ДОДАТОК А

Лістинг програмного коду Python-сервісу (main.py)

```
from functools import lru_cache
from io import BytesIO
import base64

import cv2
import numpy as np
import tensorflow as tf
from fastapi import FastAPI, File, HTTPException, UploadFile
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse
from PIL import Image
from PIL.ExifTags import TAGS
from tensorflow.keras.applications.efficientnet import preprocess_input

from config import AI_THRESHOLD, IMAGE_SIZE, MODEL_PATH

app = FastAPI(
    title="AI Image Detector",
    description="Мультимодальне виявлення зображень, згенерованих штучним інтелектом",
    version="1.0.0",
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)
```

```
    @lru_cache(maxsize=1)
    def load_model():
        if not MODEL_PATH.exists():
            raise   FileNotFoundError(f"Model   not   found:
{MODEL_PATH}")
        return tf.keras.models.load_model(MODEL_PATH)

    def generate_gradcam(image: Image.Image) -> str:
        try:
            model = load_model()
            efficientnet = model.get_layer('efficientnetb0')
            last_conv_layer = None
            for layer in reversed(efficientnet.layers):
                if                          isinstance(layer,
tf.keras.layers.Conv2D):
                    last_conv_layer = layer.name
                    break
            if last_conv_layer is None:
                return ""
            grad_model = tf.keras.Model(
                inputs=efficientnet.input,
                outputs=[

efficientnet.get_layer(last_conv_layer).output,
                    efficientnet.output
                ]
            )
            img = image.resize(IMAGE_SIZE)
            arr = np.array(img, dtype=np.float32)
            arr = preprocess_input(arr)
            arr = np.expand_dims(arr, axis=0)
            with tf.GradientTape() as tape:
                conv_outputs, predictions = grad_model(arr)
                loss = predictions[:, 0]
            grads = tape.gradient(loss, conv_outputs)
```

```
            pooled_grads = tf.reduce_mean(grads, axis=(0, 1,
2))
            heatmap = conv_outputs[0] @ pooled_grads[...,
tf.newaxis]
            heatmap = tf.squeeze(heatmap)
            heatmap = tf.maximum(heatmap, 0)
            max_val = tf.reduce_max(heatmap)
            if max_val > 0:
                heatmap = heatmap / max_val
            heatmap = heatmap.numpy()
            img_array = np.array(image.resize(IMAGE_SIZE))
            heatmap_resized = cv2.resize(heatmap,
IMAGE_SIZE)
            heatmap_colored = cv2.applyColorMap(
                np.uint8(255 * heatmap_resized),
cv2.COLORMAP_JET
            )
            heatmap_colored = cv2.cvtColor(heatmap_colored,
cv2.COLOR_BGR2RGB)
            superimposed = cv2.addWeighted(img_array, 0.6,
heatmap_colored, 0.4, 0)
            buffer = BytesIO()
            Image.fromarray(superimposed).save(buffer,
format="JPEG", quality=85)
            return "" +
base64.b64encode(buffer.getvalue()).decode()
        except Exception as e:
            print(f"Grad-CAM error: {e}")
            return ""

    def analyze_pixels(image: Image.Image) -> float:
        img = image.resize(IMAGE_SIZE)
        arr = np.array(img, dtype=np.float32)
        arr = preprocess_input(arr)
        arr = np.expand_dims(arr, axis=0)
        model = load_model()
```

```
        prediction = model.predict(arr, verbose=0)
        if prediction.ndim == 2 and prediction.shape[1] ==
1:
            return 1.0 - float(prediction[0][0])
        return 1.0 - float(prediction[0])

    def analyze_frequency(image: Image.Image) -> float:
        gray         =         np.array(image.convert("L"),
dtype=np.float32)
        fft = np.fft.fft2(gray)
        fft_shifted = np.fft.fftshift(fft)
        magnitude = np.log1p(np.abs(fft_shifted))
        h, w = magnitude.shape
        cy, cx = h // 2, w // 2
        radius_low = min(h, w) // 8
        radius_high = min(h, w) // 4
        y_idx, x_idx = np.ogrid[:h, :w]
        dist = np.sqrt((y_idx - cy) ** 2 + (x_idx - cx) **
2)
        energy_low = magnitude[dist < radius_low].mean()
        energy_high = magnitude[(dist > radius_low) & (dist
< radius_high)].mean()
        ratio = energy_high / (energy_low + 1e-8)
        return float(np.clip((ratio - 0.3) / 0.4, 0.0, 1.0))

    def analyze_exif(image: Image.Image) -> float:
        AI_SOFTWARE_KEYWORDS = {
            "stable  diffusion",  "midjourney",  "dall-e",
"dalle",
            "firefly", "imagen", "novelai", "automatic1111",
            "comfyui", "invokeai", "adobe firefly",
        }
        try:
            exif_data = image._getexif()
        except Exception:
            exif_data = None
```

```
        if not exif_data:
            return 0.75
        tags = {TAGS.get(k, k): v for k, v in
exif_data.items()}
        software = str(tags.get("Software", "")).lower()
        if any(kw in software for kw in
AI_SOFTWARE_KEYWORDS):
            return 1.0
        real_signals = 0
        if tags.get("Make"):            real_signals += 1
        if tags.get("Model"):           real_signals += 1
        if tags.get("GPSInfo"):         real_signals += 1
        if tags.get("DateTime"):        real_signals += 1
        if tags.get("ExposureTime"):    real_signals += 1
        if tags.get("ISOSpeedRatings"): real_signals += 1
        return float(max(0.0, 1.0 - real_signals * 0.18))

    WEIGHTS = {"pixels": 0.65, "frequency": 0.20, "exif":
0.15}

    def combine_scores(pixel_score, freq_score, exif_score):
        return (
            pixel_score * WEIGHTS["pixels"] +
            freq_score  * WEIGHTS["frequency"] +
            exif_score  * WEIGHTS["exif"]
        )

    @app.get("/health")
    def health():
        return {"status": "ok"}

    @app.post("/predict")
    async def predict(file: UploadFile = File(...)):
        try:
            file_bytes = await file.read()
```

```
            image                                                =
Image.open(BytesIO(file_bytes)).convert("RGB")
            pixel_score = analyze_pixels(image)
            freq_score  = analyze_frequency(image)
            exif_score  = analyze_exif(image)
            ai_probability   =   combine_scores(pixel_score,
freq_score, exif_score)
            label  =  "ai_generated"  if  ai_probability  >=
AI_THRESHOLD else "real"
            confidence   =   ai_probability   if   label   ==
"ai_generated" else 1.0 - ai_probability
            gradcam_image = generate_gradcam(image)
            return JSONResponse(content={
                "label":          label,
                "confidence":     round(confidence, 4),
                "ai_probability": round(ai_probability, 4),
                "filename":       file.filename,
                "gradcam":        gradcam_image,
                "channels": {
                    "pixel_score":    round(pixel_score, 4),
                    "frequency_score": round(freq_score, 4),
                    "exif_score":      round(exif_score, 4),
                },
            })
        except Exception as e:
            raise               HTTPException(status_code=500,
detail=str(e))
```

ДОДАТОК Б

Лістинг програмного коду навчання моделі (train.py)

```
    from pathlib import Path

    import matplotlib.pyplot as plt
    import numpy as np
    import tensorflow as tf
    from sklearn.utils.class_weight import
compute_class_weight
    from tensorflow.keras import layers, models
    from tensorflow.keras.applications import EfficientNetB0
    from tensorflow.keras.applications.efficientnet import
preprocess_input
    from tensorflow.keras.callbacks import EarlyStopping,
ModelCheckpoint, ReduceLROnPlateau

    import sys
    sys.path.append(str(Path(__file__).resolve().parent.par
ent))
    from config import IMAGE_SIZE, MODEL_PATH
    from ml.dataset_loader import DatasetLoader

    def apply_preprocessing(dataset):
        return dataset.map(
            lambda images, labels:
(preprocess_input(images), labels),
            num_parallel_calls=tf.data.AUTOTUNE
        ).prefetch(tf.data.AUTOTUNE)

    def get_class_weights(dataset):
        labels = []
        for _, batch_labels in dataset:

labels.extend(batch_labels.numpy().flatten().tolist())
```

```
        labels = np.array(labels, dtype=np.int32)
        weights                                                    =
compute_class_weight(class_weight="balanced",
classes=np.array([0, 1]), y=labels)
        return {0: float(weights[0]), 1: float(weights[1])}

    def build_model():
        base_model = EfficientNetB0(
            input_shape=IMAGE_SIZE + (3,),
            include_top=False,
            weights="imagenet"
        )
        base_model.trainable = False
        augmentation = models.Sequential([
            layers.RandomFlip("horizontal"),
            layers.RandomRotation(0.05),
            layers.RandomZoom(0.1),
            layers.RandomContrast(0.1),
        ], name="augmentation")
        inputs = layers.Input(shape=IMAGE_SIZE + (3,))
        x = augmentation(inputs)
        x = base_model(x, training=False)
        x = layers.GlobalAveragePooling2D()(x)
        x = layers.Dropout(0.3)(x)
        x = layers.Dense(128, activation="relu")(x)
        x = layers.Dropout(0.2)(x)
        outputs = layers.Dense(1, activation="sigmoid")(x)
        model = models.Model(inputs, outputs)
        model.compile(

optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
            loss="binary_crossentropy",
            metrics=["accuracy"]
        )
        return model, base_model
```

```
    def fine_tune(model, base_model):
        base_model.trainable = True
        for layer in base_model.layers[:200]:
            layer.trainable = False
        model.compile(
            optimizer=tf.keras.optimizers.Adam(1e-5),
            loss="binary_crossentropy",
            metrics=["accuracy"]
        )
        return model

    def plot_history(h1, h2=None):
        acc          =     h1.history["accuracy"]          +
(h2.history["accuracy"]  if h2 else [])
        vacc      =      h1.history["val_accuracy"]      +
(h2.history["val_accuracy"] if h2 else [])
        loss = h1.history["loss"]     + (h2.history["loss"]
if h2 else [])
        vloss=       h1.history["val_loss"]              +
(h2.history["val_loss"]  if h2 else [])
        out = Path("models")
        out.mkdir(exist_ok=True)
        for values, labels, title, fname in [
            ([acc, vacc],    ["train", "val"], "Accuracy",
"accuracy_plot.png"),
            ([loss,  vloss],  ["train",  "val"],  "Loss",
"loss_plot.png"),
        ]:
            plt.figure(figsize=(10, 5))
            for v, l in zip(values, labels):
                plt.plot(v, label=l)
            plt.title(title)
            plt.legend()
            plt.tight_layout()
            plt.savefig(out / fname)
            plt.close()
```

```
    def main():
        loader = DatasetLoader()
        train_raw, val_raw, test_raw = loader.load_all()
        class_weights = get_class_weights(train_raw)
        train_ds = apply_preprocessing(train_raw)
        val_ds   = apply_preprocessing(val_raw)
        test_ds  = apply_preprocessing(test_raw)
        model, base_model = build_model()
        model.summary()
        callbacks = [
            ModelCheckpoint(str(MODEL_PATH),
monitor="val_loss", save_best_only=True, verbose=1),
            EarlyStopping(monitor="val_loss",   patience=4,
restore_best_weights=True, verbose=1),
            ReduceLROnPlateau(monitor="val_loss",
factor=0.3, patience=2, min_lr=1e-7, verbose=1),
        ]
        h1  =  model.fit(train_ds,  validation_data=val_ds,
epochs=8,
                         class_weight=class_weights,
callbacks=callbacks)
        model = fine_tune(model, base_model)
        h2  =  model.fit(train_ds,  validation_data=val_ds,
epochs=8,
                         class_weight=class_weights,
callbacks=callbacks)
        best = tf.keras.models.load_model(MODEL_PATH)
        loss, acc = best.evaluate(test_ds)
        print(f"Test  loss:  {loss:.4f}   |   Test  accuracy:
{acc:.4f}")
        plot_history(h1, h2)
    if __name__ == "__main__":
      main()
```

ДОДАТОК В

Лістинг програмного коду Go-сервісу (main.go)

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "html/template"
    "io"
    "log"
    "mime/multipart"
    "net/http"
    "os"
)

type PredictionResponse struct {
    Label         string  `json:"label"`
    Confidence    float64 `json:"confidence"`
    AIProbability float64 `json:"ai_probability"`
    Filename      string  `json:"filename"`
}

func pythonURL() string {
    if url := os.Getenv("PYTHON_SERVICE_URL"); url != ""
{
        return url
    }
    return "http://localhost:8000"
}

func indexHandler(w http.ResponseWriter, r
*http.Request) {
```

```
        tmpl, err := template.ParseFiles("templates/index.html")
        if err != nil {
            http.Error(w, "Template error", http.StatusInternalServerError)
            return
        }
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        tmpl.Execute(w, nil)
    }

    func uploadHandler(w http.ResponseWriter, r *http.Request) {
        if r.Method != http.MethodPost {
            http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
            return
        }
        r.ParseMultipartForm(10 << 20)
        file, header, err := r.FormFile("file")
        if err != nil {
            jsonError(w, "Failed to read file", http.StatusBadRequest)
            return
        }
        defer file.Close()
        fileBytes, err := io.ReadAll(file)
        if err != nil {
            jsonError(w, "Failed to read file bytes", http.StatusInternalServerError)
            return
        }
        body := &bytes.Buffer{}
        writer := multipart.NewWriter(body)
```

```
        part,     _     :=     writer.CreateFormFile("file",
header.Filename)
        part.Write(fileBytes)
        writer.Close()
        req,       _       :=       http.NewRequest("POST",
fmt.Sprintf("%s/predict", pythonURL()), body)
        req.Header.Set("Content-Type",
writer.FormDataContentType())
        resp, err := http.DefaultClient.Do(req)
        if err != nil {
            jsonError(w,    "ML    service    unavailable",
http.StatusServiceUnavailable)
            return
        }
        defer resp.Body.Close()
        respBytes, _ := io.ReadAll(resp.Body)
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(resp.StatusCode)
        w.Write(respBytes)
    }

    func jsonError(w http.ResponseWriter, msg string, code
int) {
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(code)

json.NewEncoder(w).Encode(map[string]string{"error": msg})
    }

    func main() {
        port := os.Getenv("PORT")
        if port == "" {
            port = "3000"
        }
        http.HandleFunc("/", indexHandler)
        http.HandleFunc("/upload", uploadHandler)
```

```
        http.HandleFunc("/health",                    func(w
http.ResponseWriter, r *http.Request) {
            w.Header().Set("Content-Type",
"application/json")
            w.Write([]byte(`{"status":"ok"}`))
        })
        log.Printf("Starting on http://localhost:%s", port)
        log.Fatal(http.ListenAndServe(":"+port, nil))
  }
```

ДОДАТОК Г

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	70 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Гуков Д.Р.			Мультимодальне виявлення зображень, згенерованих штучним інтелектом	Шифр групи	Код напр./спец.
Перевірив		Терзіян В.Я.				СШІм-24-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Чала Л.Е.					