YOLOv8

YOLO (You Only Look Once) .

.

```
ai_model/
├──── data/
│    ├──── raw/
│    ├──── processed/
│    └──── annotations/
├──── models/
│    ├──── pretrained/
│    └──── trained/
├──── scripts/
│    ├──── data_preparation.py
│    ├──── train.py
│    ├──── evaluate.py
│    └──── predict.py
├──── utils/
│    ├──── augmentation.py
│    ├──── visualization.py
│    └──── metrics.py
├──── app/
│    ├──── api.py
│    └──── inference.py
└──── requirements.txt
```

`requirements.txt` :

```
 ultralytics>=8.0.0
torch>=2.0.0
torchvision>=0.15.0
opencv-python>=4.7.0
numpy>=1.24.0
Pillow>=9.4.0
```

```
matplotlib>=3.7.0
albumentations>=1.3.0
fastapi>=0.95.0
uvicorn>=0.21.0
python-multipart>=0.0.6
```

1.

```python
# setup_environment.py
import os
import subprocess

def setup_environment():
    #
    directories = [
        'data/raw',
        'data/processed',
        'data/annotations',
        'models/pretrained',
        'models/trained',
        'scripts',
        'utils',
        'app'
    ]

    for directory in directories:
        os.makedirs(directory, exist_ok=True)

    #
    subprocess.run(['pip', 'install', '-r', 'requirements.txt'])

    print("                          !")

if __name__ == "__main__":
    setup_environment()
```

2.

:

```python
# scripts/data_preparation.py
import os
import shutil
import random
import cv2
import numpy as np
```

```python
from pathlib import Path
import yaml
import albumentations as A
from tqdm import tqdm

def create_dataset_splits(data_dir, output_dir, split_ratio=(0.7, 0.15, 0.15)):
    """



    Args:
        data_dir:
        output_dir:
        split_ratio: (                    )
    """
    # 1
    assert sum(split_ratio) == 1.0, "1                                    "

    #
    train_dir = os.path.join(output_dir, 'train')
    val_dir = os.path.join(output_dir, 'val')
    test_dir = os.path.join(output_dir, 'test')

    for directory in [train_dir, val_dir, test_dir]:
        os.makedirs(os.path.join(directory, 'images'), exist_ok=True)
        os.makedirs(os.path.join(directory, 'labels'), exist_ok=True)

    #
    image_files = [f for f in os.listdir(os.path.join(data_dir, 'images')) if
f.endswith(('.jpg', '.jpeg', '.png'))]
    random.shuffle(image_files)

    #
    n_total = len(image_files)
    n_train = int(n_total * split_ratio[0])
    n_val = int(n_total * split_ratio[1])

    #
    train_files = image_files[:n_train]
    val_files = image_files[n_train:n_train+n_val]
    test_files = image_files[n_train+n_val:]

    #
    for files, target_dir in zip([train_files, val_files, test_files], [train_dir, val_dir,
test_dir]):
        for file in tqdm(files, desc=f"                    {os.path.basename(target_dir)}"):
            #
            src_img = os.path.join(data_dir, 'images', file)
            dst_img = os.path.join(target_dir, 'images', file)
            shutil.copy(src_img, dst_img)

            # (                )
            label_file = os.path.splitext(file)[0] + '.txt'
```

```
            src_label = os.path.join(data_dir, 'labels', label_file)
            if os.path.exists(src_label):
                dst_label = os.path.join(target_dir, 'labels', label_file)
                shutil.copy(src_label, dst_label)

    #                    YAML     YOLOv8
    yaml_content = {
        'path': os.path.abspath(output_dir),
        'train': os.path.join('train', 'images'),
        'val': os.path.join('val', 'images'),
        'test': os.path.join('test', 'images'),
        'nc': 3,  #
        'names': ['crack', 'corrosion', 'exposed_rebar']  #
    }

    with open(os.path.join(output_dir, 'dataset.yaml'), 'w') as f:
        yaml.dump(yaml_content, f, default_flow_style=False)

    print(f"                              :{n_train}            {n_val}            {len(test_files)}
            ")
    return os.path.join(output_dir, 'dataset.yaml')

def apply_augmentations(data_dir, output_dir, augmentation_factor=3):
    """


    Args:
        data_dir:
        output_dir:
        augmentation_factor:
    """
    #
    os.makedirs(os.path.join(output_dir, 'images'), exist_ok=True)
    os.makedirs(os.path.join(output_dir, 'labels'), exist_ok=True)

    #
    augmentations = [
        A.HorizontalFlip(p=0.5),
        A.RandomBrightnessContrast(p=0.5),
        A.RandomGamma(p=0.5),
        A.GaussNoise(p=0.5),
        A.Blur(blur_limit=3, p=0.3),
        A.Rotate(limit=10, p=0.5)
    ]

    transform = A.Compose(
        augmentations,
        bbox_params=A.BboxParams(format='yolo', label_fields=['class_labels'])
    )

    #
    image_files = [f for f in os.listdir(os.path.join(data_dir, 'images')) if
```

```
f.endswith(('.jpg', '.jpeg', '.png'))]

    for img_file in tqdm(image_files, desc="                    "):
        #
        img_path = os.path.join(data_dir, 'images', img_file)
        img = cv2.imread(img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        #
        label_file = os.path.splitext(img_file)[0] + '.txt'
        label_path = os.path.join(data_dir, 'labels', label_file)

        bboxes = []
        class_labels = []

        if os.path.exists(label_path):
            with open(label_path, 'r') as f:
                for line in f:
                    data = line.strip().split()
                    class_id = int(data[0])
                    x_center, y_center, width, height = map(float, data[1:5])
                    bboxes.append([x_center, y_center, width, height])
                    class_labels.append(class_id)

        #
        original_img_out_path = os.path.join(output_dir, 'images', img_file)
        original_label_out_path = os.path.join(output_dir, 'labels', label_file)

        cv2.imwrite(original_img_out_path, cv2.cvtColor(img, cv2.COLOR_RGB2BGR))
        if os.path.exists(label_path):
            shutil.copy(label_path, original_label_out_path)

        #
        for i in range(augmentation_factor):
            if not bboxes:  #
                augmented = transform(image=img)
                augmented_img = augmented['image']
            else:
                augmented = transform(image=img, bboxes=bboxes,
class_labels=class_labels)
                augmented_img = augmented['image']
                augmented_bboxes = augmented['bboxes']
                augmented_class_labels = augmented['class_labels']

            #
            aug_img_file = f"{os.path.splitext(img_file)[0]}_aug_{i+1}
{os.path.splitext(img_file)[1]}"
            aug_label_file = f"{os.path.splitext(img_file)[0]}_aug_{i+1}.txt"

            aug_img_path = os.path.join(output_dir, 'images', aug_img_file)
            aug_label_path = os.path.join(output_dir, 'labels', aug_label_file)
```

```
            cv2.imwrite(aug_img_path, cv2.cvtColor(augmented_img,
cv2.COLOR_RGB2BGR))

            #
            if bboxes:
                with open(aug_label_path, 'w') as f:
                    for bbox, class_id in zip(augmented_bboxes, augmented_class_labels):
                        f.write(f"{class_id} {' '.join(map(str, bbox))}\n")

    print(f"              .                                {len(image_files) *
augmentation_factor}                .")

if __name__ == "__main__":
    #
    data_dir = 'data/raw'
    processed_dir = 'data/processed'

    #
    yaml_path = create_dataset_splits(data_dir, processed_dir)

    #
    train_dir = os.path.join(processed_dir, 'train')
    augmented_train_dir = os.path.join(processed_dir, 'train_augmented')
    apply_augmentations(train_dir, augmented_train_dir)

    #             YAML
    with open(yaml_path, 'r') as f:
        yaml_content = yaml.safe_load(f)

    yaml_content['train'] = os.path.join('train_augmented', 'images')

    with open(yaml_path, 'w') as f:
        yaml.dump(yaml_content, f, default_flow_style=False)

    print("                          !")
```

3.   YOLOv8

YOLOv8   :

```
 # scripts/train.py
import os
import yaml
import argparse
from ultralytics import YOLO

def train_model(data_yaml, model_size='m', epochs=100, batch_size=16,
img_size=640, device='0'):
    """
```

```
                YOLOv8

    Args:
        data_yaml:              YAML
        model_size:                  ('n', 's', 'm', 'l', 'x')
        epochs:
        batch_size:
        img_size:
        device:                              ('cpu', '0', '0,1', etc.)

    Returns:

    """
    #
    with open(data_yaml, 'r') as f:
        data_config = yaml.safe_load(f)

    #
    num_classes = data_config['nc']

    #                YOLOv8
    model = YOLO(f'yolov8{model_size}.pt')

    #
    model.model.model[-1].nc = num_classes

    #
    results = model.train(
        data=data_yaml,
        epochs=epochs,
        batch=batch_size,
        imgsz=img_size,
        device=device,
        project='models',
        name='trained',
        exist_ok=True,
        pretrained=True,
        optimizer='Adam',  #           Adam            SGD
        lr0=0.001,  #
        lrf=0.01,  #
        momentum=0.937,
        weight_decay=0.0005,
        warmup_epochs=3.0,
        warmup_momentum=0.8,
        warmup_bias_lr=0.1,
        box=7.5,  #
        cls=0.5,  #
        hsv_h=0.015,  #               HSV
        hsv_s=0.7,
        hsv_v=0.4,
        degrees=0.0,  #         -/+
        translate=0.1,  #        -/+
```

```
        scale=0.5,  #       -/+
        shear=0.0,  #         -/+
        perspective=0.0,  #       -/+
        flipud=0.0,  #
        fliplr=0.5,  #
        mosaic=1.0,  #
        mixup=0.0,  #
        copy_paste=0.0  #
    )

    #
    best_model_path = os.path.join('models', 'trained', 'weights', 'best.pt')

    print(f"          !                          :{best_model_path}")
    return best_model_path

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='                YOLOv8
                    ')
    parser.add_argument('--data', type=str, default='data/processed/dataset.yaml',
help='            YAML            ')
    parser.add_argument('--model-size', type=str, default='m', choices=['n', 's', 'm', 'l',
'x'], help='                ')
    parser.add_argument('--epochs', type=int, default=100, help='
            ')
    parser.add_argument('--batch-size', type=int, default=16, help='              ')
    parser.add_argument('--img-size', type=int, default=640, help='
          ')
    parser.add_argument('--device', type=str, default='0', help='
          ')

    args = parser.parse_args()

    train_model(
        args.data,
        args.model_size,
        args.epochs,
        args.batch_size,
        args.img_size,
        args.device
    )
```

4.

:

```
 # scripts/evaluate.py
import os
import argparse
```

```python
import json
from ultralytics import YOLO
import matplotlib.pyplot as plt
import numpy as np
from pathlib import Path

def evaluate_model(model_path, data_yaml, img_size=640, device='0'):
    """


    Args:
        model_path:
        data_yaml:          YAML
        img_size:
        device:

    Returns:

    """
    #
    model = YOLO(model_path)

    #
    results = model.val(
        data=data_yaml,
        imgsz=img_size,
        device=device,
        project='models',
        name='evaluation',
        exist_ok=True,
        verbose=True
    )

    #
    results_dir = os.path.join('models', 'evaluation')
    os.makedirs(results_dir, exist_ok=True)

    #
    metrics = {
        'precision': float(results.box.p),  #
        'recall': float(results.box.r),  #
        'mAP50': float(results.box.map50),  #                  IoU=0.5
        'mAP50-95': float(results.box.map),  #                   IoU=0.5:0.95
        'fitness': float(results.fitness)  #
    }

    #                    JSON
    with open(os.path.join(results_dir, 'metrics.json'), 'w') as f:
        json.dump(metrics, f, indent=4)

    #
    plt.figure(figsize=(10, 6))
```

```
    plt.bar(metrics.keys(), metrics.values())
    plt.title('Model Evaluation Metrics')
    plt.ylabel('Score')
    plt.ylim(0, 1)
    plt.savefig(os.path.join(results_dir, 'metrics.png'))

    # (                    )              -
    if hasattr(results, 'pr_curve'):
        plt.figure(figsize=(10, 6))
        for i, class_name in enumerate(results.names):
            precision = results.pr_curve[i][:, 0]
            recall = results.pr_curve[i][:, 1]
            plt.plot(recall, precision, label=f'{class_name} (AP={results.ap[i]:.3f})')

        plt.xlabel('Recall')
        plt.ylabel('Precision')
        plt.title('Precision-Recall Curve')
        plt.legend()
        plt.grid(True)
        plt.savefig(os.path.join(results_dir, 'pr_curve.png'))

    print(f"                          !                                :{results_dir}")
    return metrics

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='                  YOLOv8
                        ')
    parser.add_argument('--model', type=str, default='models/trained/weights/
best.pt', help='                              ')
    parser.add_argument('--data', type=str, default='data/processed/dataset.yaml',
help='              YAML              ')
    parser.add_argument('--img-size', type=int, default=640, help='
            ')
    parser.add_argument('--device', type=str, default='0', help='
            ')

    args = parser.parse_args()

    evaluate_model(
        args.model,
        args.data,
        args.img_size,
        args.device
    )
```

5.

:

```python
# scripts/predict.py
import os
import argparse
import cv2
import numpy as np
from ultralytics import YOLO
from pathlib import Path
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from PIL import Image

def predict_defects(model_path, image_path, output_dir='output',
conf_threshold=0.25, img_size=640, device='0'):
    """


    Args:
        model_path:
        image_path:
        output_dir:
        conf_threshold:
        img_size:
        device:

    Returns:

    """
    #
    model = YOLO(model_path)

    #
    results = model.predict(
        source=image_path,
        conf=conf_threshold,
        imgsz=img_size,
        device=device,
        save=True,
        project=output_dir,
        name='predictions',
        exist_ok=True,
        verbose=True
    )

    #
    os.makedirs(os.path.join(output_dir, 'predictions'), exist_ok=True)

    #
    for i, result in enumerate(results):
        #
        if isinstance(image_path, list):
            img_name = os.path.basename(image_path[i])
```

```
        else:
            img_name = os.path.basename(image_path)

        #
        img = cv2.imread(result.path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        #
        plt.figure(figsize=(12, 8))
        plt.imshow(img)

        #
        boxes = result.boxes.xyxy.cpu().numpy()
        confs = result.boxes.conf.cpu().numpy()
        cls_ids = result.boxes.cls.cpu().numpy().astype(int)

        for box, conf, cls_id in zip(boxes, confs, cls_ids):
            x1, y1, x2, y2 = box
            class_name = result.names[cls_id]

            #
            rect = patches.Rectangle(
                (x1, y1), x2-x1, y2-y1,
                linewidth=2,
                edgecolor=plt.cm.tab10(cls_id),
                facecolor='none'
            )
            plt.gca().add_patch(rect)

            #
            plt.text(
                x1, y1-5,
                f'{class_name} {conf:.2f}',
                color='white',
                fontsize=10,
                bbox=dict(facecolor=plt.cm.tab10(cls_id), alpha=0.8, edgecolor='none',
pad=1)
            )

        #
        output_path = os.path.join(output_dir, 'predictions',
f'result_{os.path.splitext(img_name)[0]}.png')
        plt.axis('off')
        plt.tight_layout()
        plt.savefig(output_path, dpi=300, bbox_inches='tight')
        plt.close()

        #
        defect_counts = {}
        for cls_id in cls_ids:
            class_name = result.names[cls_id]
            defect_counts[class_name] = defect_counts.get(class_name, 0) + 1
```

```
        report_path = os.path.join(output_dir, 'predictions',
f'report_{os.path.splitext(img_name)[0]}.txt')
        with open(report_path, 'w') as f:
            f.write(f"                                    :{img_name}\n")
            f.write("="*50 + "\n\n")
            f.write(f"                              :{len(boxes)}\n\n")
            f.write("                  :\n")
            for class_name, count in defect_counts.items():
                f.write(f"- {class_name}: {count}\n")

            f.write("\n                  :\n")
            for i, (box, conf, cls_id) in enumerate(zip(boxes, confs, cls_ids)):
                x1, y1, x2, y2 = box
                class_name = result.names[cls_id]
                width = x2 - x1
                height = y2 - y1
                area = width * height

                f.write(f"        #{i+1}:\n")
                f.write(f"         : {class_name}\n")
                f.write(f"              : {conf:.2f}\n")
                f.write(f"           : ({x1:.1f}, {y1:.1f})      ({x2:.1f}, {y2:.1f})\n")
                f.write(f"             : {width:.1f} × {height:.1f}       \n")
                f.write(f"                          :{area:.1f}             \n\n")

    print(f"                    !                  :{os.path.join(output_dir, 'predictions')}")
    return results

if __name__ == "__main__":
    parser =
argparse.ArgumentParser(description='
YOLOv8')
    parser.add_argument('--model', type=str, default='models/trained/weights/
best.pt', help='                          )
    parser.add_argument('--image', type=str, required=True, help='
        ')
    parser.add_argument('--output', type=str, default='output', help='
         ')
    parser.add_argument('--conf', type=float, default=0.25, help='                    )
    parser.add_argument('--img-size', type=int, default=640, help='                      )
    parser.add_argument('--device', type=str, default='0', help='
        ')

    args = parser.parse_args()

    predict_defects(
        args.model,
        args.image,
        args.output,
        args.conf,
        args.img_size,
```

```
        args.device
    )
```

6. (API)

FastAPI :

```python
 # app/api.py
import os
import io
import base64
import json
import uuid
from fastapi import FastAPI, File, UploadFile, Form, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse
import uvicorn
from PIL import Image
import numpy as np
import cv2
from ultralytics import YOLO
import sys

#
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

#
app = FastAPI(title="Structural Defect Detection API",
description="API for detecting structural defects in images")

#         CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  #
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

#
MODEL_PATH = os.environ.get("MODEL_PATH", "models/trained/weights/best.pt")
model = None

@app.on_event("startup")
async def startup_event():
    global model
    try:
        model = YOLO(MODEL_PATH)
        print(f"                                :{MODEL_PATH}")
```

```
    except Exception as e:
        print(f"                    :{e}")
        model = None

@app.get("/")
async def root():
    return {"message": "                                                    }

@app.get("/health")
async def health_check():
    if model is None:
        raise HTTPException(status_code=503, detail="                    ")
    return {"status": "healthy", "model_loaded": True}

@app.post("/detect")
async def detect_defects(
    file: UploadFile = File(...),
    conf_threshold: float = Form(0.25),
    img_size: int = Form(640)
):
    if model is None:
        raise HTTPException(status_code=503, detail="                    ")

    try:
        #
        contents = await file.read()
        image = Image.open(io.BytesIO(contents))

        #
        temp_file_name = f"temp_{uuid.uuid4()}.jpg"
        temp_file_path = os.path.join("/tmp", temp_file_name)
        image.save(temp_file_path)

        #
        results = model.predict(
            source=temp_file_path,
            conf=conf_threshold,
            imgsz=img_size,
            save=False,
            verbose=False
        )[0]

        #
        boxes = results.boxes.xyxy.cpu().numpy().tolist()
        confs = results.boxes.conf.cpu().numpy().tolist()
        cls_ids = results.boxes.cls.cpu().numpy().astype(int).tolist()

        #
        defects = []
        for box, conf, cls_id in zip(boxes, confs, cls_ids):
            x1, y1, x2, y2 = box
            width = x2 - x1
```

```
            height = y2 - y1
            area = width * height

            defect = {
                "type": results.names[cls_id],
                "confidence": conf,
                "location": {
                    "x1": x1,
                    "y1": y1,
                    "x2": x2,
                    "y2": y2
                },
                "dimensions": {
                    "width": width,
                    "height": height,
                    "area": area
                }
            }
            defects.append(defect)

        #
        defect_summary = {}
        for defect in defects:
            defect_type = defect["type"]
            defect_summary[defect_type] = defect_summary.get(defect_type, 0) + 1

        #
        img = cv2.imread(temp_file_path)
        for defect in defects:
            x1, y1 = int(defect["location"]["x1"]), int(defect["location"]["y1"])
            x2, y2 = int(defect["location"]["x2"]), int(defect["location"]["y2"])
            defect_type = defect["type"]
            conf = defect["confidence"]

            #
            if defect_type == "crack":
                color = (0, 0, 255)  #
            elif defect_type == "corrosion":
                color = (0, 165, 255)  #
            elif defect_type == "exposed_rebar":
                color = (0, 255, 255)  #
            else:
                color = (255, 0, 0)  #

            #
            cv2.rectangle(img, (x1, y1), (x2, y2), color, 2)

            #
            label = f"{defect_type} {conf:.2f}"
            (w, h), _ = cv2.getTextSize(label, cv2.FONT_HERSHEY_SIMPLEX, 0.5, 1)
            cv2.rectangle(img, (x1, y1 - 20), (x1 + w, y1), color, -1)
            cv2.putText(img, label, (x1, y1 - 5), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,
```

```
255, 255), 1)

        #                    Base64
        _, buffer = cv2.imencode('.jpg', img)
        img_base64 = base64.b64encode(buffer).decode('utf-8')

        #
        os.remove(temp_file_path)

        #
        response = {
            "success": True,
            "total_defects": len(defects),
            "defect_summary": defect_summary,
            "defects": defects,
            "image": f"{img_base64}"
        }

        return JSONResponse(content=response)

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"                    :
{str(e)}")

if __name__ == "__main__":
    uvicorn.run("api:app", host="0.0.0.0", port=8000, reload=True)
```

7.

                                                    :

```
 # app/inference.py
import os
import cv2
import numpy as np
from ultralytics import YOLO
import base64
import io
from PIL import Image

class DefectDetector:
    """
                                                YOLOv8
    """
    def __init__(self, model_path, conf_threshold=0.25, img_size=640, device='0'):
        """


        Args:
```

```
        model_path:
        conf_threshold:
        img_size:
        device:
    """
    self.model = YOLO(model_path)
    self.conf_threshold = conf_threshold
    self.img_size = img_size
    self.device = device

def detect_from_file(self, image_path):
    """


    Args:
        image_path:

    Returns:

    """
    results = self.model.predict(
        source=image_path,
        conf=self.conf_threshold,
        imgsz=self.img_size,
        device=self.device,
        verbose=False
    )[0]

    return self._process_results(results, cv2.imread(image_path))

def detect_from_image(self, image):
    """


    Args:
        image:              (NumPy array     PIL Image)

    Returns:

    """
    #                     NumPy array             PIL Image
    if isinstance(image, Image.Image):
        image_np = np.array(image)
        #            RGB      BGR                    RGB
        if image_np.shape[-1] == 3:
            image_np = cv2.cvtColor(image_np, cv2.COLOR_RGB2BGR)
    else:
        image_np = image

    results = self.model.predict(
        source=image_np,
        conf=self.conf_threshold,
```

```
            imgsz=self.img_size,
            device=self.device,
            verbose=False
        )[0]

        return self._process_results(results, image_np)

    def detect_from_bytes(self, image_bytes):
        """


        Args:
            image_bytes:

        Returns:

        """
        image = Image.open(io.BytesIO(image_bytes))
        return self.detect_from_image(image)

    def detect_from_base64(self, base64_string):
        """
                                    Base64

        Args:
            base64_string:          Base64

        Returns:

        """
        #
        if ',' in base64_string:
            base64_string = base64_string.split(',')[1]

        image_bytes = base64.b64decode(base64_string)
        return self.detect_from_bytes(image_bytes)

    def _process_results(self, results, original_image):
        """


        Args:
            results:
            original_image:

        Returns:

        """
        #
        boxes = results.boxes.xyxy.cpu().numpy().tolist()
        confs = results.boxes.conf.cpu().numpy().tolist()
        cls_ids = results.boxes.cls.cpu().numpy().astype(int).tolist()
```

```
    #
    defects = []
    for box, conf, cls_id in zip(boxes, confs, cls_ids):
        x1, y1, x2, y2 = box
        width = x2 - x1
        height = y2 - y1
        area = width * height

        defect = {
            "type": results.names[cls_id],
            "confidence": conf,
            "location": {
                "x1": x1,
                "y1": y1,
                "x2": x2,
                "y2": y2
            },
            "dimensions": {
                "width": width,
                "height": height,
                "area": area
            }
        }
        defects.append(defect)

    #
    defect_summary = {}
    for defect in defects:
        defect_type = defect["type"]
        defect_summary[defect_type] = defect_summary.get(defect_type, 0) + 1

    #
    img = original_image.copy()
    for defect in defects:
        x1, y1 = int(defect["location"]["x1"]), int(defect["location"]["y1"])
        x2, y2 = int(defect["location"]["x2"]), int(defect["location"]["y2"])
        defect_type = defect["type"]
        conf = defect["confidence"]

        #
        if defect_type == "crack":
            color = (0, 0, 255)  #
        elif defect_type == "corrosion":
            color = (0, 165, 255)  #
        elif defect_type == "exposed_rebar":
            color = (0, 255, 255)  #
        else:
            color = (255, 0, 0)  #

        #
        cv2.rectangle(img, (x1, y1), (x2, y2), color, 2)
```

```
            #
            label = f"{defect_type} {conf:.2f}"
            (w, h), _ = cv2.getTextSize(label, cv2.FONT_HERSHEY_SIMPLEX, 0.5, 1)
            cv2.rectangle(img, (x1, y1 - 20), (x1 + w, y1), color, -1)
            cv2.putText(img, label, (x1, y1 - 5), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,
255, 255), 1)

        #                    Base64
        _, buffer = cv2.imencode('.jpg', img)
        img_base64 = base64.b64encode(buffer).decode('utf-8')

        #
        results_dict = {
            "total_defects": len(defects),
            "defect_summary": defect_summary,
            "defects": defects,
            "image": f"{img_base64}"
        }

        return results_dict

#
if __name__ == "__main__":
    #
    detector = DefectDetector(
        model_path="models/trained/weights/best.pt",
        conf_threshold=0.25,
        img_size=640,
        device='0'
    )

    #
    image_path = "data/test/images/test_image.jpg"
    results = detector.detect_from_file(image_path)

    print(f"              {results['total_defects']}      :")
    for defect_type, count in results['defect_summary'].items():
        print(f"- {defect_type}: {count}")
```

8.

:

```
 // frontend/src/services/analysis.js
import axios from 'axios';

const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:8000';
```

```
export const analyzeImage = async (file, options = {}) => {
  const { confThreshold = 0.25, imgSize = 640 } = options;

  const formData = new FormData();
  formData.append('file', file);
  formData.append('conf_threshold', confThreshold);
  formData.append('img_size', imgSize);

  try {
    const response = await axios.post(`${API_URL}/detect`, formData, {
      headers: {
        'Content-Type': 'multipart/form-data'
      }
    });

    return response.data;
  } catch (error) {
    console.error('Error analyzing image:', error);
    throw error;
  }
};

export const getHealthStatus = async () => {
  try {
    const response = await axios.get(`${API_URL}/health`);
    return response.data;
  } catch (error) {
    console.error('Error checking API health:', error);
    throw error;
  }
};
```

9.

:

```
 // frontend/src/pages/Analysis.js (        )
import React, { useState, useEffect } from 'react';
import { /* ...                          ... */} from '@mui/material';
import { /* ...                          ... */} from '@mui/icons-material';
import { useTranslation } from 'react-i18next';
import { useDropzone } from 'react-dropzone';
import { analyzeImage, getHealthStatus } from '../services/analysis';

const Analysis = () => {
  const { t } = useTranslation();
  const [files, setFiles] = useState([]);
  const [isAnalyzing, setIsAnalyzing] = useState(false);
```

```
const [analysisComplete, setAnalysisComplete] = useState(false);
const [analysisResults, setAnalysisResults] = useState(null);
const [error, setError] = useState('');
const [apiStatus, setApiStatus] = useState('unknown');

useEffect(() => {
 //
 const checkApiStatus = async () => {
  try {
   await getHealthStatus();
   setApiStatus('healthy');
  } catch (error) {
   setApiStatus('unhealthy');
   setError(t('analysis.apiError'));
  }
 };

 checkApiStatus();
}, [t]);

const onDrop = acceptedFiles => {
 setFiles(acceptedFiles.map(file => Object.assign(file, {
  preview: URL.createObjectURL(file)
 })));
};

const { getRootProps, getInputProps, isDragActive } = useDropzone({
 onDrop,
 accept: {
  'image/*': ['.jpeg', '.jpg', '.png'],
 },
 maxFiles: 1
});

const handleAnalyze = async () => {
 if (files.length === 0) {
  setError(t('analysis.noFilesError'));
  return;
 }

 if (apiStatus !== 'healthy') {
  setError(t('analysis.apiError'));
  return;
 }

 setError('');
 setIsAnalyzing(true);

 try {
  const result = await analyzeImage(files[0], {
   confThreshold: 0.25,
   imgSize: 640
```

```
      });

      setAnalysisResults(result);
      setIsAnalyzing(false);
      setAnalysisComplete(true);
    } catch (error) {
      setError(t('analysis.processingError'));
      setIsAnalyzing(false);
    }
  };

  const clearAnalysis = () => {
    setFiles([]);
    setAnalysisComplete(false);
    setAnalysisResults(null);
    setError('');

    // URL
    files.forEach(file => URL.revokeObjectURL(file.preview));
  };

  const getSeverityColor = (confidence) => {
    if (confidence > 0.8) {
      return '#f44336'; //  -
    } else if (confidence > 0.5) {
      return '#ff9800'; //  -
    } else {
      return '#4caf50'; //  -
    }
  };

  const getSeverityLevel = (confidence) => {
    if (confidence > 0.8) {
      return t('severity.high');
    } else if (confidence > 0.5) {
      return t('severity.medium');
    } else {
      return t('severity.low');
    }
  };

  return (
    <Container maxWidth="lg" sx={{ mt: 4, mb: 4 }}>
      <Typography variant="h4" component="h1" gutterBottom>
        {t('analysis.title')}
      </Typography>

      {error && (
        <Alert severity="error" sx={{ mb: 3 }}>
          {error}
        </Alert>
      )}
```

```
{apiStatus === 'unhealthy' && (
  <Alert severity="warning" sx={{ mb: 3 }}>
    {t('analysis.apiWarning')}
  </Alert>
)}

{!analysisComplete ? (
  <Paper sx={{ p: 3, mb: 4 }}>
    <Box
      {...getRootProps()}
      sx={{
        border: '2px dashed #cccccc',
        borderRadius: 2,
        p: 3,
        textAlign: 'center',
        bgcolor: isDragActive ? '#f0f8ff' : 'background.paper',
        cursor: 'pointer'
      }}
    >
      <input {...getInputProps()} />
      <CloudUploadIcon sx={{ fontSize: 48, color: 'primary.main', mb: 2 }} />
      <Typography variant="h6" gutterBottom>
        {isDragActive
          ? t('analysis.dropzone.active')
          : t('analysis.dropzone.inactive')}
      </Typography>
      <Typography variant="body2" color="textSecondary">
        {t('analysis.dropzone.hint')}
      </Typography>
    </Box>

    {files.length > 0 && (
      <Box sx={{ mt: 3 }}>
        <Typography variant="h6" gutterBottom>
          {t('analysis.selectedFiles')}
        </Typography>
        <Grid container spacing={2}>
          {files.map((file, index) => (
            <Grid item xs={12} sm={6} md={4} key={index}>
              <Card>
                <CardMedia
                  component="img"
                  height="200"
                  image={file.preview}
                  alt={file.name}
                />
                <CardContent sx={{ py: 1 }}>
                  <Typography variant="body2" noWrap>
                    {file.name}
                  </Typography>
                  <Typography variant="caption" color="textSecondary">
```

```
                {(file.size / 1024 / 1024).toFixed(2)} MB
              </Typography>
            </CardContent>
          </Card>
        </Grid>
      ))}
    </Grid>

    <Box sx={{ mt: 3, display: 'flex', justifyContent: 'center' }}>
      <Button
        variant="contained"
        color="primary"
        size="large"
        onClick={handleAnalyze}
        disabled={isAnalyzing || apiStatus !== 'healthy'}
        startIcon={isAnalyzing ? <CircularProgress size={24} color="inherit" /> :
<BugReportIcon />}
        sx={{ mr: 2 }}
      >
        {isAnalyzing ? t('analysis.analyzing') : t('analysis.analyze')}
      </Button>
      <Button
        variant="outlined"
        color="secondary"
        size="large"
        onClick={clearAnalysis}
        disabled={isAnalyzing}
      >
        {t('analysis.clear')}
      </Button>
    </Box>
  </Box>
)}
</Paper>
) : (
<Box>
  <Alert severity="success" sx={{ mb: 3 }}>
    {t('analysis.analysisComplete')}
  </Alert>

  <Paper sx={{ p: 3, mb: 4 }}>
    <Typography variant="h5" gutterBottom>
      {t('analysis.results.summary')}
    </Typography>

    <Grid container spacing={3} sx={{ mb: 3 }}>
      <Grid item xs={12} sm={4}>
        <Paper
          sx={{
            p: 2,
            textAlign: 'center',
            bgcolor: '#e3f2fd'
```

```
            }}
          >
            <Typography variant="h6" gutterBottom>
              {t('analysis.results.totalDefects')}
            </Typography>
            <Typography variant="h3">
              {analysisResults.total_defects}
            </Typography>
          </Paper>
        </Grid>
        <Grid item xs={12} sm={8}>
          <Paper sx={{ p: 2, height: '100%' }}>
            <Typography variant="h6" gutterBottom>
              {t('analysis.results.defectTypes')}
            </Typography>
            <Box sx={{ display: 'flex', flexWrap: 'wrap', gap: 1 }}>
              {Object.entries(analysisResults.defect_summary).map(([type, count]) =>
(
                <Chip
                  key={type}
                  label={`${t(`defects.${type}`)}: ${count}`}
                  color={
                    type === 'crack' ? 'error' :
                    type === 'corrosion' ? 'warning' :
                    type === 'exposed_rebar' ? 'secondary' : 'primary'
                  }
                />
              ))}
            </Box>
          </Paper>
        </Grid>
      </Grid>

      <Divider sx={{ mb: 3 }} />

      <Typography variant="h5" gutterBottom>
        {t('analysis.results.detailedResults')}
      </Typography>

      <Grid container spacing={3}>
        <Grid item xs={12} md={6}>
          <Card>
            <CardMedia
              component="img"
              height="400"
              image={analysisResults.image}
              alt="Analyzed image"
            />
          </Card>
        </Grid>
        <Grid item xs={12} md={6}>
          <Paper sx={{ p: 2, height: '100%', maxHeight: 400, overflow: 'auto' }}>
```

```
            <Typography variant="h6" gutterBottom>
              {t('analysis.results.defectList')}
            </Typography>
            <List>
              {analysisResults.defects.map((defect, index) => (
                <ListItem key={index} divider>
                  <ListItemIcon>
                    {defect.confidence > 0.8 ? (
                      <WarningIcon sx={{ color: getSeverityColor(defect.confidence) }} />
                    ) : (
                      <CheckCircleIcon sx={{ color:
getSeverityColor(defect.confidence) }} />
                    )}
                  </ListItemIcon>
                  <ListItemText
                    primary={`${t(`defects.${defect.type}`)}`}
                    secondary={`${t('analysis.results.confidence')}: ${(defect.confidence
* 100).toFixed(0)}%`}
                  />
                  <Chip
                    label={getSeverityLevel(defect.confidence)}
                    sx={{
                      bgcolor: getSeverityColor(defect.confidence),
                      color: 'white'
                    }}
                    size="small"
                  />
                </ListItem>
              ))}
            </List>
          </Paper>
        </Grid>
      </Grid>

      <Box sx={{ mt: 3, display: 'flex', justifyContent: 'center' }}>
        <Button
          variant="contained"
          color="primary"
          size="large"
          sx={{ mr: 2 }}
        >
          {t('analysis.results.generateReport')}
        </Button>
        <Button
          variant="outlined"
          color="secondary"
          size="large"
          onClick={clearAnalysis}
        >
          {t('analysis.newAnalysis')}
        </Button>
      </Box>
```

```
        </Paper>
      </Box>
    ))}
  </Container>
 );
};

export default Analysis;
```

10.

:

```
// frontend/public/locales/ar/translation.json (       )
{
 "analysis": {
  "apiError": "                              .                                    ,"
  "apiWarning": "                              .
    .",
  "processingError": "                              .                                    ."
 },
 "defects": {
  "crack": "       ",
  "corrosion": "       ",
  "exposed_rebar": "                          "
 }
}
```

```
// frontend/public/locales/en/translation.json (       )
{
 "analysis": {
  "apiError": "Could not connect to the AI service. Please try again later.",
  "apiWarning": "The AI service is currently unavailable. Some features may not
work properly.",
  "processingError": "An error occurred while processing the image. Please try
again."
 },
 "defects": {
  "crack": "Crack",
  "corrosion": "Corrosion",
  "exposed_rebar": "Exposed Reinforcement Bars"
 }
}
```

11.

:

```
 #!/bin/bash
# run_api.sh

#
export MODEL_PATH="models/trained/weights/best.pt"

#
cd app
uvicorn api:app --host 0.0.0.0 --port 8000 --reload
```

12.

( ) :

```
 #!/bin/bash
# run_app.sh

#
cd app
uvicorn api:app --host 0.0.0.0 --port 8000 &
API_PID=$!

#
cd ../frontend
npm start

#
trap "kill $API_PID" EXIT
```

YOLOv8.

:

1. :

.

2. : YOLOv8 .
3. : .
4. : .

5. : FastAPI
.
6. : .
7. : .

- : (Cracks) -
(Corrosion) - (Exposed Reinforcement Bars)

.