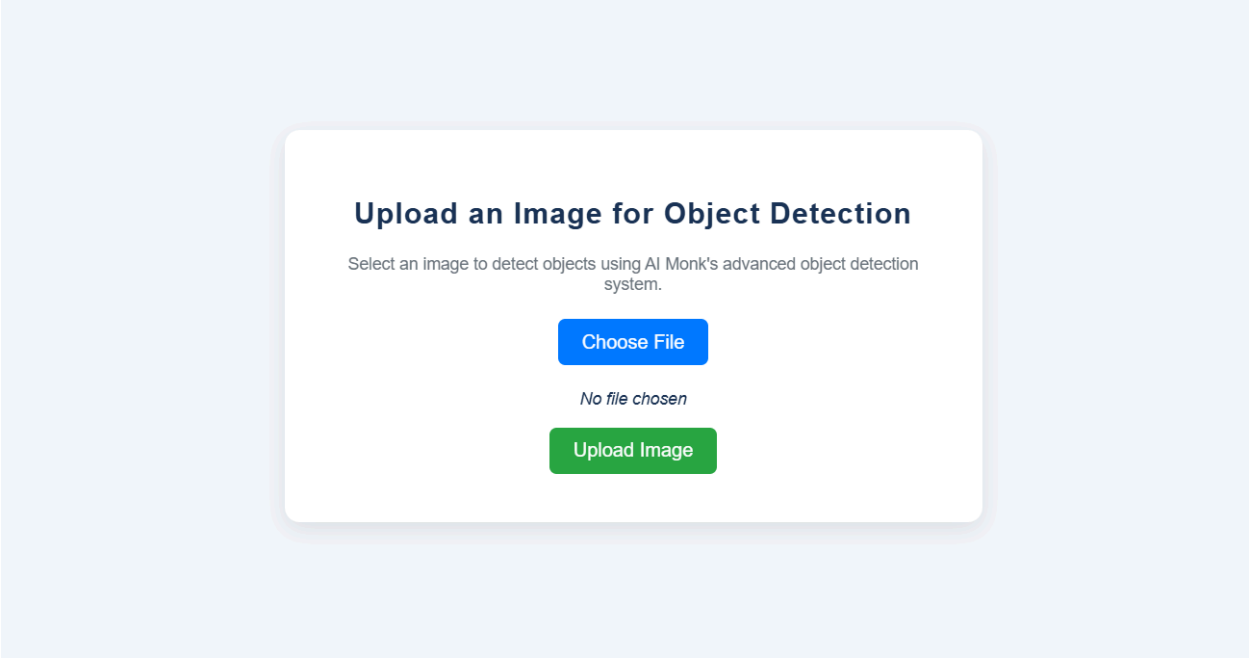
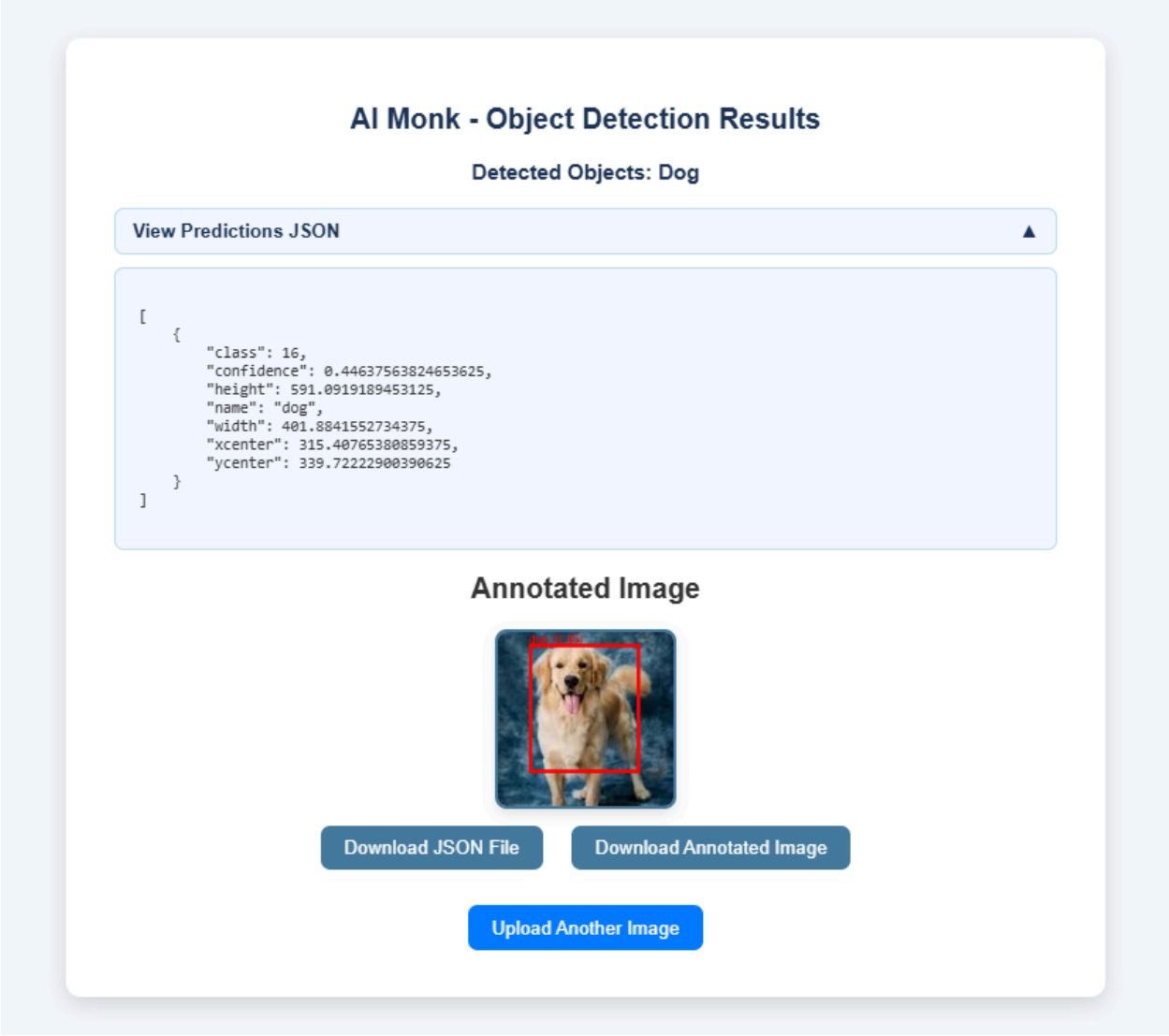
RADHA KRISHNA GARG
GITHUB REPO- https://github.com/sinisterdaddy/AIMONK

Have added all the functionalities as needed.

YOU CAN TEST IT ON -
https://aimonk-zg9c.onrender.com/
Here you can directly select the image, upload it and see the results, you can see the detected objects, predicted json and download those as well.





Also the AI Backend is hosted on AWS
You can test using the endpoint-
http://13.233.159.36:8080/predict

Project Overview

The objective of this project was to develop a microservice consisting of two primary components: a UI backend service and an AI backend service. The UI backend is responsible for handling user inputs, such as image uploads, while the AI backend utilizes a lightweight object detection model to process those images and return detected objects as a structured JSON response.

In this implementation, I have utilized **Flask** as the backend framework and **YOLOv5** as the object detection model. The entire solution is containerized using **Docker** to ensure it can be easily replicated across different environments.

This document outlines the steps taken to implement the solution, the architecture design, and how the various components communicate.

Architecture Overview

1. UI Backend Service (Flask)

- **Purpose**: The UI service accepts image files uploaded by the user. It then sends these images to the AI backend for processing.
- **Technology**: Flask (Python framework)
- **Functionality**:
    - Accepts HTTP POST requests containing image data.
    - Returns processed images with object detection bounding boxes and a JSON response with the detection results.

2. AI Backend Service (YOLOv5)

- **Purpose**: The AI service runs the YOLOv5 object detection model, processes the image, and generates predictions, including bounding box coordinates, class labels, and confidence scores.
- **Technology**: YOLOv5 (Pre-trained model from Ultralytics, available at: https://github.com/ultralytics/yolov5)
- **Functionality**:
    - Loads the pre-trained YOLOv5 model.
    - Runs inference on the uploaded image.
    - Returns the results (bounding boxes, class labels, confidence scores) in a structured JSON format.

3. Communication between Services

- The UI backend communicates with the AI backend via HTTP requests. When a user uploads an image, the UI backend forwards the image to the AI backend, which processes it and returns the results. The UI service then displays the processed image (with bounding boxes) along with the JSON response to the user.

4. Containerization (Docker)

- The entire application (both the UI backend and AI backend services) is containerized using Docker to provide an isolated and reproducible environment.
- This allows easy deployment of the application without worrying about dependencies and environment configurations.

Approach

Step 1: Setting up the Flask UI Backend

1. **Initial Setup**:
    - Installed Flask via `pip install flask`.
    - Created a basic Flask app to handle user requests for image uploads.
2. **Image Upload Handling**:
    - The application exposes an endpoint (`/upload`) to accept image files via HTTP POST requests.
    - The uploaded image is saved temporarily on the server for further processing.
3. **Sending Image to AI Backend**:
    - The Flask app sends the uploaded image as a POST request to the AI backend (using HTTP).
    - The image is passed in the body of the request, which is received by the AI service for inference.
4. **Returning Results**:
    - The results from the AI service (bounding boxes, class labels, confidence scores) are returned to the frontend as a JSON response.
    - Additionally, the processed image (with bounding boxes) is returned to the user.
5. **Frontend Rendering**:
    - The image is displayed with bounding boxes drawn over the detected objects.
    - The detection results (in JSON format) are displayed below or beside the image.

Step 2: Setting up the YOLOv5 AI Backend

1. **Installing Dependencies**:
    - Installed necessary dependencies for YOLOv5 (e.g., PyTorch, OpenCV, and other required libraries).
    - YOLOv5 was cloned from Ultralytics GitHub repo.
2. **Loading the YOLOv5 Model**:
    - Used the pre-trained YOLOv5 model (`yolov5s.pt` or `yolov5m.pt` depending on the model size).
    - The model was loaded into memory using PyTorch (`torch.load`).

3. **Inference**:
    - The image passed by the UI backend is preprocessed (resized, normalized, etc.) to match the input size expected by YOLOv5.
    - Inference is performed on the image using the YOLOv5 `detect()` method, which outputs a tensor containing bounding boxes, class labels, and confidence scores.
4. **Post-processing**:
    - The results are filtered to include only detections with a confidence score above a certain threshold (e.g., 0.5).
    - The bounding boxes are transformed back to the original image dimensions (if needed).
5. **Formatting the Response**:
    - The results are packaged into a structured JSON response, including the class labels, bounding box coordinates, and confidence scores for each detected object.
    - This response is sent back to the UI backend for rendering.

Step 3: Containerization with Docker

1. **Creating the Dockerfile**:
    - Created a `Dockerfile` to containerize the Flask UI service and AI backend service.
    - The Dockerfile installs dependencies, copies the necessary files, and sets up the environment for running both services.
2. **Building the Docker Image**:
    - Built Docker images for both services (UI backend and AI backend) using `docker build`.
3. **Docker Compose**:
    - Used `docker-compose.yml` to define and manage the multi-container setup, allowing both the UI and AI services to run in isolated containers.
    - Configured the services to communicate over a local network, ensuring that the UI backend can reach the AI backend using internal Docker networking.
4. **Running the Application**:
    - Ran the application using `docker-compose up`, which starts both the UI and AI backend services in parallel.
    - Verified the application by testing it in a browser, uploading images, and ensuring that the predictions and bounding boxes were displayed correctly.

Deliverables

1. **Project Folder**:
    - The project folder contains the following:
        - The full implementation of the Flask UI and AI backend services.
        - Docker configuration files (Dockerfile and `docker-compose.yml`).
        - Any necessary dependency files (e.g., `requirements.txt`).
2. **Documentation**:
    - This document outlines the approach taken to develop the solution, including steps, technologies used, and decisions made during development.
3. **Processed Images and JSON Files**:
    - Example processed images with object detection bounding boxes applied.
    - Corresponding JSON files with detection results (bounding box coordinates, class labels, confidence scores).
4. **Screenshots**:
    - Screenshots of the frontend UI displaying the image with bounding boxes and the JSON output from the object detection service.

Conclusion

This microservice-based solution allows users to upload images and receive object detection results through an intuitive web interface. By using Flask for the UI backend and YOLOv5 for the AI backend, the system efficiently handles image processing and displays results in real-time. The use of Docker ensures that the application is easy to deploy and replicate across different environments.