# Assessment Solution: Object Detection Microservice

This document details the development process and technical decisions made to create a containerized, real-time object detection web service, as per the technical assessment requirements.

## 1. Initial Interpretation and Strategy

The initial request outlined a microservice architecture with two distinct components: a UI backend and an AI backend. While this is a valid pattern, I opted for a more modern, streamlined single-service architecture for this specific problem.

### Justification for a Single-Service Model:

- **Reduced Complexity:** For an application with a single primary function (upload image -> get detection), a dual microservice approach introduces unnecessary network overhead and deployment complexity.
- **Efficiency:** A single FastAPI microservice can handle both serving the user interface and running the AI model efficiently, especially with Python's modern asynchronous capabilities.
- **Maintainability:** A single codebase is simpler to manage, debug, and version control.

This strategic decision allowed for a focus on creating a high-quality, robust application rather than managing inter-service communication.

## 2. Technology Selection

The following technologies were chosen to build the solution:

- ### Backend Framework: FastAPI
    - **Reasoning:** While Flask was an option, FastAPI was chosen for its superior performance, asynchronous support (ideal for I/O operations like file uploads), automatic interactive API documentation (via Swagger UI), and data validation using Python type hints.
    - *Reference: FastAPI Official Documentation*
- ### AI Model: YOLOv11 (from Ultralytics)
    - **Reasoning:** The assessment suggested YOLOv3. However, I chose the state-of-the-art YOLOv11 model. The ultralytics Python package provides a remarkably simple interface for loading models and performing inference, abstracting away complex boilerplate code. YOLOv11 also offers a better balance of speed and accuracy than its predecessors. The yolo11n.pt (nano) version is lightweight and perfect for running on a CPU also.
    - *Reference: Ultralytics Documentation*
- ### Containerization: Docker
    - **Reasoning:** As required by the assessment, Docker was used to containerize the application. This ensures that the application runs in a consistent, isolated

environment with all its dependencies, solving the classic "it works on my machine" problem and making deployment trivial.

# 3. Step-by-Step Development Process

The solution was built iteratively, focusing on adding and refining features in logical steps.

## Step 1: Core Object Detection Logic

The first step was to create a Python function that could take an image and return structured data. Using the ultralytics library, this was straightforward. The core logic involves loading the yolo11n.pt model and calling it on an input image. The result object conveniently contains all necessary information like class names, confidence scores, and bounding box coordinates, including the logic for dynamic scaling of visual annotations.

## Step 2: Building the Web Interface and API

With the detection function in place, the FastAPI application was created. An HTML file (templates/index.html) was designed to provide a minimal but functional UI, and JavaScript's fetch API was used to create an asynchronous request to the backend. A FastAPI endpoint (/predict/) was established to receive the uploaded image file and return the results.

## Step 3: Implementing File Logging

To meet the deliverable of providing output files, the backend was enhanced to save the results of every prediction. Using Python's os and json libraries, the application now automatically saves the input image, the annotated output image, and the JSON data into three separate folders (inputs/, outputs/, results/).

## Step 4: Containerization with Docker

The final step was to create a Dockerfile to package the application. A python:3.10-slim image was chosen, and after some iterative debugging to resolve missing shared libraries for OpenCV (libGL.so.1 and libgthread-2.0.so.0), a robust and reliable Dockerfile was finalized.

# 4. Scalability and Production Considerations

This solution provides a robust, self-contained service that successfully meets all the assessment's requirements. However, in a real-world production environment, I would typically implement several additional layers to ensure high availability, scalability, and resilience.

Drawing from my experience in deploying large-scale solutions, the next steps would involve:

- **Horizontal Scaling & Load Balancing:** The application is stateless, making it trivial to scale horizontally. One could deploy multiple container instances and place them behind a load balancer like **Nginx** or a cloud-native equivalent (e.g., AWS ALB). This would distribute incoming traffic, prevent any single instance from becoming a bottleneck, and ensure service availability even if one container fails.
- **Automated Port Allocation & Service Discovery:** For multi-instance deployments on a single host or cluster, a script could be developed to automatically detect and assign open ports. In

a more advanced setup using orchestration tools like Kubernetes, service discovery and networking are handled automatically.

- **GPU-Accelerated Inference:** While this implementation is optimized for CPU, a production deployment would leverage GPU resources for significantly faster inference. The Dockerfile would be adapted to use a base image with CUDA support, and the application logic would ensure the model is loaded onto the GPU. This can be scaled to **multi-GPU servers** for parallel processing of large batches of requests.

These advanced deployment strategies were intentionally omitted to keep the project scope aligned with the simplicity of the assignment. The current containerized solution, however, serves as the perfect foundational building block for such a scaled-up, production-grade system.

# 5. Final Instructions for Replication

To replicate this solution, please follow the steps outlined below.

## Build the Docker Image

Navigate to the project directory containing the Dockerfile and run the build command:

docker build -t yolo-fastapi-service .

## Run the Docker Container

Once the image is built, run the following command to start the application. The -p 8000:8000 flag maps your local port to the container's port.

docker run -p 8000:8000 yolo-fastapi-service