

ДОДАТОК А

Лістинг коду проєкту

А.1 Структура проєкту

```
MultimodalCryptoSystem/
├─ AIService/                                # Python мікросервіс штучного інтелекту
│   ├── main.py                             # FastAPI сервер, логіка Stacking-ансамблю та інференс
│   ├── requirements.txt                     # Залежності (TensorFlow, XGBoost, FastAPI)
│   └─ Dockerfile                           # Контейнеризація AI-сервісу
├─ WebUI/                                    # C# .NET 8 Backend та Інтерфейс
│   ├── Dockerfile                          # Контейнеризація Web-сервісу
│   └─ CryptoWebApp/
│       ├── Program.cs                      # Конфігурація DI, налаштування БД та пайплайну
│       ├── appsettings.json                # Конфігурація середовища
│       ├── Controllers/                   # Контролери MVC
│       │   ├── AdminController.cs
│       │   ├── HomeController.cs
│       │   └─ ...
│       ├── Data/
│       │   └─ ApplicationDbContext.cs      # Контекст бази даних (Entity Framework)
│       ├── Models/                        # Структури даних БД
│       │   ├── MarketData.cs              # Ринкові котирування (Gate.io)
│       │   ├── OnChainData.cs             # Активність китів (DeBank)
│       │   ├── TextSentimentData.cs       # Сентимент-аналіз новин (FinBERT)
│       │   ├── PredictionHistory.cs        # Збереження прогнозів нейромережі
│       │   ├── DelistNewsModel.cs         # Моделі парсера делістингів
│       │   └─ ... (інші моделі бізнес-логіки)
│       ├── Services/                      # Ядро бізнес-логіки (Парсери та взаємодія)
│       │   ├── AiPredictionService.cs     # Комунікація з Python AI-мікросервісом
│       │   ├── GateIoParserService.cs     # Агрегація біржових даних
│       │   ├── OnChainParserService.cs    # Агрегація блокчейн-даних
│       │   ├── SentimentParserService.cs  # Агрегація та скоринг новин
│       │   ├── InvestmentDecisionService.cs # Розрахунок торгових сигналів та ризиків
│       │   ├── DelistPollingService.cs    # Telegram-бот для моніторингу делістингів
│       │   └─ ... (інші фонові сервіси)
│       ├── Areas/Admin/                   # Адміністративна панель керування
│       └─ Views/                          # Razor-сторінки клієнтського інтерфейсу
└─ docker-compose.yml                      # Оркестрація мікросервісів (PostgreSQL, .NET, Python,
Caddy)
└─ bitbucket-pipelines.yml                 # CI/CD пайплайн для автоматичного деплою на Hetzner
```

A.2 Тренування моделі в Google Colab

```

import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

# 1. Load our custom multimodal dataset
file_path = 'multimodal_crypto_dataset (7).csv'
print(f"Читання даних з файлу {file_path}...")

# FIXED: replaced 'Date' with 'Timestamp' because that is how the column is
named in your CSV
df = pd.read_csv(file_path, parse_dates=['Timestamp'],
index_col='Timestamp')

# Fill possible missing values with previous values (Forward Fill)
df = df.fillna(method='ffill')

print(f"Дані успішно завантажено! Загальна кількість днів: {len(df)}")
print("-" * 50)

# 2. Extract multimodal features
# According to the thesis topic, we use: market data (OHLCV) + text data
(Sentiment) + on-chain data (ActiveAddresses)
features = ['Open', 'High', 'Low', 'Close', 'Volume', 'SentimentScore',
'ActiveAddresses']
data = df[features].values

# 3. Normalize the data (MinMaxScaler)
# Bring all values (from price to sentiment) into the [2] range for stable
neural network training
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)

# Find the index of the target variable 'Close' to build the y vector
target_idx = features.index('Close')
```

```

# 4. Build the sliding window

# Use a 30-day lookback to predict the price for day 31
SEQ_LENGTH = 30

def create_sequences(data, seq_length, target_index):
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:(i + seq_length)])      # 30 days by 7 features
        y.append(data[i + seq_length, target_index]) # Next-day 'Close'
    prediction
    return np.array(X), np.array(y)

X, y = create_sequences(scaled_data, SEQ_LENGTH, target_idx)

print("=== РЕЗУЛЬТАТИ ПРЕПРОЦЕСИНГУ ===")

print(f"Розмірність вхідного тензора X: {X.shape} -> (Кількість_прикладів, Розмір_вікна, Кількість_ознак)")

print(f"Розмірність вектора цільової змінної y: {y.shape} -> (Кількість_прикладів,)")

# 5. Split into training and test sets (80% / 20%)
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

print(f"Тренувальна вибірка: X={X_train.shape}, y={y_train.shape}")
print(f"Тестова вибірка: X={X_test.shape}, y={y_test.shape}")

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv1D, MaxPooling1D, Bidirectional, LSTM, Dense, Dropout, Attention, GlobalAveragePooling1D
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt

print("Ініціалізація архітектури CNN-BiLSTM-Attention...")

# 1. Input layer (shape: 30 days, 7 features)
# FIXED: use indexes 1 and 2

```

```

inputs = Input(shape=(X_train.shape[1], X_train.shape[2]))

# 2. Convolutional block (CNN) - extracting local patterns (price jumps,
volumes)

x = Conv1D(filters=64, kernel_size=3, activation='relu',
padding='same')(inputs)
x = MaxPooling1D(pool_size=2)(x)

# 3. Recurrent block (BiLSTM) - learning temporal dependencies in both
directions

# return_sequences=True is required to pass the full sequence into the
Attention layer

lstm_out = Bidirectional(LSTM(64, return_sequences=True))(x)

# 4. Attention mechanism - focusing on important days

# Use lstm_out both as the query and as the value
attention_out = Attention()([lstm_out, lstm_out])

# 5. Pooling (Global Average Pooling) and fully connected layers (Dense)
x = GlobalAveragePooling1D()(attention_out)
x = Dropout(0.3)(x) # Protection against overfitting
x = Dense(32, activation='relu')(x)

# 6. Output layer (predicting 1 value - the next day's closing price)
outputs = Dense(1, activation='linear')(x)

# Build and compile the model

model = Model(inputs=inputs, outputs=outputs)

model.compile(optimizer=Adam(learning_rate=0.001), loss='mse',
metrics=['mae'])

model.summary()

# 7. Train the model

print("\nПочаток процесу навчання моделі (Експериментальне
моделювання)...\n")

```

```

history = model.fit(
    X_train, y_train,
    epochs=50,          # Number of training epochs
    batch_size=32,      # Batch size
    validation_data=(X_test, y_test), # Validation data
    verbose=1
)

# 8. Visualize training results (Loss/MSE)
plt.figure(figsize=(10, 6))

plt.plot(history.history['loss'], label='Помилка на тренуванні (Train Loss)')

plt.plot(history.history['val_loss'], label='Помилка на валідації (Val Loss)')

plt.title('Графік процесу навчання CNN-BiLSTM-Attention')
plt.xlabel('Епохи')
plt.ylabel('Середньоквадратична помилка (MSE)')
plt.legend()
plt.grid(True)
plt.show()

from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np
import matplotlib.pyplot as plt

print("Генерація прогнозу на тестовій вибірці...")

# 1. Get predictions from the model
predictions = model.predict(X_test)

# 2. Inverse normalization (Inverse Transform)
# Use the correct index 2 (which corresponds to 7 features)
num_features = X_train.shape[2]

# For predictions
dummy_pred = np.zeros((len(predictions), num_features))
dummy_pred[:, target_idx] = predictions.flatten()
pred_prices = scaler.inverse_transform(dummy_pred)[:, target_idx]

```

```

# For real values (y_test)
dummy_test = np.zeros((len(y_test), num_features))
dummy_test[:, target_idx] = y_test.flatten()
real_prices = scaler.inverse_transform(dummy_test[:, target_idx])

# 3. Calculate the final business quality metrics of the model
rmse = np.sqrt(mean_squared_error(real_prices, pred_prices))
mae = mean_absolute_error(real_prices, pred_prices)
mape = np.mean(np.abs((real_prices - pred_prices) / real_prices)) * 100

print(f"\n=== ФІНАЛЬНІ МЕТРИКИ МОДЕЛІ (для Розділу 4 диплому) ===")
print(f"RMSE (Коренева середньоквадратична помилка): ${rmse:.2f}")
print(f"MAE (Середня абсолютна помилка): ${mae:.2f}")
print(f"MAPE (Середня абсолютна помилка у відсотках): {mape:.2f}%")

# 4. Visualization: actual price vs predicted price
plt.figure(figsize=(14, 7))

plt.plot(real_prices, label='Реальна ціна BTC ($)', color='blue',
linewidth=2)

plt.plot(pred_prices, label='Прогноз моделі ($)', color='red',
linestyle='dashed', linewidth=2)

plt.title('Порівняння реальних та прогнозованих цін (CNN-BiLSTM-Attention)')

plt.xlabel('Дні (Тестова вибірка)')
plt.ylabel('Ціна закриття (USD)')
plt.legend()

plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# 5. Save the model for our C# microservice
model.save('multimodal_crypto_model.keras')

print("\nМодель успішно збережена у файл
'multimodal_crypto_model.keras'!")

import pandas as pd
import numpy as np

```

```

import xgboost as xgb

import joblib

from sklearn.model_selection import train_test_split

# 1. Load your multimodal data
print("Завантаження даних...")
df = pd.read_csv('multimodal_crypto_dataset (7).csv')

# 2. Calculate the target variable: next-day percentage price change
(Returns)

# This will force the model to look specifically for momentum and trend
reversals!
df['Target_Return'] = df['Close'].pct_change().shift(-1)
df = df.dropna()

# 3. Prepare the feature space (using your 7 modalities)
features = ['Open', 'High', 'Low', 'Close', 'Volume', 'SentimentScore',
'ActiveAddresses']
X = df[features].values
y = df['Target_Return'].values

# 4. Split into training and test sets (preserving chronology)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15,
shuffle=False)

# 5. Train XGBoost (Model 2)
print("Тренування ансамблевої моделі XGBoost...")
xgb_model = xgb.XGBRegressor(
    n_estimators=150,
    learning_rate=0.05,
    max_depth=5,
    random_state=42,
    objective='reg:squarederror'
)
xgb_model.fit(X_train, y_train)

```

```
# 6. Save the trained model

joblib.dump(xgb_model, 'xgboost_model.pkl')

print("УСПІХ! Модель збережена у файл 'xgboost_model.pkl'. Завантажте його
на ваш комп'ютер!")
```

A.3 main.py (FastAPI, логіка Stacking-ансамблю та інференс)

```
from datetime import datetime, timezone

import json

import os

import tempfile

from threading import Event, Lock

from uuid import uuid4

import zipfile

import joblib


from fastapi import FastAPI, BackgroundTasks, HTTPException
from fastapi.responses import JSONResponse
from pydantic import BaseModel
from typing import Dict, List, Optional

import numpy as np

import tensorflow as tf

from sklearn.preprocessing import MinMaxScaler


# FastAPI initialization
app = FastAPI(
    title="Multimodal Crypto Prediction API",
    description="Adaptive AI microservice for master's thesis (CSAIm-24-1)",
    version="2.0.0"
)


SERVICE_DIR = os.path.dirname(os.path.abspath(__file__))
MODEL_DIR = os.getenv("MODEL_DIR", os.path.join(SERVICE_DIR, "model_data"))
PRIMARY_MODEL_PATH = os.path.join(MODEL_DIR,
    "multimodal_crypto_model.keras")
XGB_MODEL_PATH = os.path.join(MODEL_DIR, "xgboost_model.pkl")
MODEL_CANDIDATES = (
```

```

PRIMARY_MODEL_PATH,
os.path.join(SERVICE_DIR, "multimodal_crypto_model.keras"),
"multimodal_crypto_model.keras",
)
xgb_model = None

try:
    if os.path.exists(XGB_MODEL_PATH):
        xgb_model = joblib.load(XGB_MODEL_PATH)
        print("XGBoost model loaded for ensemble inference.")
except Exception as e:
    print(f"Failed to load XGBoost model: {e}")

SCALER_METADATA_FILE = "scaler_metadata.json"
SEQ_LENGTH = 30
TARGET_IDX = 3 # Index of the 'Close' column (Open=0, High=1, Low=2,
Close=3, Volume=4, Sentiment=5, ActiveAddresses=6)
TRAINING_STATUS_LOCK = Lock()
TRAINING_STOP_EVENT = Event()
TRAINING_STATUS: Dict[str, Optional[str]] = {
    "status": "idle",
    "state": "Idle",
    "job_id": None,
    "message": "Донавчання ще не запускалось.",
    "updated_at_utc": None,
    "started_at_utc": None,
    "finished_at_utc": None,
}

# Data structure for prediction
class PredictionRequest(BaseModel):
    data: List[List[float]] # Array shaped (30 days, 7 features)

class BatchPredictionRequest(BaseModel):
    sequences: List[List[List[float]]] # Array of multiple windows shaped
(30 days, 7 features)

```

```
# Data structure for retraining (we pass fresh data, for example from the
last 60-100 days)
```

```
class RetrainRequest(BaseModel):
```

```
    data: List[List[float]]
```

```
def get_model_path() -> str:
```

```
    for candidate in MODEL_CANDIDATES:
```

```
        if os.path.exists(candidate):
```

```
            return candidate
```

```
    return PRIMARY_MODEL_PATH
```

```
def ensure_model_dir() -> None:
```

```
    os.makedirs(MODEL_DIR, exist_ok=True)
```

```
def get_scaler_metadata_path(model_path: Optional[str] = None) -> str:
```

```
    resolved_model_path = model_path or get_model_path()
```

```
    model_directory = os.path.dirname(resolved_model_path) or "."
```

```
    return os.path.join(model_directory, SCALER_METADATA_FILE)
```

```
def strip_unsupported_keras_fields(value):
```

```
    if isinstance(value, dict):
```

```
        cleaned = {}
```

```
        for key, item in value.items():
```

```
            if key == "quantization_config":
```

```
                continue
```

```
            cleaned[key] = strip_unsupported_keras_fields(item)
```

```
        return cleaned
```

```
    if isinstance(value, list):
```

```
        return [strip_unsupported_keras_fields(item) for item in value]
```

```
    return value
```

```

def load_model_with_compat(model_path: str):
    try:
        return tf.keras.models.load_model(model_path, compile=False)
    except Exception as exc:
        error_text = str(exc)

        needs_compat_patch = model_path.endswith(".keras") and
"quantization_config" in error_text

        if not needs_compat_patch:
            raise

    with tempfile.TemporaryDirectory() as temp_dir:
        patched_model_path = os.path.join(temp_dir, "patched_model.keras")

        with zipfile.ZipFile(model_path, "r") as source_archive:
            with zipfile.ZipFile(patched_model_path, "w",
compression=zipfile.ZIP_DEFLATED) as target_archive:
                for archive_info in source_archive.infolist():
                    file_bytes = source_archive.read(archive_info.filename)
                    if archive_info.filename == "config.json":
                        config = json.loads(file_bytes.decode("utf-8"))
                        file_bytes =
json.dumps(strip_unsupported_keras_fields(config)).encode("utf-8")

                        target_archive.writestr(archive_info, file_bytes)

        return tf.keras.models.load_model(patched_model_path,
compile=False)

def load_scaler_metadata(feature_count: int) -> Optional[Dict[str,
np.ndarray]]:
    metadata_path = get_scaler_metadata_path()
    if not os.path.exists(metadata_path):
        return None

    try:
        with open(metadata_path, "r", encoding="utf-8") as metadata_file:
            metadata = json.load(metadata_file)

```

```

        feature_mins = np.asarray(metadata.get("feature_mins", []),
dtype=np.float32)

        feature_maxs = np.asarray(metadata.get("feature_maxs", []),
dtype=np.float32)

        if feature_mins.size != feature_count or feature_maxs.size !=
feature_count:

            return None

        return {

            "feature_mins": feature_mins,

            "feature_maxs": feature_maxs,

        }

    except (OSError, ValueError, TypeError, json.JSONDecodeError):

        return None


def save_scaler_metadata(scaler: MinMaxScaler, model_path: str) -> None:

    metadata_path = get_scaler_metadata_path(model_path)

    metadata = {

        "feature_mins": scaler.data_min_.astype(float).tolist(),

        "feature_maxs": scaler.data_max_.astype(float).tolist(),

        "target_idx": TARGET_IDX,

        "updated_at_utc": utc_now_iso(),

    }

    with open(metadata_path, "w", encoding="utf-8") as metadata_file:

        json.dump(metadata, metadata_file)


def scale_sequence(input_data: np.ndarray, feature_mins: np.ndarray,
feature_maxs: np.ndarray) -> np.ndarray:

    feature_ranges = feature_maxs - feature_mins

    scaled = np.divide(

        input_data - feature_mins,

        feature_ranges,

        out=np.zeros_like(input_data, dtype=np.float32),

```

```

        where=feature_ranges != 0,
    )
    return scaled.astype(np.float32)

def inverse_scale_close_values(prediction_scaled, feature_mins: np.ndarray,
                               feature_maxs: np.ndarray):
    close_min = float(feature_mins[TARGET_IDX])
    close_max = float(feature_maxs[TARGET_IDX])
    close_range = close_max - close_min

    if close_range == 0:
        return [close_min for _ in np.ravel(prediction_scaled)]

    return [float(close_min + (float(value) * close_range)) for value in
            np.ravel(prediction_scaled)]

def prepare_model_for_training(model):
    model.compile(
        optimizer=tf.keras.optimizers.Adam(),
        loss="mse",
        metrics=["mae"],
    )
    return model

def utc_now_iso() -> str:
    return datetime.now(timezone.utc).isoformat()

def get_training_status_snapshot() -> Dict[str, Optional[str]]:
    with TRAINING_STATUS_LOCK:
        snapshot = dict(TRAINING_STATUS)

    snapshot["model_loaded"] = os.path.exists(get_model_path())
    return snapshot

```

```

def clear_training_stop_request() -> None:
    TRAINING_STOP_EVENT.clear()

def request_training_stop() -> None:
    TRAINING_STOP_EVENT.set()

def is_training_stop_requested() -> bool:
    return TRAINING_STOP_EVENT.is_set()

def prepare_scaled_sequences(sequences: List[List[List[float]]]):
    if not sequences:
        raise HTTPException(status_code=400, detail="Потрібно передати  
щонайменше одну послідовність для прогнозу.")

    validated_sequences = []
    scaled_sequences = []
    feature_count = None

    for index, sequence in enumerate(sequences):
        input_data = np.array(sequence, dtype=np.float32)
        if input_data.ndim != 2:
            raise HTTPException(status_code=400, detail=f"Послідовність  
#{index + 1} повинна бути двовимірним масивом.")

        if input_data.shape[0] != SEQ_LENGTH:
            raise HTTPException(status_code=400, detail=f"Послідовність  
#{index + 1} повинна містити рівно {SEQ_LENGTH} рядків.")

        current_feature_count = input_data.shape[1]
        if current_feature_count <= TARGET_IDX:
            raise HTTPException(status_code=400, detail=f"У послідовності  
#{index + 1} відсутня колонка Close для зворотної нормалізації.")

        if feature_count is None:
            feature_count = current_feature_count

        elif current_feature_count != feature_count:

```

```

        raise HTTPException(status_code=400, detail="Усі послідовності
        повинні містити однакову кількість ознак.")

```

```

        validated_sequences.append(input_data)

    scaler_state = load_scaler_metadata(feature_count)
    if scaler_state is None:
        batch_scaler = MinMaxScaler(feature_range=(0, 1))
        batch_scaler.fit(np.concatenate(validated_sequences, axis=0))
        scaler_state = {
            "feature_mins": batch_scaler.data_min_.astype(np.float32),
            "feature_maxs": batch_scaler.data_max_.astype(np.float32),
        }

    for input_data in validated_sequences:
        scaled_sequences.append(scale_sequence(input_data,
        scaler_state["feature_mins"], scaler_state["feature_maxs"]))

    return np.stack(scaled_sequences), scaler_state

def inverse_transform_predictions(prediction_scaled, scaler_state):
    return inverse_scale_close_values(
        prediction_scaled,
        scaler_state["feature_mins"],
        scaler_state["feature_maxs"],
    )

def set_training_status(
    *,
    status: str,
    state: str,
    message: str,
    job_id: Optional[str] = None,
    started_at_utc: Optional[str] = None,
    finished_at_utc: Optional[str] = None,
) -> Dict[str, Optional[str]]:

```

```

with TRAINING_STATUS_LOCK:

    TRAINING_STATUS["status"] = status

    TRAINING_STATUS["state"] = state

    TRAINING_STATUS["job_id"] = job_id

    TRAINING_STATUS["message"] = message

    TRAINING_STATUS["started_at_utc"] = started_at_utc

    TRAINING_STATUS["finished_at_utc"] = finished_at_utc

    TRAINING_STATUS["updated_at_utc"] = utc_now_iso()


    snapshot = dict(TRAINING_STATUS)


    snapshot["model_loaded"] = os.path.exists(get_model_path())

    return snapshot


class TrainingCancellationCallback(tf.keras.callbacks.Callback):

    def __init__(self, job_id: str, started_at_utc: str):

        super().__init__()

        self.job_id = job_id

        self.started_at_utc = started_at_utc


    def on_train_batch_end(self, batch, logs=None):

        if is_training_stop_requested():

            self.model.stop_training = True

            set_training_status(

                status="stopping",

                state="Stopping",

                message="Отримано запит на зупинку донавчання. Завершуємо
поточний крок.",

                job_id=self.job_id,

                started_at_utc=self.started_at_utc,

                finished_at_utc=None,

            )


@app.get("/")

def read_root():

    model_exists = os.path.exists(get_model_path())

```

```

    return {
        "status": "AI Microservice is running",
        "model_loaded": model_exists,
        "architecture": "CNN-BiLSTM-Attention (Adaptive)"
    }

@app.get("/training-status")
def get_training_status():
    return get_training_status_snapshot()

@app.post("/stop-training")
def stop_training():
    current_status = get_training_status_snapshot()
    current_state = current_status.get("state")

    if current_state not in {"Queued", "Running", "Stopping"}:
        current_status["status"] = current_status.get("status") or "idle"
        current_status["message"] = current_status.get("message") or
        "Активне донавчання зараз не виконується."
        return JSONResponse(status_code=409, content=current_status)

    request_training_stop()
    return set_training_status(
        status="stopping",
        state="Stopping",
        message="Запит на зупинку донавчання прийнято.",
        job_id=current_status.get("job_id"),
        started_at_utc=current_status.get("started_at_utc"),
        finished_at_utc=None,
    )

# =====
# PREDICTION ENDPOINT (INFERENCE)
# =====

@app.post("/predict")
def get_prediction(request: PredictionRequest):

```

```

model_path = get_model_path()

if not os.path.exists(model_path):
    raise HTTPException(status_code=404, detail="Модель ще не навчена
або файл не знайдено.")

try:
    X_input, scaler_state = prepare_scaled_sequences([request.data])
    model = load_model_with_compat(model_path)
    base_prediction_scaled = model.predict(X_input, verbose=0)
    base_price = inverse_transform_predictions(base_prediction_scaled,
scaler_state)[0]

    final_price = base_price

    if xgb_model is not None:
        last_day_features = np.array([request.data[-1]],
dtype=np.float32)
        predicted_return =
float(np.ravel(xgb_model.predict(last_day_features))[0])
        last_actual_close = float(request.data[-1][TARGET_IDX])
        xgb_price = last_actual_close * (1 + predicted_return)
        final_price = (base_price * 0.6) + (xgb_price * 0.4)

    return {
        "predicted_close_price": float(final_price),
        "message": "Прогноз успішно згенеровано"
    }
except HTTPException:
    raise

except Exception as exc:
    raise HTTPException(status_code=500, detail=f"Помилка під час
прогнозування: {exc}") from exc

@app.post("/predict-batch")
def get_batch_prediction(request: BatchPredictionRequest):
    model_path = get_model_path()
    if not os.path.exists(model_path):

```

```

        raise HTTPException(status_code=404, detail="Модель ще не навчена
або файл не знайдено.")

    try:

        X_input, scaler_state = prepare_scaled_sequences(request.sequences)

        model = load_model_with_compat(model_path)

        prediction_scaled = model.predict(X_input, verbose=0)

        predicted_prices = inverse_transform_predictions(prediction_scaled,
scaler_state)

        return {

            "predicted_close_prices": predicted_prices,

            "message": "Пакетний прогноз успішно згенеровано"

        }

    except HTTPException:

        raise

    except Exception as exc:

        raise HTTPException(status_code=500, detail=f"Помилка під час
пакетного прогнозування: {exc}") from exc

# =====

# RETRAINING ENDPOINT (CONTINUOUS LEARNING)

# =====

@app.post("/retrain")

def retrain_model(request: RetrainRequest, background_tasks:
BackgroundTasks):

    if not os.path.exists(get_model_path()):

        raise HTTPException(status_code=404, detail="Базова модель
відсутня. Додайте файл multimodal_crypto_model.keras.")

    if len(request.data) <= SEQ_LENGTH:

        raise HTTPException(status_code=400, detail="Недостатньо даних для
формування хоча б одного ковзного вікна (треба > 30).")

    current_status = get_training_status_snapshot()

    if current_status["state"] in {"Queued", "Running"}:

        current_status["status"] = "busy"

        current_status["message"] = current_status["message"] or
"Донавчання вже виконується."

```

```

        return JsonResponse(status_code=409, content=current_status)

    clear_training_stop_request()
    job_id = str(uuid4())
    queued_status = set_training_status(
        status="queued",
        state="Queued",
        message="Запит на донавчання прийнято та поставлено в чергу.",
        job_id=job_id,
        started_at_utc=None,
        finished_at_utc=None,
    )

    # Add the retraining task to the background process so the C# server
    gets an immediate HTTP response
    background_tasks.add_task(perform_incremental_training, request.data,
                              job_id)

    return queued_status

def perform_incremental_training(raw_data: List[List[float]], job_id: str):
    print("Початок фонового донавчання моделі...")
    started_at_utc = utc_now_iso()

    if is_training_stop_requested():
        set_training_status(
            status="stopped",
            state="Stopped",
            message="Донавчання зупинено до старту виконання.",
            job_id=job_id,
            started_at_utc=None,
            finished_at_utc=utc_now_iso(),
        )
        clear_training_stop_request()
        return

    set_training_status(

```

```

        status="running",
        state="Running",
        message=f"Фонове донавчання виконується. Рядків у пакеті:
{len(raw_data)}.",
        job_id=job_id,
        started_at_utc=started_at_utc,
        finished_at_utc=None,
    )

    try:
        model = load_model_with_compat(get_model_path())
        model = prepare_model_for_training(model)

        data = np.array(raw_data)
        scaler = MinMaxScaler(feature_range=(0, 1))
        scaled_data = scaler.fit_transform(data)

        X, y = [], []
        for i in range(len(scaled_data) - SEQ_LENGTH):
            X.append(scaled_data[i:(i + SEQ_LENGTH)])
            y.append(scaled_data[i + SEQ_LENGTH, TARGET_IDX])

        X = np.array(X)
        y = np.array(y)

        # Fine-tune the model on new data (Transfer Learning)
        # Use only 5-10 epochs to avoid catastrophic forgetting of older
patterns
        model.fit(
            X,
            y,
            epochs=10,
            batch_size=16,
            verbose=1,
            callbacks=[TrainingCancellationCallback(job_id,
started_at_utc)],
        )

```

```

if is_training_stop_requested():
    set_training_status(
        status="stopped",
        state="Stopped",
        message="Донавчання було зупинено користувачем.",
        job_id=job_id,
        started_at_utc=started_at_utc,
        finished_at_utc=utc_now_iso(),
    )
    clear_training_stop_request()
    print("Донавчання зупинено користувачем.")
    return

# Save the updated weights
ensure_model_dir()
model.save(PRIMARY_MODEL_PATH)
save_scaler_metadata(scaler, PRIMARY_MODEL_PATH)
set_training_status(
    status="completed",
    state="Completed",
    message=f"Донавчання завершено успішно. Оновлено модель на {len(raw_data)} рядках.",
    job_id=job_id,
    started_at_utc=started_at_utc,
    finished_at_utc=utc_now_iso(),
)
clear_training_stop_request()
print("Донавчання завершено. Модель оновлено!")
except Exception as e:
    set_training_status(
        status="failed",
        state="Failed",
        message=f"Помилка під час донавчання: {e}",
        job_id=job_id,
        started_at_utc=started_at_utc,

```

```
        finished_at_utc=utc_now_iso(),  
    )  
    clear_training_stop_request()  
    print(f"Помилка під час донавчання: {e}")
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення					Найменування		Дод. відомості	
					Текстові документи			
1.					Пояснювальна записка		86 с.	