
29885.86

      nameDest  oldbalanceDest  newbalanceDest  isFraud
isFlaggedFraud
0  M1979787155             0.0             0.0        0
0
1  M2044282225             0.0             0.0        0
0
2   C553264065             0.0             0.0        1
0
3    C38997010         21182.0             0.0        1
0
4  M1230701703             0.0             0.0        0
0
```

```
print("Fraud Ratio:", df['isFraud'].mean())
print("Min step:", df['step'].min())
print("Max step:", df['step'].max())
```

```
Fraud Ratio: 0.001290820448180152
Min step: 1
Max step: 743
```

```
# STEP 3 — Memory Optimization (Very Important)

df['type'] = df['type'].astype('category')

for col in ['amount', 'oldbalanceOrg', 'newbalanceOrig',
            'oldbalanceDest', 'newbalanceDest']:
```

```
    df[col] = df[col].astype('float32')

for col in ['step', 'isFraud', 'isFlaggedFraud']:
    df[col] = df[col].astype('int32')

print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6362620 entries, 0 to 6362619
Data columns (total 11 columns):
 #   Column          Dtype
---  ------          -----
 0   step            int32
 1   type            category
 2   amount          float32
 3   nameOrig        object
 4   oldbalanceOrg   float32
 5   newbalanceOrig  float32
 6   nameDest        object
 7   oldbalanceDest  float32
 8   newbalanceDest  float32
 9   isFraud         int32
 10  isFlaggedFraud  int32
dtypes: category(1), float32(5), int32(3), object(2)
memory usage: 297.3+ MB
None
```

```
# STEP 4 — Temporal Split (Clean)

train_df = df[df['step'] <= 500].copy()
val_df   = df[(df['step'] > 500) & (df['step'] <= 600)].copy()
test_df  = df[df['step'] > 600].copy()

print("Train:", train_df.shape)
print("Val:", val_df.shape)
print("Test:", test_df.shape)
```

```
Train: (6061807, 11)
Val: (197240, 11)
Test: (103573, 11)
```

```
# STEP 5 — Free Unused Memory

del df
gc.collect()
```

```
190
```

```
# PHASE 2 — GRAPH CONSTRUCTION (Baby Step)

# STEP 1 — Encode Nodes (Unified Mapping)
# Combine unique accounts from train only
all_accounts = pd.concat([
    train_df['nameOrig'],
    train_df['nameDest']
]).unique()

account2id = {acc: idx for idx, acc in enumerate(all_accounts)}

print("Total unique nodes (train):", len(account2id))
```

```
Total unique nodes (train): 8639932
```

```
# STEP 2 — Build Edge Index (Train Only)
```

```
# Map to integer IDs
src = train_df['nameOrig'].map(account2id).values
dst = train_df['nameDest'].map(account2id).values

edge_index = np.vstack([src, dst])

print("Edge index shape:", edge_index.shape)
```

```
Edge index shape: (2, 6061807)
```

```
# STEP 3 — Convert to Torch Tensors (Memory-Safe)

import torch

edge_index = torch.tensor(edge_index, dtype=torch.long)

print(edge_index.shape)
```

```
torch.Size([2, 6061807])
```

```
# STEP 4 — Create Edge LabelsDo NOT Create Node Feature Matrix Yet

edge_labels = torch.tensor(train_df['isFraud'].values,
dtype=torch.float32)

print(edge_labels.shape)
```

```
torch.Size([6061807])
```

```
# Install PyG

!pip install torch-geometric
```

```
Collecting torch-geometric
  Downloading torch_geometric-2.7.0-py3-none-any.whl.metadata (63 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 63.7/63.7 kB 2.5 MB/s eta
0:00:00
ent already satisfied: aiohttp in /usr/local/lib/python3.12/dist-
packages (from torch-geometric) (3.13.3)
```

```
Requirement already satisfied: fsspec in
/usr/local/lib/python3.12/dist-packages (from torch-geometric)
(2025.3.0)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.12/dist-packages (from torch-geometric) (3.1.6)
Requirement already satisfied: numpy in
/usr/local/lib/python3.12/dist-packages (from torch-geometric) (2.0.2)
Requirement already satisfied: psutil>=5.8.0 in
/usr/local/lib/python3.12/dist-packages (from torch-geometric) (5.9.5)
Requirement already satisfied: pyparsing in
/usr/local/lib/python3.12/dist-packages (from torch-geometric) (3.3.1)
Requirement already satisfied: requests in
/usr/local/lib/python3.12/dist-packages (from torch-geometric)
(2.32.4)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-
packages (from torch-geometric) (4.67.1)
Requirement already satisfied: xxhash in
/usr/local/lib/python3.12/dist-packages (from torch-geometric) (3.6.0)
Requirement already satisfied: aiohappyeyeballs>=2.5.0 in
/usr/local/lib/python3.12/dist-packages (from aiohttp->torch-
geometric) (2.6.1)
Requirement already satisfied: aiosignal>=1.4.0 in
/usr/local/lib/python3.12/dist-packages (from aiohttp->torch-
geometric) (1.4.0)
Requirement already satisfied: attrs>=17.3.0 in
/usr/local/lib/python3.12/dist-packages (from aiohttp->torch-
geometric) (25.4.0)
Requirement already satisfied: frozenlist>=1.1.1 in
/usr/local/lib/python3.12/dist-packages (from aiohttp->torch-
geometric) (1.8.0)
Requirement already satisfied: multidict<7.0,>=4.5 in
/usr/local/lib/python3.12/dist-packages (from aiohttp->torch-
geometric) (6.7.0)
Requirement already satisfied: propcache>=0.2.0 in
/usr/local/lib/python3.12/dist-packages (from aiohttp->torch-
geometric) (0.4.1)
Requirement already satisfied: yarl<2.0,>=1.17.0 in
/usr/local/lib/python3.12/dist-packages (from aiohttp->torch-
geometric) (1.22.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.12/dist-packages (from jinja2->torch-geometric)
(3.0.3)
Requirement already satisfied: charset_normalizer<4,>=2 in
/usr/local/lib/python3.12/dist-packages (from requests->torch-
geometric) (3.4.4)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.12/dist-packages (from requests->torch-
geometric) (3.11)
Requirement already satisfied: urllib3<3,>=1.21.1 in
```

```
/usr/local/lib/python3.12/dist-packages (from requests->torch-
geometric) (2.5.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.12/dist-packages (from requests->torch-
geometric) (2026.1.4)
Requirement already satisfied: typing-extensions>=4.2 in
/usr/local/lib/python3.12/dist-packages (from aiosignal>=1.4.0-
>aiohttp->torch-geometric) (4.15.0)
Downloading torch_geometric-2.7.0-py3-none-any.whl (1.3 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 23.5 MB/s eta
0:00:00a 0:00:01
etric
Successfully installed torch-geometric-2.7.0
```

```python
num_nodes = len(account2id)

x = torch.ones((num_nodes, 1), dtype=torch.float32)

# PHASE 2 — Build PyG Data Object

# STEP 1 — Import PyG Data

from torch_geometric.data import Data

# STEP 2 — Create Data Object

data = Data(
    x=x,
    edge_index=edge_index
)

# Attach edge labels separately
data.edge_label = edge_labels

print(data)
```

```
Data(x=[8639932, 1], edge_index=[2, 6061807], edge_label=[6061807])
```

```python
# TRAIN/VAL EDGE SPLIT

# STEP 1 — Create Train Edge Indices

num_edges = data.edge_index.size(1)

train_edge_idx = torch.arange(num_edges)

print("Total train edges:", len(train_edge_idx))
```

```
Total train edges: 6061807
```

```python
# STEP 2 — Create Small Validation Subset
```

```python
val_size = 50000

perm = torch.randperm(num_edges)
val_edge_idx = perm[:val_size]
train_edge_idx = perm[val_size:]

print("Train edges:", len(train_edge_idx))
print("Val edges:", len(val_edge_idx))
```

```
Train edges: 6011807
Val edges: 50000
```

```python
# PHASE 3 — SCALABLE EDGE SAMPLING

import torch
import torch_geometric

print("Torch:", torch.__version__)
print("PyG:", torch_geometric.__version__)
```

```
Torch: 2.9.0+cu126
PyG: 2.7.0
```

```python
# STEP 1 — Build Fast Adjacency Map

from collections import defaultdict

adj = defaultdict(list)

src_nodes = edge_index[0].numpy()
dst_nodes = edge_index[1].numpy()

for s, d in zip(src_nodes, dst_nodes):
    adj[s].append(d)
    adj[d].append(s)  # make undirected for message passing

print("Adjacency built.")
```

```
Adjacency built.
```

```python
# STEP 2 — Create Edge Mini-Batch Function

import random

def sample_subgraph(edge_ids, num_neighbors=10):

    batch_src = edge_index[0, edge_ids]
    batch_dst = edge_index[1, edge_ids]

    nodes = set(batch_src.tolist() + batch_dst.tolist())

    # 1-hop neighbors
    for n in list(nodes):
        neighbors = adj[n]
        sampled = random.sample(neighbors, min(len(neighbors),
num_neighbors))
        nodes.update(sampled)

    nodes = list(nodes)
    node_map = {n:i for i,n in enumerate(nodes)}

    # build subgraph edges
    sub_edges = []
    for n in nodes:
        for nbr in adj[n]:
            if nbr in node_map:
                sub_edges.append([node_map[n], node_map[nbr]])

    sub_edge_index = torch.tensor(sub_edges,
dtype=torch.long).t().contiguous()

    return nodes, sub_edge_index

# STEP 3 — Test One Batch

batch_ids = train_edge_idx[:2048]

nodes, sub_edge_index = sample_subgraph(batch_ids)

print("Subgraph nodes:", len(nodes))
print("Subgraph edges:", sub_edge_index.shape)
```

```
Subgraph nodes: 14533
Subgraph edges: torch.Size([2, 24978])
```

```
# Integrate Model with Manual Sampler

# STEP 1 — Define Model (Simple + Safe)

import torch.nn as nn
from torch_geometric.nn import SAGEConv

class EdgeGNN(nn.Module):
    def __init__(self, hidden_dim=32):
        super().__init__()

        self.conv1 = SAGEConv(1, hidden_dim)
        self.conv2 = SAGEConv(hidden_dim, hidden_dim)

        self.edge_mlp = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, x, edge_index, edge_label_index):

        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)

        src, dst = edge_label_index
        edge_emb = torch.cat([x[src], x[dst]], dim=1)

        return self.edge_mlp(edge_emb).squeeze()

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = EdgeGNN().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.BCEWithLogitsLoss()

print("Model ready on:", device)
```

```
Model ready on: cuda
```

```
# STEP 2 — Upgrade Sampler (Critical Fix)

def sample_subgraph_with_labels(edge_ids, num_neighbors=10):

    batch_src = edge_index[0, edge_ids]
    batch_dst = edge_index[1, edge_ids]
    batch_labels = edge_labels[edge_ids]

    nodes = set(batch_src.tolist() + batch_dst.tolist())

    for n in list(nodes):
        neighbors = adj[n]
        sampled = random.sample(neighbors, min(len(neighbors),
num_neighbors))
        nodes.update(sampled)

    nodes = list(nodes)
    node_map = {n:i for i,n in enumerate(nodes)}

    # Build subgraph edges
    sub_edges = []
    for n in nodes:
        for nbr in adj[n]:
            if nbr in node_map:
                sub_edges.append([node_map[n], node_map[nbr]])

    sub_edge_index = torch.tensor(sub_edges,
dtype=torch.long).t().contiguous()

    # Map supervision edges to local indices
    local_src = torch.tensor([node_map[n.item()] for n in batch_src])
    local_dst = torch.tensor([node_map[n.item()] for n in batch_dst])
    edge_label_index = torch.stack([local_src, local_dst])

    return nodes, sub_edge_index, edge_label_index, batch_labels
```

```
# STEP 3 — Single Training Step Test

model.train()

batch_ids = train_edge_idx[:2048]

nodes, sub_edge_index, edge_label_index, batch_labels = \
    sample_subgraph_with_labels(batch_ids)

# Move to device
sub_edge_index = sub_edge_index.to(device)
edge_label_index = edge_label_index.to(device)
batch_labels = batch_labels.to(device)

# Node features
x_sub = torch.ones((len(nodes), 1), dtype=torch.float32).to(device)

optimizer.zero_grad()

logits = model(x_sub, sub_edge_index, edge_label_index)

loss = criterion(logits, batch_labels)

loss.backward()
optimizer.step()

print("Single step loss:", loss.item())
```

```
Single step loss: 0.6302144527435303
```

```
# Controlled Training Loop

# STEP 1 — Replace Loss with Weighted BCE

pos_weight = (len(edge_labels) - edge_labels.sum()) /
edge_labels.sum()
pos_weight = pos_weight.to(device)

criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
print("Pos weight:", pos_weight.item())
```

```
Pos weight: 1089.0570068359375
```

```
# STEP 2 — Small Epoch Training (200 Batches Only)

import tqdm

model.train()

num_batches = 200
batch_size = 2048

for i in tqdm.tqdm(range(num_batches)):

    start = i * batch_size
    end = start + batch_size

    batch_ids = train_edge_idx[start:end]

    nodes, sub_edge_index, edge_label_index, batch_labels = \
        sample_subgraph_with_labels(batch_ids)

    sub_edge_index = sub_edge_index.to(device)
    edge_label_index = edge_label_index.to(device)
    batch_labels = batch_labels.to(device)

    x_sub = torch.ones((len(nodes), 1),
dtype=torch.float32).to(device)

    optimizer.zero_grad()

    logits = model(x_sub, sub_edge_index, edge_label_index)
    loss = criterion(logits, batch_labels)

    loss.backward()
    optimizer.step()

    if i % 20 == 0:
        print(f"Batch {i}, Loss: {loss.item():.4f}")
```

```
  2%|▏         | 4/200 [00:00<00:11, 17.71it/s]

Batch 0, Loss: 1.4470

 12%|█▏        | 24/200 [00:01<00:09, 18.74it/s]

Batch 20, Loss: 1.4851

 22%|██▏       | 44/200 [00:02<00:08, 18.82it/s]

Batch 40, Loss: 1.4361

 32%|███▏      | 64/200 [00:05<00:10, 13.48it/s]

Batch 60, Loss: 1.0284
```

```
 42%|████      | 84/200 [00:06<00:06, 18.47it/s]

Batch 80, Loss: 1.4306

 52%|█████     | 104/200 [00:07<00:05, 18.96it/s]

Batch 100, Loss: 1.7963

 62%|██████    | 124/200 [00:08<00:04, 18.31it/s]

Batch 120, Loss: 1.0696

 72%|███████   | 144/200 [00:09<00:02, 18.78it/s]

Batch 140, Loss: 1.7963

 82%|████████  | 164/200 [00:12<00:02, 12.20it/s]

Batch 160, Loss: 1.0548

 92%|█████████ | 184/200 [00:13<00:00, 18.83it/s]

Batch 180, Loss: 2.2001

100%|██████████| 200/200 [00:13<00:00, 14.33it/s]
```

```python
# STEP 1 — Select Fraud Seeds (Train Only)

fraud_edge_ids = train_edge_idx[
    edge_labels[train_edge_idx] == 1
]

print("Total fraud edges in train:", len(fraud_edge_ids))
```

```
Total fraud edges in train: 5511
```

```python
# STEP 2 — Define Injection Parameters

num_mules = 100          # number of synthetic mule nodes
origins_per_mule = 20    # fraud origins per mule

# STEP 3 — Create New Mule Node IDs

current_max_node = edge_index.max().item()
new_mule_ids = torch.arange(
    current_max_node + 1,
    current_max_node + 1 + num_mules
)

print("New mule node range:",
      new_mule_ids[0].item(),
      "to",
      new_mule_ids[-1].item())
```

```
New mule node range: 8639932 to 8640031
```

```python
# STEP 4 — Inject Collusion Edges

import random

injected_edges = []
injected_labels = []

fraud_src_nodes = edge_index[0, fraud_edge_ids].tolist()

for i, mule_id in enumerate(new_mule_ids):

    sampled_origins = random.sample(
        fraud_src_nodes,
        origins_per_mule
    )

    for origin in sampled_origins:
        injected_edges.append([origin, mule_id.item()])
        injected_labels.append(1.0)

injected_edges = torch.tensor(injected_edges, dtype=torch.long).t()
injected_labels = torch.tensor(injected_labels, dtype=torch.float32)

print("Injected edges:", injected_edges.shape)
```

```
Injected edges: torch.Size([2, 2000])
```

```python
# STEP 5 — Merge Into Graph

edge_index = torch.cat([edge_index, injected_edges], dim=1)
edge_labels = torch.cat([edge_labels, injected_labels])

print("New total edges:", edge_index.shape[1])
```

```
New total edges: 6063807
```

```python
# STEP 6 — Update Adjacency

for s, d in injected_edges.t().tolist():
    adj[s].append(d)
    adj[d].append(s)

# STEP 7 — Update Train Edge Index

new_edge_start = train_edge_idx.max().item() + 1
new_edge_ids = torch.arange(
    edge_index.shape[1] - injected_edges.shape[1],
    edge_index.shape[1]
)
```

```
train_edge_idx = torch.cat([train_edge_idx, new_edge_ids])

print("Updated train edges:", len(train_edge_idx))
```

```
Updated train edges: 6013807
```

```
# STEP 1 — Reset Model

model = EdgeGNN().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Recompute pos_weight because edges increased
pos_weight = (len(edge_labels) - edge_labels.sum()) /
edge_labels.sum()
pos_weight = pos_weight.to(device)

criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)

print("New pos_weight:", pos_weight.item())
```

```
New pos_weight: 800.9848022460938
```

```
# STEP 2 — Controlled 200 Batch Training

import tqdm
import numpy as np

model.train()

num_batches = 200
batch_size = 2048

losses = []

for i in tqdm.tqdm(range(num_batches)):

    start = i * batch_size
    end = start + batch_size

    batch_ids = train_edge_idx[start:end]

    nodes, sub_edge_index, edge_label_index, batch_labels = \
        sample_subgraph_with_labels(batch_ids)

    sub_edge_index = sub_edge_index.to(device)
    edge_label_index = edge_label_index.to(device)
    batch_labels = batch_labels.to(device)

    x_sub = torch.ones((len(nodes), 1),
dtype=torch.float32).to(device)
```

```
    optimizer.zero_grad()

    logits = model(x_sub, sub_edge_index, edge_label_index)
    loss = criterion(logits, batch_labels)

    loss.backward()
    optimizer.step()

    losses.append(loss.item())

    if i % 20 == 0:
        print(f"Batch {i}, Loss: {loss.item():.4f}")

print("Average Loss:", np.mean(losses))
print("First 10 losses:", losses[:10])
print("Last 10 losses:", losses[-10:])
```

```
  2%|▏         | 4/200 [00:00<00:11, 17.57it/s]

Batch 0, Loss: 1.2759

 12%|█▏        | 24/200 [00:01<00:09, 18.17it/s]

Batch 20, Loss: 1.2313

 22%|██▏       | 44/200 [00:02<00:09, 16.89it/s]

Batch 40, Loss: 1.2219

 32%|███▏      | 64/200 [00:05<00:15,  8.85it/s]

Batch 60, Loss: 0.8535

 42%|████▏     | 84/200 [00:06<00:06, 18.09it/s]

Batch 80, Loss: 1.2214

 52%|█████▏    | 104/200 [00:07<00:05, 18.60it/s]

Batch 100, Loss: 1.5431

 62%|██████▏   | 124/200 [00:08<00:04, 18.14it/s]

Batch 120, Loss: 0.9009

 72%|███████▏  | 144/200 [00:09<00:02, 18.67it/s]

Batch 140, Loss: 1.5657

 82%|████████▏ | 164/200 [00:12<00:04,  7.29it/s]

Batch 160, Loss: 0.8773

 92%|█████████▏| 184/200 [00:13<00:00, 17.90it/s]
```

```
Batch 180, Loss: 1.9560

100%|██████████| 200/200 [00:14<00:00, 14.20it/s]

Average Loss: 1.1516465798020363
First 10 losses: [1.2759370803833008, 0.7678236961364746,
0.7167457342147827, 0.6670475006103516, 0.6199246644973755,
0.5813686847686768, 0.885597376823425, 0.5179439783096313,
1.2319939136505127, 0.8511680364608765]
Last 10 losses: [1.60493803024292, 1.2321748733520508,
0.4875519871711731, 1.2327152490615845, 1.60843026638031,
0.8586360216140747, 0.8587345480918884, 0.4836242198944092,
1.612107515335083, 1.9914066791534424]
```

```
# Validation Evaluation

model.eval()

all_preds = []
all_labels = []

with torch.no_grad():

    num_val_batches = 10
    batch_size = 1024

    for i in range(num_val_batches):

        start = i * batch_size
        end = start + batch_size

        batch_ids = val_edge_idx[start:end]

        nodes, sub_edge_index, edge_label_index, batch_labels = \
            sample_subgraph_with_labels(batch_ids)

        sub_edge_index = sub_edge_index.to(device)
        edge_label_index = edge_label_index.to(device)
        batch_labels = batch_labels.to(device)

        x_sub = torch.ones((len(nodes), 1),
dtype=torch.float32).to(device)

        logits = model(x_sub, sub_edge_index, edge_label_index)
        probs = torch.sigmoid(logits)

        all_preds.extend(probs.cpu().numpy())
        all_labels.extend(batch_labels.cpu().numpy())
```

```python
pr_auc = average_precision_score(all_labels, all_preds)

print("Validation PR-AUC:", pr_auc)
```

```
Validation PR-AUC: 0.001171875
```

```python
# PHASE CLEAN REBUILD (Minimal Steps Only)
 # STEP 1 — Load + Optimize (Same As Before)

import pandas as pd
import numpy as np
import torch
import gc

df = pd.read_csv('/kaggle/input/datasets/nirajrock/inductive-graph-
neural-network-framework-d/PS_20174392719_1491204439457_log.csv')

# Memory optimize
df['type'] = df['type'].astype('category')

for col in ['amount','oldbalanceOrg','newbalanceOrig',
            'oldbalanceDest','newbalanceDest']:
    df[col] = df[col].astype('float32')

for col in ['step','isFraud','isFlaggedFraud']:
    df[col] = df[col].astype('int32')

print(df.shape)
```

```
(6362620, 11)
```

```python
# STEP 2 — Temporal Split

train_df = df[df['step'] <= 500].copy()
val_df   = df[(df['step'] > 500) & (df['step'] <= 600)].copy()
test_df  = df[df['step'] > 600].copy()

del df
gc.collect()

print(train_df.shape, val_df.shape, test_df.shape)
```

```
(6061807, 11) (197240, 11) (103573, 11)
```

```python
# STEP 3 — Build Graph ONLY From Train

all_accounts = pd.concat([
    train_df['nameOrig'],
    train_df['nameDest']
]).unique()

account2id = {acc: idx for idx, acc in enumerate(all_accounts)}
```

```python
src = train_df['nameOrig'].map(account2id).values
dst = train_df['nameDest'].map(account2id).values

edge_index = torch.tensor(
    np.vstack([src, dst]),
    dtype=torch.long
)

edge_labels = torch.tensor(
    train_df['isFraud'].values,
    dtype=torch.float32
)

print("Edges:", edge_index.shape)
print("Nodes:", len(account2id))
```

```
Edges: torch.Size([2, 6061807])
Nodes: 8639932
```

```python
# STEP 4 — STORE CLEAN BOUNDARY

original_train_edge_count = edge_index.shape[1]
print("Original train edges:", original_train_edge_count)
```

```
Original train edges: 6061807
```

```python
# STEP 5 — Build Adjacency (Clean)

from collections import defaultdict

adj = defaultdict(list)

src_nodes = edge_index[0].tolist()
dst_nodes = edge_index[1].tolist()

for s, d in zip(src_nodes, dst_nodes):
    adj[s].append(d)
    adj[d].append(s)

print("Adjacency ready.")
```

```
Adjacency ready.
```

```python
# Step 1 — Identify Late Train Edges (Clean Way)

late_train_mask = (
    (train_df['step'] >= 450) &
    (train_df['step'] <= 500)
)

late_train_edge_ids = torch.arange(original_train_edge_count)
```

```
[late_train_mask.values]

print("Late train edges:", len(late_train_edge_ids))
```

```
Late train edges: 47106
```

```
# TEMPORAL INJECTION — TRAIN PHASE (Controlled)

# Step 2 — Get Late-Train Fraud Origins

# Identify fraud edges inside late window
late_train_fraud_mask = (
    (train_df['step'] >= 450) &
    (train_df['step'] <= 500) &
    (train_df['isFraud'] == 1)
)

late_train_fraud_edge_ids = torch.arange(original_train_edge_count)[
    late_train_fraud_mask.values
]

print("Late train fraud edges:", len(late_train_fraud_edge_ids))
```

```
Late train fraud edges: 520
```

```
# STEP 3 — TRAIN TEMPORAL COLLUSION INJECTION

# Injection Parameters
num_train_mules = 100
origins_per_mule = 5   # 100 × 5 = 500 edges

# Create Mule Node IDs

current_max_node = edge_index.max().item()

new_train_mule_ids = torch.arange(
    current_max_node + 1,
    current_max_node + 1 + num_train_mules
)

print("Train mule node range:",
      new_train_mule_ids[0].item(),
      "to",
      new_train_mule_ids[-1].item())
```

```
Train mule node range: 8639932 to 8640031
```

```
# Sample Fraud Origins

import random

late_fraud_src_nodes = edge_index[0,
late_train_fraud_edge_ids].tolist()

injected_train_edges = []
injected_train_labels = []

for mule_id in new_train_mule_ids:

    sampled_origins = random.sample(
        late_fraud_src_nodes,
        origins_per_mule
    )

    for origin in sampled_origins:
        injected_train_edges.append([origin, mule_id.item()])
        injected_train_labels.append(1.0)

injected_train_edges = torch.tensor(
    injected_train_edges,
    dtype=torch.long
).t()

injected_train_labels = torch.tensor(
    injected_train_labels,
    dtype=torch.float32
)

print("Injected train edges:", injected_train_edges.shape)
```

```
Injected train edges: torch.Size([2, 500])
```

```
# Merge Into Graph

edge_index = torch.cat([edge_index, injected_train_edges], dim=1)
edge_labels = torch.cat([edge_labels, injected_train_labels])

print("New total edges:", edge_index.shape[1])
```

```
New total edges: 6062307
```

```
# Update Adjacency

for s, d in injected_train_edges.t().tolist():
    adj[s].append(d)
    adj[d].append(s)

print("Adjacency updated.")
```

```
Adjacency updated.
```

```
# TEMPORAL CONTINUATION IN VALIDATION

 #  4 — Identify Early Validation Fraud Edges
```

```
early_val_mask = (
    (val_df['step'] >= 510) &
    (val_df['step'] <= 540)
)

early_val_edge_count = early_val_mask.sum()

print("Early validation edges:", early_val_edge_count)
```

```
Early validation edges: 62151
```

```
# STEP 5 — Identify Early Validation Fraud Edges

early_val_fraud_mask = (
    (val_df['step'] >= 510) &
    (val_df['step'] <= 540) &
    (val_df['isFraud'] == 1)
)

early_val_fraud_count = early_val_fraud_mask.sum()

print("Early validation fraud edges:", early_val_fraud_count)
```

```
Early validation fraud edges: 320
```

```
# Early validation fraud edges: 320

# STEP 6 — VALIDATION CONTINUATION INJECTION

num_val_mules = 50
origins_per_val_mule = 4

# Create Validation Mule Node IDs

current_max_node = edge_index.max().item()

new_val_mule_ids = torch.arange(
    current_max_node + 1,
    current_max_node + 1 + num_val_mules
)

print("Validation mule node range:",
      new_val_mule_ids[0].item(),
      "to",
      new_val_mule_ids[-1].item())
```

```
Validation mule node range: 8640032 to 8640081
```

```
# Sample Early Validation Fraud Origins

early_val_fraud_origins = val_df.loc[
    early_val_fraud_mask, 'nameOrig'
]

# Keep only those already in train graph
early_val_fraud_origins = [
    account2id[o]
    for o in early_val_fraud_origins
    if o in account2id
]

print("Mapped early val fraud origins:", len(early_val_fraud_origins))
```

```
Mapped early val fraud origins: 1
```

```
# STEP 1 — Create Persistent Mule Nodes
```

```
original_train_edge_count
```

```
6061807
```

```
edge_index = edge_index[:, :original_train_edge_count]
edge_labels = edge_labels[:original_train_edge_count]

print("Rolled back edges:", edge_index.shape[1])
```

```
Rolled back edges: 6061807
```

```
from collections import defaultdict

adj = defaultdict(list)

src_nodes = edge_index[0].tolist()
dst_nodes = edge_index[1].tolist()

for s, d in zip(src_nodes, dst_nodes):
    adj[s].append(d)
    adj[d].append(s)

print("Adjacency rebuilt clean.")
```

```
Adjacency rebuilt clean.
```

```
# FINAL PERSISTENT COLLUSION DESIGN
```

```
# STEP 1 — Create Persistent Mule Nodes (Only Once)

num_persistent_mules = 50
origins_per_mule_train = 6

current_max_node = edge_index.max().item()

persistent_mule_ids = torch.arange(
    current_max_node + 1,
    current_max_node + 1 + num_persistent_mules
)

print("Persistent mule node range:",
      persistent_mule_ids[0].item(),
      "to",
      persistent_mule_ids[-1].item())
```

```
Persistent mule node range: 8639932 to 8639981
```

```
# STEP 2 — Get Late-Train Fraud Origins (Fresh, Clean)

late_train_fraud_mask = (
    (train_df['step'] >= 450) &
    (train_df['step'] <= 500) &
    (train_df['isFraud'] == 1)
)

late_train_fraud_edge_ids = torch.arange(original_train_edge_count)[
    late_train_fraud_mask.values
]

print("Late train fraud edges:", len(late_train_fraud_edge_ids))
```

```
Late train fraud edges: 520
```

```
# STEP 3 — Inject Persistent Ring (Train Phase)

import random

origins_per_mule_train = 6

late_fraud_src_nodes = edge_index[0,
late_train_fraud_edge_ids].tolist()

train_injected_edges = []
train_injected_labels = []

for mule_id in persistent_mule_ids:

    sampled_origins = random.sample(
        late_fraud_src_nodes,
        origins_per_mule_train
    )

    for origin in sampled_origins:
        train_injected_edges.append([origin, mule_id.item()])
        train_injected_labels.append(1.0)

train_injected_edges = torch.tensor(
    train_injected_edges,
    dtype=torch.long
).t()
```

```
train_injected_labels = torch.tensor(
    train_injected_labels,
    dtype=torch.float32
)

print("Train injected edges:", train_injected_edges.shape)
```

```
Train injected edges: torch.Size([2, 300])
```

```
# STEP 4 — Merge Into Graph (Train Phase)

edge_index = torch.cat([edge_index, train_injected_edges], dim=1)
edge_labels = torch.cat([edge_labels, train_injected_labels])

print("Edges after train injection:", edge_index.shape[1])
```

```
Edges after train injection: 6062107
```

```
# STEP 5 — Rebuild Adjacency (Clean Update)

from collections import defaultdict

adj = defaultdict(list)

src_nodes = edge_index[0].tolist()
dst_nodes = edge_index[1].tolist()

for s, d in zip(src_nodes, dst_nodes):
    adj[s].append(d)
    adj[d].append(s)

print("Adjacency rebuilt with persistent ring.")
```

```
Adjacency rebuilt with persistent ring.
```

```
# PHASE T2 — VALIDATION CONTINUATION (CRITICAL)

# STEP 1 — Identify Early Validation Fraud Origins

early_val_fraud_mask = (
    (val_df['step'] >= 510) &
    (val_df['step'] <= 540) &
    (val_df['isFraud'] == 1)
)

early_val_fraud_accounts = val_df.loc[
    early_val_fraud_mask, 'nameOrig'
].unique()

print("Unique early val fraud accounts:",
len(early_val_fraud_accounts))
```

```
Unique early val fraud accounts: 320
```

```python
# PHASE T2 — VALIDATION CONTINUATION INJECTION

# STEP 2 — Create New Node IDs for Validation Origins

import random

origins_per_mule_val = 4

# Start new node range AFTER current graph max
current_max_node = edge_index.max().item()

val_origin_node_ids = torch.arange(
    current_max_node + 1,
    current_max_node + 1 + len(early_val_fraud_accounts)
)

print("Validation origin node range:",
      val_origin_node_ids[0].item(),
      "to",
      val_origin_node_ids[-1].item())
```

```
Validation origin node range: 8639982 to 8640301
```

```python
# STEP 3 — Build Mapping (Val Fraud Account → New Node ID)

val_account_to_node = {
    acc: node_id.item()
    for acc, node_id in zip(
        early_val_fraud_accounts,
        val_origin_node_ids
    )
}

print("Validation origin mapping size:", len(val_account_to_node))
```

```
Validation origin mapping size: 320
```

```python
# STEP 4 — Inject Continuation Edges (Persistent Mules Grow)

val_injected_edges = []
val_injected_labels = []

for mule_id in persistent_mule_ids:

    sampled_accounts = random.sample(
        list(val_account_to_node.keys()),
        origins_per_mule_val
    )

    for acc in sampled_accounts:
        origin_node = val_account_to_node[acc]
        val_injected_edges.append([origin_node, mule_id.item()])
        val_injected_labels.append(1.0)

val_injected_edges = torch.tensor(
    val_injected_edges,
    dtype=torch.long
).t()

val_injected_labels = torch.tensor(
    val_injected_labels,
    dtype=torch.float32
)

print("Validation injected edges:", val_injected_edges.shape)
```

```
Validation injected edges: torch.Size([2, 200])
```

```
# STEP 5 — Merge Into Graph

edge_index = torch.cat([edge_index, val_injected_edges], dim=1)
edge_labels = torch.cat([edge_labels, val_injected_labels])

print("Edges after validation injection:", edge_index.shape[1])
```

```
Edges after validation injection: 6062307
```

```
# STEP 6 — Rebuild Adjacency (Final Graph State)

from collections import defaultdict

adj = defaultdict(list)

src_nodes = edge_index[0].tolist()
dst_nodes = edge_index[1].tolist()

for s, d in zip(src_nodes, dst_nodes):
    adj[s].append(d)
    adj[d].append(s)

print("Adjacency rebuilt with persistent ring expansion.")
```

```
Adjacency rebuilt with persistent ring expansion.
```

```
# Now Critical Step: Proper Training Setup

# STEP 1 — Recreate Edge Splits Cleanly

train_edge_cutoff = original_train_edge_count +
train_injected_edges.shape[1]

train_edge_idx = torch.arange(train_edge_cutoff)
```

```python
val_edge_idx = torch.arange(
    train_edge_cutoff,
    edge_index.shape[1]
)

print("Train edges:", len(train_edge_idx))
print("Val edges:", len(val_edge_idx))
```

```
Train edges: 6062107
Val edges: 200
```

```python
# STEP 2 — Reset Model

model = EdgeGNN().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

pos_weight = (len(edge_labels[train_edge_idx]) -
              edge_labels[train_edge_idx].sum()) / \
             edge_labels[train_edge_idx].sum()

pos_weight = pos_weight.to(device)

criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)

print("New pos_weight:", pos_weight.item())
```

```
New pos_weight: 1033.312744140625
```

```python
# Now Train Properly (Controlled)

import tqdm
import numpy as np

model.train()

num_batches = 200
batch_size = 2048

losses = []

for i in tqdm.tqdm(range(num_batches)):

    start = i * batch_size
    end = start + batch_size

    batch_ids = train_edge_idx[start:end]

    nodes, sub_edge_index, edge_label_index, batch_labels = \
        sample_subgraph_with_labels(batch_ids)

    sub_edge_index = sub_edge_index.to(device)
    edge_label_index = edge_label_index.to(device)
    batch_labels = batch_labels.to(device)

    x_sub = torch.ones((len(nodes), 1),
dtype=torch.float32).to(device)

    optimizer.zero_grad()

    logits = model(x_sub, sub_edge_index, edge_label_index)
    loss = criterion(logits, batch_labels)

    loss.backward()
    optimizer.step()

    losses.append(loss.item())

    if i % 20 == 0:
        print(f"Batch {i}, Loss: {loss.item():.4f}")

print("Average Loss:", np.mean(losses))
```

```
  2%|▏         | 4/200 [00:00<00:06, 31.51it/s]

Batch 0, Loss: 5.9952

 12%|█▏        | 23/200 [00:01<00:09, 18.96it/s]

Batch 20, Loss: 0.9314

 22%|██▏       | 43/200 [00:02<00:09, 16.86it/s]

Batch 40, Loss: 0.7679

 32%|███▏      | 63/200 [00:03<00:08, 16.19it/s]

Batch 60, Loss: 0.5903

 42%|████▏     | 83/200 [00:06<00:11, 10.11it/s]

Batch 80, Loss: 0.4695

 52%|█████▏    | 103/200 [00:08<00:06, 15.70it/s]

Batch 100, Loss: 0.4099

 62%|██████▏   | 123/200 [00:09<00:04, 15.44it/s]

Batch 120, Loss: 0.3158

 72%|███████▏  | 143/200 [00:10<00:03, 15.28it/s]

Batch 140, Loss: 1.6847
```

```
 82%|████████  | 163/200 [00:13<00:05,  7.36it/s]

Batch 160, Loss: 1.6220

 92%|█████████ | 183/200 [00:15<00:01, 14.97it/s]

Batch 180, Loss: 1.7315

100%|██████████| 200/200 [00:16<00:00, 12.24it/s]

Average Loss: 0.958220670670271



# Then Evaluate ONLY On Continuation Edges

from sklearn.metrics import average_precision_score

model.eval()

all_preds = []
all_labels = []

with torch.no_grad():

    batch_ids = val_edge_idx

    nodes, sub_edge_index, edge_label_index, batch_labels = \
        sample_subgraph_with_labels(batch_ids)

    sub_edge_index = sub_edge_index.to(device)
    edge_label_index = edge_label_index.to(device)
    batch_labels = batch_labels.to(device)

    x_sub = torch.ones((len(nodes), 1),
dtype=torch.float32).to(device)

    logits = model(x_sub, sub_edge_index, edge_label_index)
    probs = torch.sigmoid(logits)

    all_preds.extend(probs.cpu().numpy())
    all_labels.extend(batch_labels.cpu().numpy())

pr_auc = average_precision_score(all_labels, all_preds)

print("Validation PR-AUC (persistent continuation):", pr_auc)

Validation PR-AUC (persistent continuation): 1.0

# STEP 1 — Create Balanced Validation Set"
```

```
import torch

# 1 Get non-fraud edges from TRAIN
non_fraud_train_edges = train_edge_idx[
    edge_labels[train_edge_idx] == 0
]

# 2 Randomly sample 200 negatives
random_negatives = non_fraud_train_edges[
    torch.randperm(len(non_fraud_train_edges))[:200]
]

# 3 Combine with injected continuation edges
balanced_val_edge_idx = torch.cat([
    val_edge_idx,        # 200 injected fraud edges
    random_negatives     # 200 non-fraud edges
])

print("Balanced validation size:", len(balanced_val_edge_idx))
```

```
Balanced validation size: 400
```

```
# STEP 2 — Evaluate Model on Balanced Set

from sklearn.metrics import average_precision_score, roc_auc_score

model.eval()

all_preds = []
all_labels = []

with torch.no_grad():

    batch_ids = balanced_val_edge_idx

    nodes, sub_edge_index, edge_label_index, batch_labels = \
        sample_subgraph_with_labels(batch_ids)

    sub_edge_index = sub_edge_index.to(device)
    edge_label_index = edge_label_index.to(device)
    batch_labels = batch_labels.to(device)

    x_sub = torch.ones((len(nodes), 1),
dtype=torch.float32).to(device)

    logits = model(x_sub, sub_edge_index, edge_label_index)
    probs = torch.sigmoid(logits)

    all_preds.extend(probs.cpu().numpy())
    all_labels.extend(batch_labels.cpu().numpy())
```

```
# Compute metrics
pr_auc = average_precision_score(all_labels, all_preds)
roc_auc = roc_auc_score(all_labels, all_preds)

print("Balanced Validation PR-AUC:", pr_auc)
print("Balanced Validation ROC-AUC:", roc_auc)
```

```
Balanced Validation PR-AUC: 0.5
Balanced Validation ROC-AUC: 0.11499999999999999
```

```
# STEP 1 — Compute Node Features (Train Only)

import numpy as np

num_nodes = edge_index.max().item() + 1

# Initialize
out_degree = torch.zeros(num_nodes)
in_degree = torch.zeros(num_nodes)
fraud_degree = torch.zeros(num_nodes)
amount_sum = torch.zeros(num_nodes)
tx_count = torch.zeros(num_nodes)

# Only use train edges (not validation injected)
for i in range(train_edge_cutoff):

    src = edge_index[0, i]
    dst = edge_index[1, i]
    label = edge_labels[i]

    out_degree[src] += 1
    in_degree[dst] += 1

    tx_count[src] += 1

    if label == 1:
        fraud_degree[src] += 1

# Fraud ratio
fraud_ratio = fraud_degree / (out_degree + 1e-6)

# Normalize degrees (log scale)
out_degree = torch.log1p(out_degree)
in_degree = torch.log1p(in_degree)

# Stack node features
x = torch.stack([
    out_degree,
    in_degree,
    fraud_ratio
], dim=1)
```

```
print("Node feature matrix shape:", x.shape)
```

```
Node feature matrix shape: torch.Size([8640302, 3])
```

```
# STEP 2 — Redefine Feature-Enhanced Model

import torch.nn as nn
from torch_geometric.nn import SAGEConv

class EdgeGNN(nn.Module):
    def __init__(self, in_dim=3, hidden_dim=64):
        super().__init__()

        self.conv1 = SAGEConv(in_dim, hidden_dim)
        self.conv2 = SAGEConv(hidden_dim, hidden_dim)

        self.edge_mlp = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, x, edge_index, edge_label_index):

        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)

        src, dst = edge_label_index
        edge_emb = torch.cat([x[src], x[dst]], dim=1)

        return self.edge_mlp(edge_emb).squeeze()

# STEP 3 — Reinitialize Model Cleanly

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = EdgeGNN().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Compute pos_weight only on train edges
pos_weight = (
    (len(train_edge_idx) - edge_labels[train_edge_idx].sum())
    / edge_labels[train_edge_idx].sum()
)

pos_weight = pos_weight.to(device)

criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
```

```python
print("Model reset with node features.")
print("Pos weight:", pos_weight.item())
```

```
Model reset with node features.
Pos weight: 1033.312744140625
```

```python
# STEP 4 — Modify Training Loop (Use Real x)

model.train()

num_batches = 200
batch_size = 2048

for i in range(num_batches):

    start = i * batch_size
    end = start + batch_size

    batch_ids = train_edge_idx[start:end]

    nodes, sub_edge_index, edge_label_index, batch_labels = \
        sample_subgraph_with_labels(batch_ids)

    sub_edge_index = sub_edge_index.to(device)
    edge_label_index = edge_label_index.to(device)
    batch_labels = batch_labels.to(device)

    x_sub = x[nodes].to(device)

    optimizer.zero_grad()

    logits = model(x_sub, sub_edge_index, edge_label_index)
    loss = criterion(logits, batch_labels)

    loss.backward()
    optimizer.step()

    if i % 20 == 0:
        print(f"Batch {i}, Loss: {loss.item():.4f}")
```

```
Batch 0, Loss: 6.1225
Batch 20, Loss: 0.8600
Batch 40, Loss: 0.4034
Batch 60, Loss: 0.1951
Batch 80, Loss: 0.3663
Batch 100, Loss: 0.2447
Batch 120, Loss: 0.1006
Batch 140, Loss: 0.4878
Batch 160, Loss: 0.3707
Batch 180, Loss: 0.2172
```

```
# STEP 5 — Balanced Evaluation (Same as Before)

x_sub = x[nodes].to(device)

# Now Run Balanced Evaluation (Final Test)

from sklearn.metrics import average_precision_score, roc_auc_score

model.eval()

all_preds = []
all_labels = []

with torch.no_grad():

    batch_ids = balanced_val_edge_idx

    nodes, sub_edge_index, edge_label_index, batch_labels = \
        sample_subgraph_with_labels(batch_ids)

    sub_edge_index = sub_edge_index.to(device)
    edge_label_index = edge_label_index.to(device)
    batch_labels = batch_labels.to(device)

    x_sub = x[nodes].to(device)

    logits = model(x_sub, sub_edge_index, edge_label_index)
    probs = torch.sigmoid(logits)

    all_preds.extend(probs.cpu().numpy())
    all_labels.extend(batch_labels.cpu().numpy())

pr_auc = average_precision_score(all_labels, all_preds)
roc_auc = roc_auc_score(all_labels, all_preds)

print("Balanced Validation PR-AUC:", pr_auc)
print("Balanced Validation ROC-AUC:", roc_auc)
```

```
Balanced Validation PR-AUC: 0.998820110268719
Balanced Validation ROC-AUC: 0.9989
```