```
# Import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, f1_score
```

```
# Load the dataset
file_path = '/content/PS_20174392719_1491204439457_log.csv'  # Update this path if needed
data = pd.read_csv(file_path)

# Display the first 5 rows
data.head()
```

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlagg
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	M1979787155	0.0	0.0	0.0	
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	M2044282225	0.0	0.0	0.0	
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	C553264065	0.0	0.0	1.0	
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	C38997010	21182.0	0.0	1.0	
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	M1230701703	0.0	0.0	0.0	

```
# Check for missing values
data.isnull().sum()
```

	0
step	0
type	0
amount	0
nameOrig	0
oldbalanceOrg	0
newbalanceOrig	0
nameDest	0
oldbalanceDest	0
newbalanceDest	1
isFraud	1
isFlaggedFraud	1

dtype: int64

```
# Drop rows with missing values
data = data.dropna()

# Verify that there are no missing values left
data.isnull().sum()
```

	0
step	0
type	0
amount	0
nameOrig	0
oldbalanceOrg	0
newbalanceOrig	0
nameDest	0
oldbalanceDest	0
newbalanceDest	0
isFraud	0
isFlaggedFraud	0

dtype: int64

```
# Check the distribution of the target variable (isFraud)
fraud_counts = data['isFraud'].value_counts()
print(fraud_counts)

# Plot the distribution
sns.countplot(x='isFraud', data=data)
plt.title('Fraud vs Non-Fraud Transactions')
plt.show()
```

```
isFraud
0.0    69750
1.0      107
Name: count, dtype: int64
```


```
# Drop unnecessary columns
data = data.drop(['nameOrig', 'nameDest'], axis=1)

# Display the first 5 rows after dropping columns
data.head()
```

	step	type	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	170136.0	160296.36	0.0	0.0	0.0	0.0
1	1	PAYMENT	1864.28	21249.0	19384.72	0.0	0.0	0.0	0.0
2	1	TRANSFER	181.00	181.0	0.00	0.0	0.0	1.0	0.0
3	1	CASH_OUT	181.00	181.0	0.00	21182.0	0.0	1.0	0.0
4	1	PAYMENT	11668.14	41554.0	29885.86	0.0	0.0	0.0	0.0

```
# One-hot encode the 'type' column
data = pd.get_dummies(data, columns=['type'], drop_first=True)

# Display the first 5 rows after encoding
data.head()
```

	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud	type_CASH_OUT	type_DEBIT	typ
0	1	9839.64	170136.0	160296.36	0.0	0.0	0.0	0.0	False	False	
1	1	1864.28	21249.0	19384.72	0.0	0.0	0.0	0.0	False	False	
2	1	181.00	181.0	0.00	0.0	0.0	1.0	0.0	False	False	
3	1	181.00	181.0	0.00	21182.0	0.0	1.0	0.0	True	False	
4	1	11668.14	41554.0	29885.86	0.0	0.0	0.0	0.0	False	False	

```
# Scale numerical features
scaler = StandardScaler()
data[['amount', 'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest', 'newbalanceDest']] = scaler.fit_transform(
    data[['amount', 'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest', 'newbalanceDest']]
)

# Display the first 5 rows after scaling
data.head()
```

	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud	type_CASH_OUT	type_DEBIT	typ
0	1	-0.465278	-0.262996	-0.268568	-0.355541	-0.412687	0.0	0.0	False	False	
1	1	-0.489346	-0.316330	-0.318362	-0.355541	-0.412687	0.0	0.0	False	False	
2	1	-0.494425	-0.323877	-0.325212	-0.355541	-0.412687	1.0	0.0	False	False	
3	1	-0.494425	-0.323877	-0.325212	-0.346726	-0.412687	1.0	0.0	True	False	
4	1	-0.459760	-0.309056	-0.314651	-0.355541	-0.412687	0.0	0.0	False	False	

```
# Define features (X) and target (y)
X = data.drop('isFraud', axis=1)
y = data['isFraud']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Print the shapes of the resulting datasets
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (55885, 11)
X_test shape: (13972, 11)
y_train shape: (55885,)
y_test shape: (13972,)
```

```
# Initialize the Random Forest Classifier
model = RandomForestClassifier(random_state=42, class_weight='balanced')

# Train the model
model.fit(X_train, y_train)
```

```
RandomForestClassifier
RandomForestClassifier(class_weight='balanced', random_state=42)
```

```
# Make predictions
y_pred = model.predict(X_test)

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)

# Classification Report
print("Classification Report:")
print(classification_report(y_test, y_pred))

# F1-Score
f1 = f1_score(y_test, y_pred)
print(f"F1-Score: {f1}")
```

```
Confusion Matrix:
[[13949     2]
 [   13     8]]
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00     13951
         1.0       0.80      0.38      0.52        21

    accuracy                           1.00     13972
   macro avg       0.90      0.69      0.76     13972
weighted avg       1.00      1.00      1.00     13972

F1-Score: 0.5161290322580645
```

```
# Install imbalanced-learn library (if not already installed)
!pip install imbalanced-learn

# Import SMOTE
from imblearn.over_sampling import SMOTE

# Apply SMOTE to the training data
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

# Check the new class distribution
print("Resampled Class Distribution:")
print(y_train_resampled.value_counts())
```

```
Requirement already satisfied: imbalanced-learn in /usr/local/lib/python3.11/dist-packages (0.13.0)
Requirement already satisfied: numpy<3,>=1.24.3 in /usr/local/lib/python3.11/dist-packages (from imbalanced-learn) (1.26.4)
Requirement already satisfied: scipy<2,>=1.10.1 in /usr/local/lib/python3.11/dist-packages (from imbalanced-learn) (1.13.1)
Requirement already satisfied: scikit-learn<2,>=1.3.2 in /usr/local/lib/python3.11/dist-packages (from imbalanced-learn) (1.6.1)
Requirement already satisfied: sklearn-compat<1,>=0.1 in /usr/local/lib/python3.11/dist-packages (from imbalanced-learn) (0.1.3)
Requirement already satisfied: joblib<2,>=1.1.1 in /usr/local/lib/python3.11/dist-packages (from imbalanced-learn) (1.4.2)
Requirement already satisfied: threadpoolctl<4,>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from imbalanced-learn) (3.5.0)
Resampled Class Distribution:
isFraud
0.0    55799
1.0    55799
Name: count, dtype: int64
```

```
# Initialize the Random Forest Classifier
model_resampled = RandomForestClassifier(random_state=42)

# Train the model on resampled data
model_resampled.fit(X_train_resampled, y_train_resampled)

# Make predictions on the test set
y_pred_resampled = model_resampled.predict(X_test)

# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_resampled))

print("Classification Report:")
print(classification_report(y_test, y_pred_resampled))
```

```
print("F1-Score:", f1_score(y_test, y_pred_resampled))
```

```
Confusion Matrix:
[[13923    28]
 [    5    16]]
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00     13951
         1.0       0.36      0.76      0.49        21

    accuracy                           1.00     13972
   macro avg       0.68      0.88      0.75     13972
weighted avg       1.00      1.00      1.00     13972

F1-Score: 0.49230769230769234
```

```
# Initialize a simpler Random Forest Classifier
model_tuned = RandomForestClassifier(
    random_state=42,
    class_weight='balanced',
    max_depth=10,          # Limit the depth of the trees
    min_samples_split=10,  # Require more samples to split a node
    n_estimators=100       # Use fewer trees
)

# Train the model
model_tuned.fit(X_train, y_train)

# Make predictions
y_pred_tuned = model_tuned.predict(X_test)

# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_tuned))

print("Classification Report:")
print(classification_report(y_test, y_pred_tuned))

print("F1-Score:", f1_score(y_test, y_pred_tuned))
```

```
Confusion Matrix:
[[13946     5]
 [    6    15]]
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00     13951
         1.0       0.75      0.71      0.73        21

    accuracy                           1.00     13972
   macro avg       0.87      0.86      0.87     13972
weighted avg       1.00      1.00      1.00     13972

F1-Score: 0.7317073170731707
```

```
# Install XGBoost (if not already installed)
!pip install xgboost

# Import XGBoost
from xgboost import XGBClassifier

# Initialize XGBoost with scale_pos_weight to handle class imbalance
model_xgb = XGBClassifier(
    random_state=42,
    scale_pos_weight=len(y_train[y_train == 0]) / len(y_train[y_train == 1]),  # Adjust for class imbalance
    max_depth=5,          # Limit the depth of the trees
    learning_rate=0.1,    # Reduce the learning rate
    n_estimators=100      # Use fewer trees
)

# Train the model
model_xgb.fit(X_train, y_train)

# Make predictions
y_pred_xgb = model_xgb.predict(X_test)
```

```
# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_xgb))

print("Classification Report:")
print(classification_report(y_test, y_pred_xgb))

print("F1-Score:", f1_score(y_test, y_pred_xgb))
```

```
Requirement already satisfied: xgboost in /usr/local/lib/python3.11/dist-packages (2.1.4)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from xgboost) (1.26.4)
Requirement already satisfied: nvidia-nccl-cu12 in /usr/local/lib/python3.11/dist-packages (from xgboost) (2.21.5)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from xgboost) (1.13.1)
Confusion Matrix:
[[13934    17]
 [    3    18]]
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00     13951
         1.0       0.51      0.86      0.64        21

    accuracy                           1.00     13972
   macro avg       0.76      0.93      0.82     13972
weighted avg       1.00      1.00      1.00     13972

F1-Score: 0.6428571428571429
```

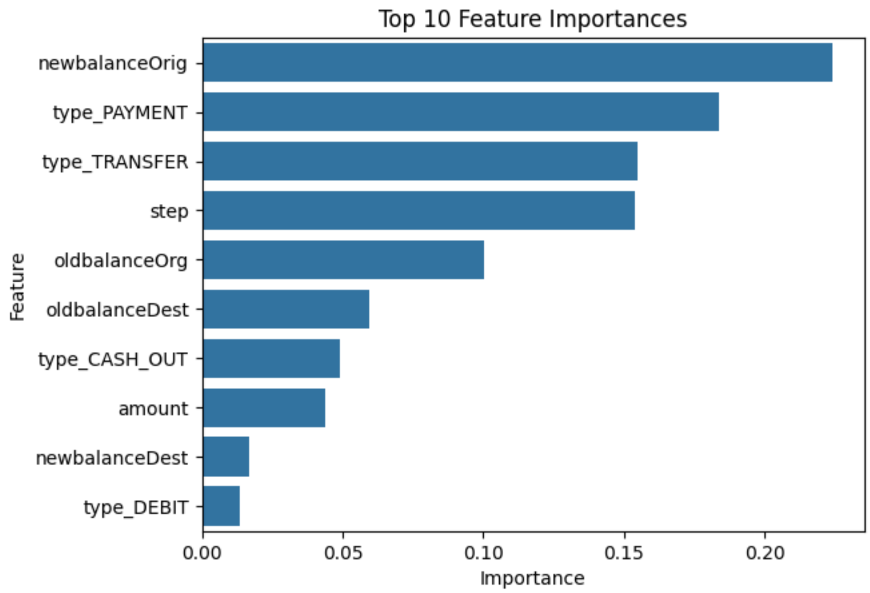
```
# Get feature importances from the XGBoost model
feature_importances = model_xgb.feature_importances_

# Create a DataFrame to display feature importances
feature_importance_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

# Display the top 10 features
print(feature_importance_df.head(10))

# Plot feature importances
sns.barplot(x='Importance', y='Feature', data=feature_importance_df.head(10))
plt.title('Top 10 Feature Importances')
plt.show()
```

```
             Feature  Importance
3    newbalanceOrig    0.224126
9      type_PAYMENT    0.183793
10    type_TRANSFER    0.154898
0              step    0.154052
2     oldbalanceOrg    0.100325
4    oldbalanceDest    0.059661
7     type_CASH_OUT    0.049044
1            amount    0.043929
5    newbalanceDest    0.016578
8        type_DEBIT    0.013593
```




```python
# Select the top 8 features (you can adjust this number)
top_features = feature_importance_df['Feature'].head(8).tolist()

# Filter the training and testing data to include only the top features
X_train_top = X_train[top_features]
X_test_top = X_test[top_features]

# Retrain the XGBoost model on the top features
model_xgb_top = XGBClassifier(
    random_state=42,
    scale_pos_weight=len(y_train[y_train == 0]) / len(y_train[y_train == 1]),
    max_depth=5,
    learning_rate=0.1,
    n_estimators=100
)

# Train the model
model_xgb_top.fit(X_train_top, y_train)

# Make predictions
y_pred_xgb_top = model_xgb_top.predict(X_test_top)

# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_xgb_top))

print("Classification Report:")
print(classification_report(y_test, y_pred_xgb_top))

print("F1-Score:", f1_score(y_test, y_pred_xgb_top))
```

```
Confusion Matrix:
[[13924    27]
 [    3    18]]
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00     13951
         1.0       0.40      0.86      0.55        21

    accuracy                           1.00     13972
   macro avg       0.70      0.93      0.77     13972
```

```
weighted avg       1.00      1.00      1.00     13972

F1-Score: 0.5454545454545454
```

```
# Get predicted probabilities for the fraudulent class
y_pred_proba = model_xgb.predict_proba(X_test)[:, 1]

# Adjust the decision threshold to 0.8 (you can experiment with other values)
threshold = 0.8
y_pred_adjusted = (y_pred_proba >= threshold).astype(int)

# Evaluate the model with the adjusted threshold
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_adjusted))

print("Classification Report:")
print(classification_report(y_test, y_pred_adjusted))

print("F1-Score:", f1_score(y_test, y_pred_adjusted))
```

```
Confusion Matrix:
[[13941    10]
 [    6    15]]
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00     13951
         1.0       0.60      0.71      0.65        21

    accuracy                           1.00     13972
   macro avg       0.80      0.86      0.83     13972
weighted avg       1.00      1.00      1.00     13972

F1-Score: 0.6521739130434783
```

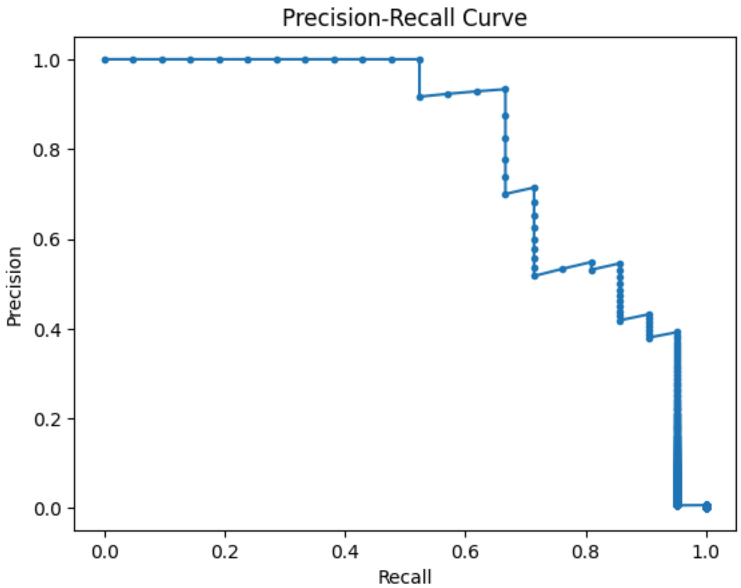
```
from sklearn.metrics import precision_recall_curve

# Compute precision-recall curve
precision, recall, thresholds = precision_recall_curve(y_test, y_pred_proba)

# Plot the precision-recall curve
plt.plot(recall, precision, marker='.')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.show()

# Find the threshold that maximizes the F1-score
f1_scores = 2 * (precision * recall) / (precision + recall)
optimal_idx = np.argmax(f1_scores)
optimal_threshold = thresholds[optimal_idx]

print("Optimal Threshold:", optimal_threshold)
print("Optimal F1-Score:", f1_scores[optimal_idx])
```




```
Optimal Threshold: 0.963593
Optimal F1-Score: 0.7777777777777778
```

```
# Use the optimal threshold to make predictions
y_pred_optimal = (y_pred_proba >= optimal_threshold).astype(int)

# Evaluate the model with the optimal threshold
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_optimal))

print("Classification Report:")
print(classification_report(y_test, y_pred_optimal))

print("F1-Score:", f1_score(y_test, y_pred_optimal))
```

```
Confusion Matrix:
[[13950     1]
 [    7    14]]
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00     13951
         1.0       0.93      0.67      0.78        21

    accuracy                           1.00     13972
   macro avg       0.97      0.83      0.89     13972
weighted avg       1.00      1.00      1.00     13972

F1-Score: 0.7777777777777778
```

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix, f1_score
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
import lightgbm as lgb
from sklearn.neural_network import MLPClassifier
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import mlflow
import optuna
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import joblib
import logging
import warnings
import json
from typing import Dict, List, Tuple
import redis
```

```
from kafka import KafkaConsumer, KafkaProducer

# Set up logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class TransactionDataset(Dataset):
    def __init__(self, features, labels):
        self.features = torch.FloatTensor(features)
        self.labels = torch.FloatTensor(labels)

    def __len__(self):
        return len(self.features)

    def __getitem__(self, idx):
        return self.features[idx], self.labels[idx]

class DeepFraudDetector(nn.Module):
    def __init__(self, input_dim):
        super(DeepFraudDetector, self).__init__()
        self.layer1 = nn.Linear(input_dim, 256)
        self.layer2 = nn.Linear(256, 128)
        self.layer3 = nn.Linear(128, 64)
        self.layer4 = nn.Linear(64, 1)

        self.batch_norm1 = nn.BatchNorm1d(256)
        self.batch_norm2 = nn.BatchNorm1d(128)
        self.batch_norm3 = nn.BatchNorm1d(64)

        self.dropout = nn.Dropout(0.3)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.batch_norm1(self.layer1(x)))
        x = self.dropout(x)
        x = self.relu(self.batch_norm2(self.layer2(x)))
        x = self.dropout(x)
        x = self.relu(self.batch_norm3(self.layer3(x)))
        x = self.dropout(x)
        x = self.sigmoid(self.layer4(x))
        return x

class FraudDetectionSystem:
    def __init__(self):
        self.scaler = StandardScaler()
        self.label_encoders = {}
        self.models = {}
        self.best_model = None
        self.feature_columns = None

    def preprocess_data(self, data: pd.DataFrame) -> Tuple[pd.DataFrame, pd.Series]:
        """Preprocess the PaySim dataset"""
        logger.info("Starting data preprocessing...")

        # Drop unnecessary columns
        data = data.drop(['nameOrig', 'nameDest', 'isFlaggedFraud'], axis=1)

        # Encode categorical variables
        categorical_columns = ['type']
        for col in categorical_columns:
            if col not in self.label_encoders:
                self.label_encoders[col] = LabelEncoder()
            data[col] = self.label_encoders[col].fit_transform(data[col])

        # Create new features
        data['amount_per_oldbalance'] = data['amount'] / (data['oldbalanceOrg'] + 1)
        data['amount_per_newbalance'] = data['amount'] / (data['newbalanceOrig'] + 1)
        data['balance_difference'] = data['newbalanceOrig'] - data['oldbalanceOrg']

        # Extract labels
        labels = data['isFraud']
        features = data.drop('isFraud', axis=1)

        # Store feature columns for inference
        self.feature_columns = features.columns.tolist()
```

```
        # Scale features
        scaled_features = self.scaler.fit_transform(features)

        return pd.DataFrame(scaled_features, columns=features.columns), labels

    def train_models(self, X: pd.DataFrame, y: pd.Series):
        """Train multiple models and select the best one"""
        logger.info("Starting model training...")

        # Split data
        X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

        # Initialize models
        self.models = {
            'random_forest': RandomForestClassifier(n_estimators=100, random_state=42),
            'xgboost': xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss'),
            'lightgbm': lgb.LGBMClassifier(),
            'neural_network': MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=300)
        }

        # Train and evaluate models
        best_f1 = 0
        for name, model in self.models.items():
            logger.info(f"Training {name}...")
            model.fit(X_train, y_train)
            y_pred = model.predict(X_val)
            f1 = f1_score(y_val, y_pred)
            logger.info(f"{name} F1 Score: {f1}")

            if f1 > best_f1:
                best_f1 = f1
                self.best_model = model

        # Train Deep Learning model
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        deep_model = DeepFraudDetector(input_dim=X.shape[1]).to(device)

        # Create data loaders
        train_dataset = TransactionDataset(X_train.values, y_train.values)
        train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

        # Train deep model
        criterion = nn.BCELoss()
        optimizer = torch.optim.Adam(deep_model.parameters(), lr=0.001)

        for epoch in range(10):
            deep_model.train()
            for batch_X, batch_y in train_loader:
                batch_X, batch_y = batch_X.to(device), batch_y.to(device)
                optimizer.zero_grad()
                outputs = deep_model(batch_X)
                loss = criterion(outputs, batch_y.unsqueeze(1))
                loss.backward()
                optimizer.step()

        self.models['deep_learning'] = deep_model

    def optimize_hyperparameters(self, X: pd.DataFrame, y: pd.Series):
        """Optimize hyperparameters using Optuna"""
        def objective(trial):
            params = {
                'n_estimators': trial.suggest_int('n_estimators', 50, 300),
                'max_depth': trial.suggest_int('max_depth', 3, 10),
                'min_samples_split': trial.suggest_int('min_samples_split', 2, 10),
                'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 4)
            }

            model = RandomForestClassifier(**params)
            score = cross_val_score(model, X, y, cv=5, scoring='f1').mean()
            return score

        study = optuna.create_study(direction='maximize')
        study.optimize(objective, n_trials=50)

        return study.best_params

    def save_model(self, path: str):
```

```
        """Save the trained model and preprocessing objects"""
        model_artifacts = {
            'model': self.best_model,
            'scaler': self.scaler,
            'label_encoders': self.label_encoders,
            'feature_columns': self.feature_columns
        }
        joblib.dump(model_artifacts, path)

    def load_model(self, path: str):
        """Load the trained model and preprocessing objects"""
        model_artifacts = joblib.load(path)
        self.best_model = model_artifacts['model']
        self.scaler = model_artifacts['scaler']
        self.label_encoders = model_artifacts['label_encoders']
        self.feature_columns = model_artifacts['feature_columns']

class RealTimeInference:
    def __init__(self, model_path: str):
        self.fraud_detection = FraudDetectionSystem()
        self.fraud_detection.load_model(model_path)
        self.redis_client = redis.Redis(host='localhost', port=6379, db=0)

    def preprocess_transaction(self, transaction: Dict) -> pd.DataFrame:
        """Preprocess a single transaction for inference"""
        df = pd.DataFrame([transaction])

        # Apply the same preprocessing steps
        for col, le in self.fraud_detection.label_encoders.items():
            if col in df.columns:
                df[col] = le.transform(df[col])

        # Create the same features as in training
        df['amount_per_oldbalance'] = df['amount'] / (df['oldbalanceOrg'] + 1)
        df['amount_per_newbalance'] = df['amount'] / (df['newbalanceOrig'] + 1)
        df['balance_difference'] = df['newbalanceOrig'] - df['oldbalanceOrg']

        # Scale features
        scaled_features = self.fraud_detection.scaler.transform(df)
        return pd.DataFrame(scaled_features, columns=self.fraud_detection.feature_columns)

    def predict(self, transaction: Dict) -> Dict:
        """Make real-time predictions"""
        # Check cache first
        cache_key = f"prediction:{transaction['transactionId']}"
        cached_prediction = self.redis_client.get(cache_key)

        if cached_prediction:
            return json.loads(cached_prediction)

        # Preprocess transaction
        processed_transaction = self.preprocess_transaction(transaction)

        # Make prediction
        probability = self.fraud_detection.best_model.predict_proba(processed_transaction)[0][1

        result = {
            'transaction_id': transaction['transactionId'],
            'fraud_probability': float(probability),
            'is_fraud': probability > 0.5,
            'confidence': float(probability) if probability > 0.5 else float(1 - probability)
        }

        # Cache the result
        self.redis_client.setex(cache_key, 3600, json.dumps(result))

        return result

# FastAPI app for serving predictions
app = FastAPI()

class Transaction(BaseModel):
    transactionId: str
    type: str
    amount: float
    oldbalanceOrg: float
    newbalanceOrig: float
```

```
        oldbalanceDest: float
        newbalanceDest: float

    @app.post("/predict")
    async def predict_fraud(transaction: Transaction):
        try:
            inference_service = RealTimeInference("model.joblib")
            prediction = inference_service.predict(transaction.dict())
            return prediction
        except Exception as e:
            logger.error(f"Error making prediction: {str(e)}")
            raise HTTPException(status_code=500, detail="Prediction failed")

    # Main execution
    if __name__ == "__main__":
        # Load and preprocess data
        data = pd.read_csv("/content/PS_20174392719_1491204439457_log.csv")
```