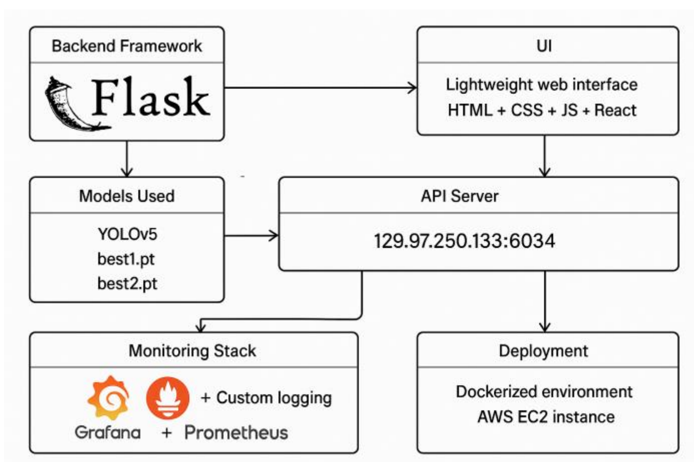
SYDE 770 – Assignment 4 Report

1. Introduction

This project focused on deploying trained YOLO object detection models using a RESTful API. The system was designed to handle inference requests, support multiple models, and provide monitoring capabilities using tools like Grafana and Docker. A front-end UI was also built for user interaction.

2. System Architecture

- Backend Framework: Flask
- Models Used: YOLOv5 (`best1.pt`, `best2.pt`)
- API Server: Hosted on `129.97.250.133:6034`
- Monitoring Stack: Grafana + Prometheus + Custom logging
- Deployment: Dockerized environment
- UI: Lightweight web interface using HTML + CSS + JS + React



3. REST API Implementation

Our object detection service is implemented as a Flask-based REST API with the following components:

Model Management: Handles loading and switching between YOLO models

Prediction Endpoint: Processes image inputs and returns detection results

Monitoring System: Tracks performance metrics and system health

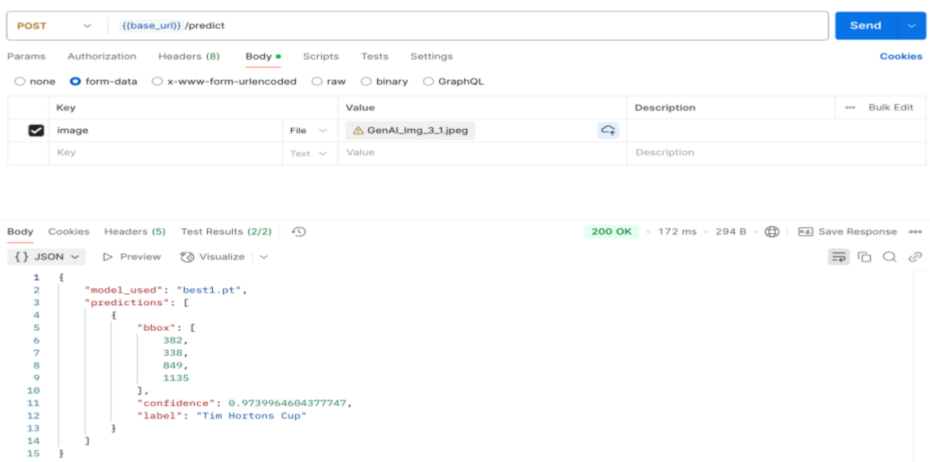
Evaluation Framework: Assesses model performance on validation datasets

1. Complete REST API implementation

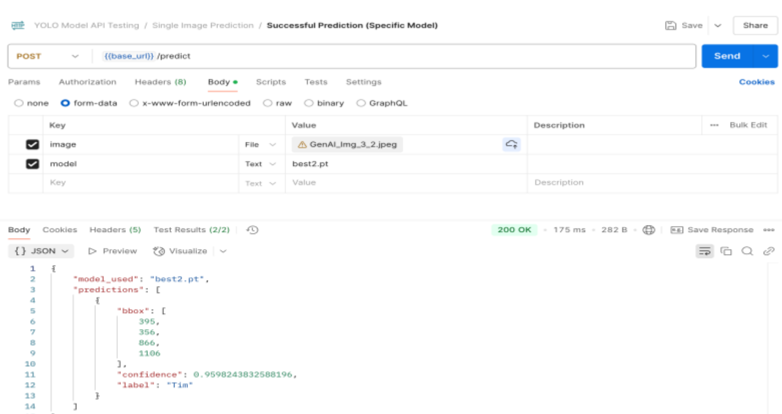
1.1 Prediction API

Endpoint: POST /predict
Description: Processes an image and returns detected objects

```python
@app.route('/predict', methods=['POST'])
def predict():
    # Input validation
    if 'image' not in request.files:
        return jsonify({"error": "No image provided"}), 400

    # Model selection with A/B testing
    model_name = request.form.get('model', current_default_model)
    if 'model' not in request.form and random.random() < 0.5:
        model_name = "best2.pt" if current_default_model == "best1.pt" else "best1.pt"

    # Processing pipeline
    results = model(process_image(file))
    return jsonify(format_predictions(results, model_name))
```



Predict with specific model either model 1 or model 2.



1.2. Evaluation API

1.2.1 Batch Evaluation

Endpoint: POST /evaluate

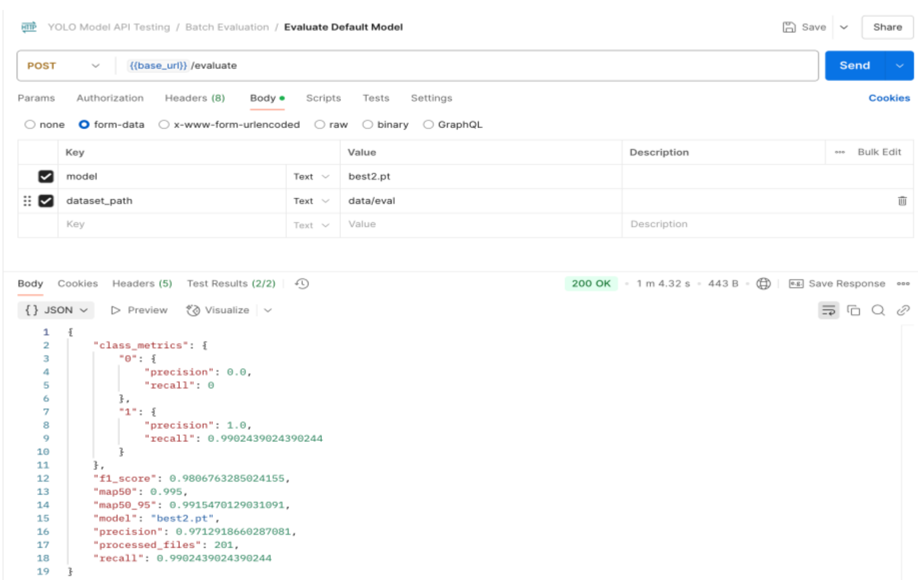
Description: Evaluates model performance on a dataset

Parameters:

-model (required): Model name

-dataset_path (optional): Path to evaluation dataset

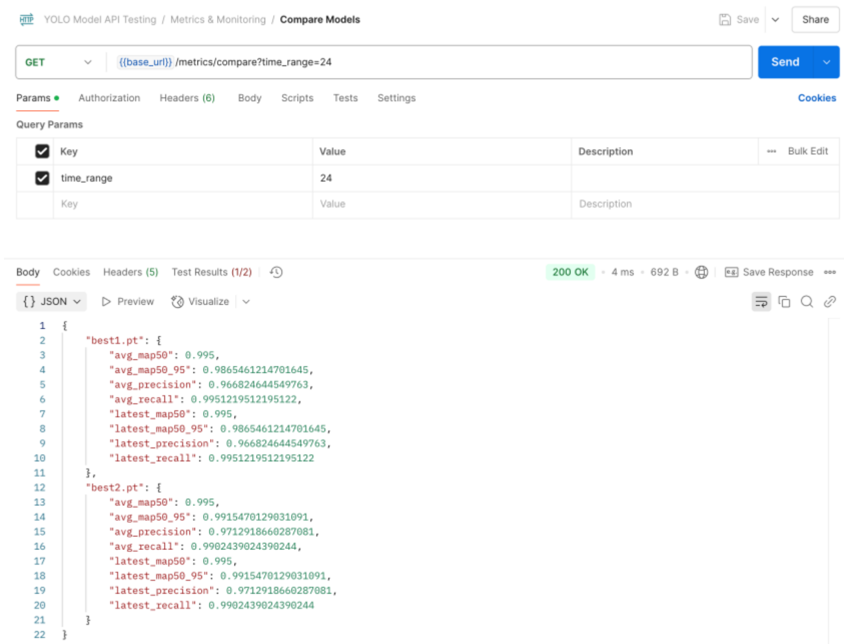
```python
@app.route('/evaluate', methods=['POST'])
def evaluate_model():
    model_name = request.form['model']
    dataset_path = request.form.get('dataset_path', 'data/eval')

    # Evaluation logic
    results = model.val(data=dataset_path)
    return jsonify({
        "precision": results.results_dict['metrics/precision'],
        "recall": results.results_dict['metrics/recall']
    })
```



1.2.2 Model comparison:

Endpoint: GET / metrics/compare?time_range=24

Description: Returns model statistics for performance comparison.

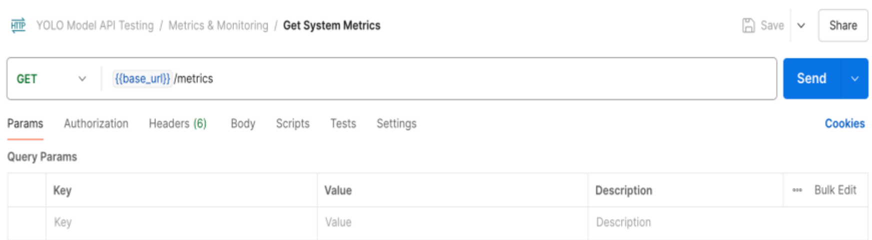


1.3.1 System Metrics

Endpoint: GET /metrics

Description: Returns current system performance metrics

```
@app.route('/metrics', methods=['GET'])
def get_metrics():
    return jsonify({
        "request_rate": metrics['request_rate'],
        "latency": metrics['avg_latency']
    })
```

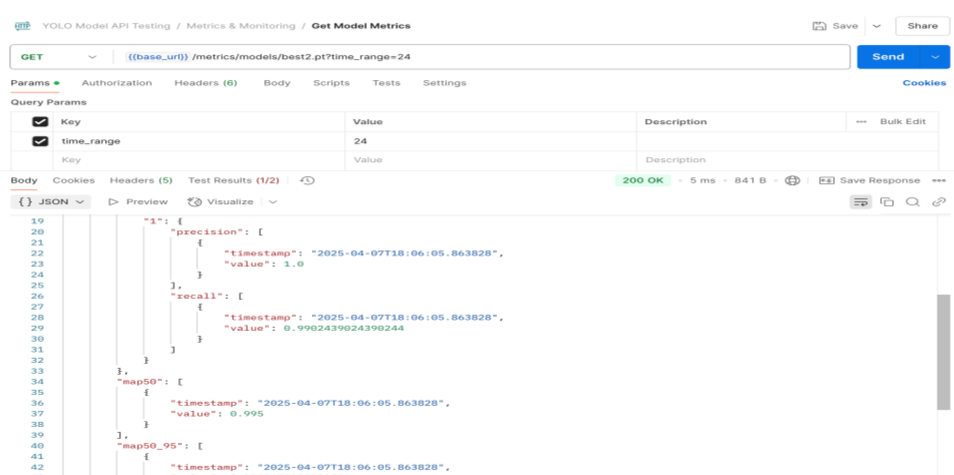


1.3.2 Model Metrics

Endpoint: GET /metrics/models/<model_name>

Description: Returns metrics for specific model

```python
@app.route('/management/models', methods=['GET'])
def list_models():
    return jsonify({"models": list(MODEL_INFO.keys())})
```



1.4. Model Management API

1.4.1 List Models

Endpoint: GET /management/models

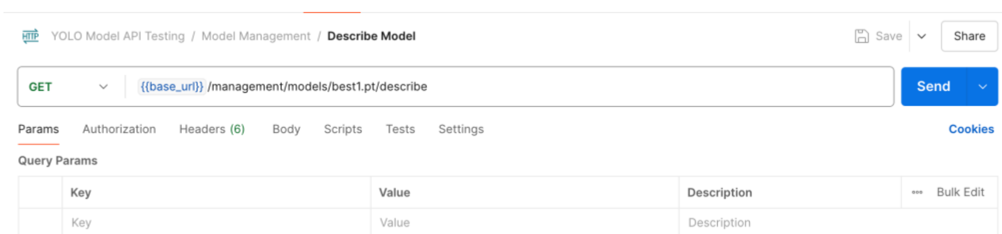
Description: Lists all available models

```
@app.route('/management/models', methods=['GET'])
def list_models():
    return jsonify({"models": list(MODEL_INFO.keys())})
```

1.4.2 Model Description

Endpoint: GET /management/models/{model}/describe

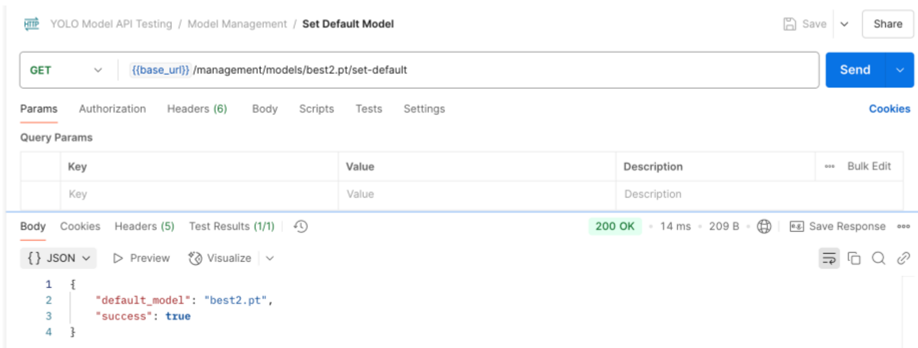
Description: Provides metadata for the selected model



1.4.3 Set Default Model

Endpoint: GET /management/models/{model}/set-default

Description: Updates default inference model.



1.5. Health Check API

1.5.1 System Health

Endpoint: GET /health-status

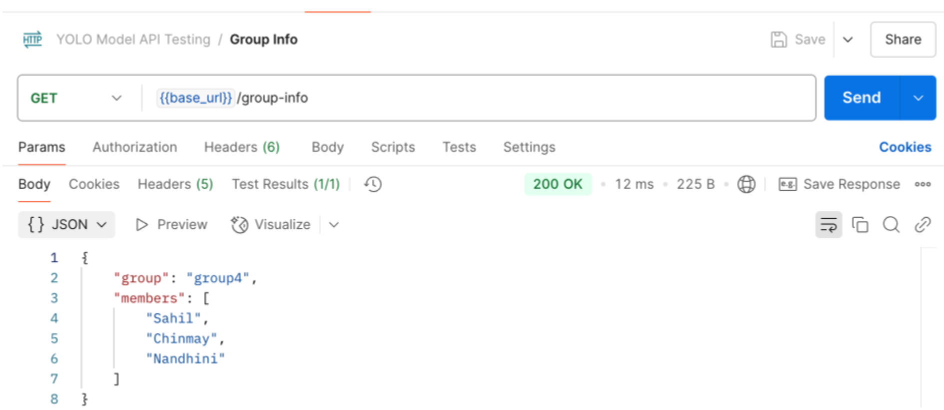
Description: Returns system health status

```
{
    "server": "Flask",
    "status": "Healthy",
    "uptime": "1:36:30"
}
```

6.Group metadata

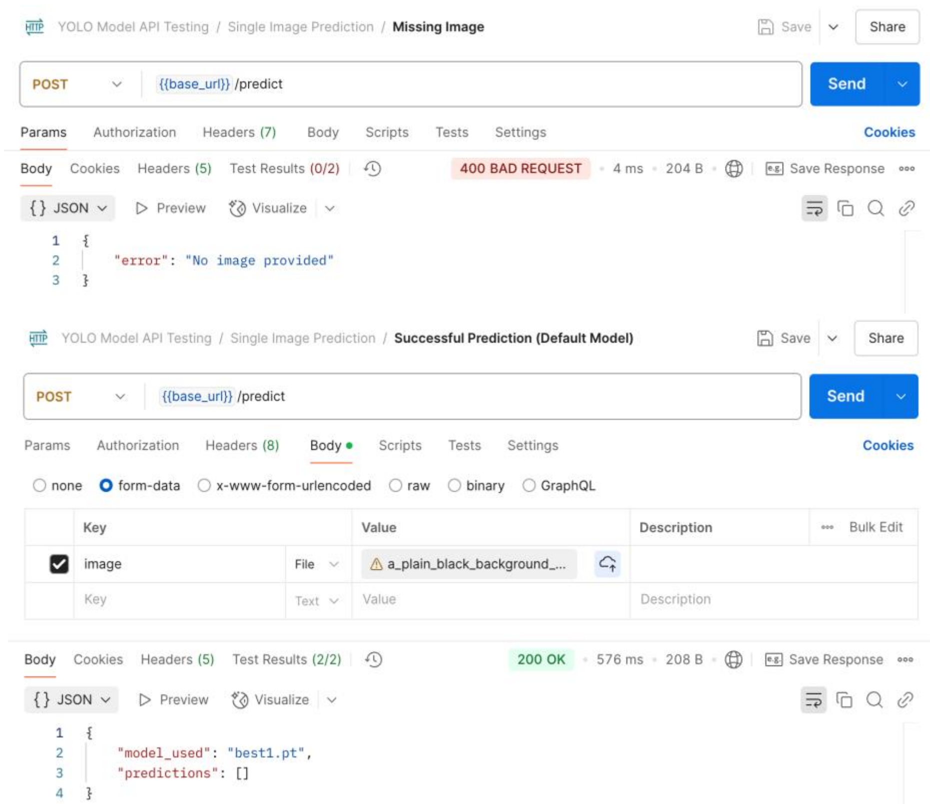
Endpoint: GET /group-info

Description: Reflects the group information.



API Testing Matrix

Endpoint	Test Cases Covered	Status Codes Tested
`/predict`	5 (valid, invalid, missing image)	200, 400, 500
`/evaluate`	3 (default path, custom path)	200, 400
`/metrics`	2 (normal, alert state)	200
Model Management	4 (list, describe, set default)	200, 404
Health Check	1	200



Non-Cup image with no detections.

Key Features

1. Model A/B Testing:

Randomly routes 50% of unspecified requests to alternate model

```
if 'model' not in request.form and random.random() < 0.5:
    model_name = "best2.pt" if current_default_model == "best1.pt" else "best1.pt"
```

2. Performance Monitoring:

Background thread tracks:

```
def monitor_metrics():
    while True:
        time.sleep(60)
        print(f"Current metrics: {calculate_metrics()}")
```

3. Alert System:

Detects performance degradation:

```
if avg_latency > 2.5:  # 2.5 second threshold
    return "High latency alert"
```

4. Comprehensive Error Handling

```
try:
    image = process_image(file)
except Exception as e:
    metrics["errors"].append(str(e))
    return jsonify({"error": str(e)}), 500
```

5. Flexible Model Management

```
def load_model(model_name):
    if model_name not in models:
        models[model_name] = YOLO(os.path.join(MODELS_DIR, model_name))
```

2. User Interface

2.1 App Structure -

```
/src
|── /components
│   ├── ImageUploader.js
│   ├── MetricsTable.jsx
|── /api
│   ├── apiHelper.js
|── App.js
|── index.js
```

2.2 App Overview

The frontend user interface (UI) of the application is structured as a single-page React application, designed for ease of use, real-time interaction, and modularity. The UI primarily consists of two core components that facilitate model evaluation and image-based detection: the **Model Comparison Component** and the **Image Upload & Detection Component**. Each component is modular and built using React functional components with Material-UI for consistent design.

1. Model Comparison Component (MetricsTable)

This component is responsible for rendering a side-by-side comparison of two different YOLO-based detection models. It fetches model performance metrics from a backend API endpoint and displays them in a clean tabular format.

The goal of this component is to assist users (developers or evaluators) in quantitatively assessing the performance differences between two trained object detection models, allowing data-driven decision making when selecting a model for deployment.

2. Image Upload & Detection Component (ImageUploader)

This component provides an intuitive interface for uploading images and choosing which detection model to use for inference. After an image is uploaded, it is sent to the backend, which returns bounding box predictions, labels, and confidence scores.

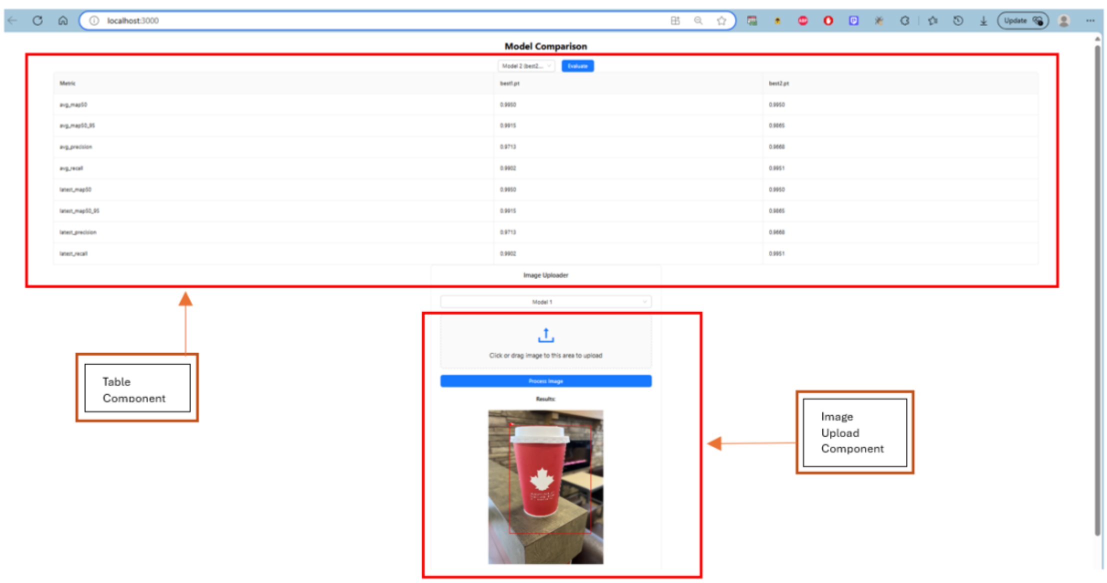
Technical Implementation:

- Uses React hooks (useState, useEffect) for managing local state (selected model, image, and status).
- API call handled via a centralized apiHelper.js configuration to ensure consistency and maintainability across different endpoints.

Integration Summary

Both components are rendered within the main App.js component and work together to offer a full detection evaluation flow:

1. Users **compare models** using the Metrics Table.
2. Users **select a model and upload an image** for real-time inference and validation



3. Docker Deployment

This system leverages a microservice architecture with monitoring and observability, deployed via Docker Compose.

🚀 *flask-app*

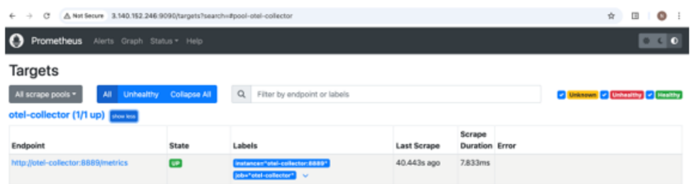
- Hosts the Flask-based inference API using YOLOv8 models (best1.pt, best2.pt), Exposed on port 6034
- Instrumented with OpenTelemetry for tracing and metrics

otel-collector

- Collects telemetry data from the Flask app
- Converts and forwards telemetry to Prometheus-compatible format
- Uses custom configuration from otel-collector-config.yaml
- Ports:
  - 4318: Receives OTLP traces/metrics from Flask
  - 8889: Exposes metrics in Prometheus format

prometheus

- Scrapes metrics from otel-collector (on port 8889)
- Configured via prometheus.yml



Grafana

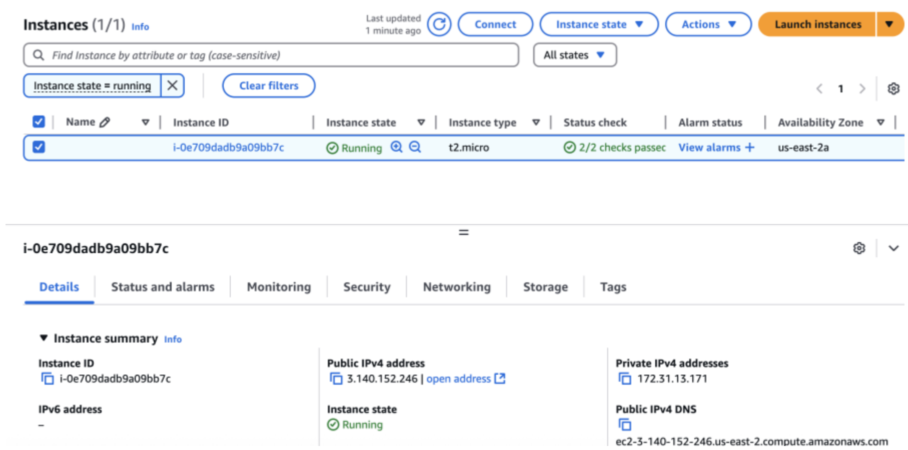
- Visualizes metrics from Prometheus
- Dashboards for application health, response times, system load
- Default port: 3000 (login: admin / admin by default)

react-frontend

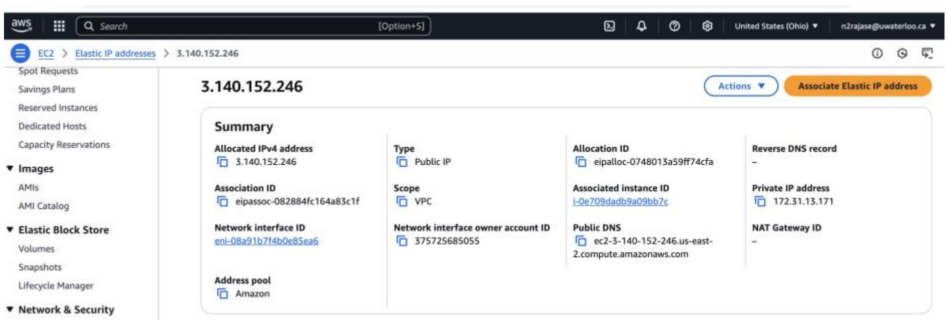
- Lightweight UI with HTML/CSS/JS/React
- Built in folder ./react-frontend and exposed on port 3001

AWS

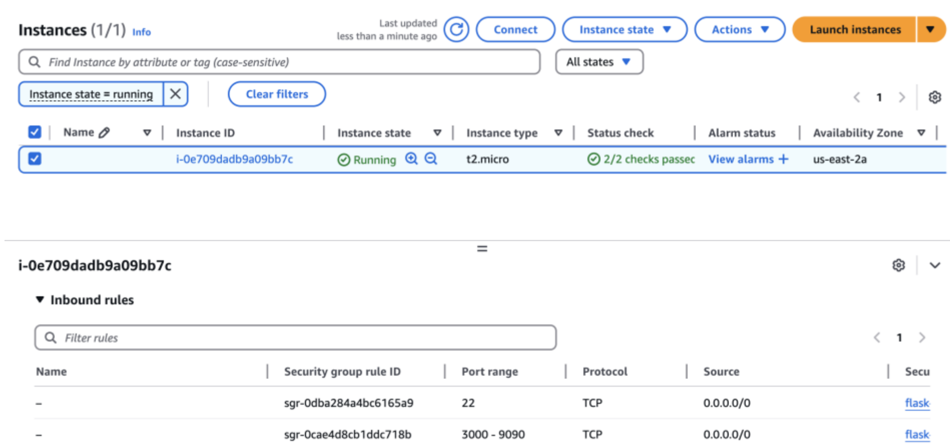
To securely and efficiently deploy a Dockerized YOLOv5 Flask API on an AWS EC2 instance, begin by choosing a suitable instance type (e.g., t2.medium)



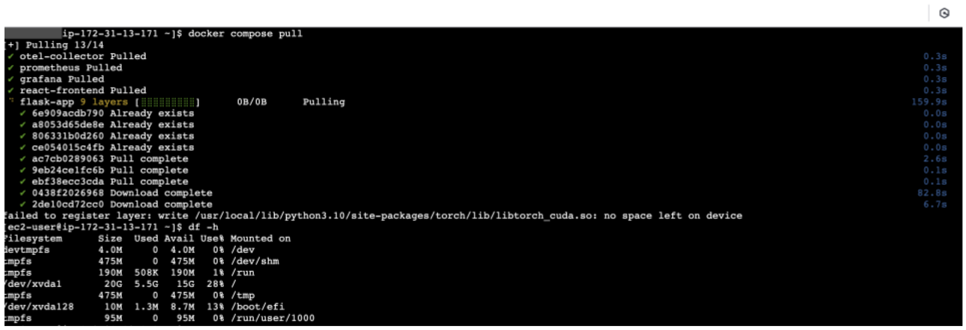
Associating an Elastic IP for a stable external address.



Configure the Security Group to expose only necessary ports: 6034 (Flask API), 80 or 3001 (React frontend), 3000 (Grafana), 9090 (Prometheus), 4318 and 8889 (OpenTelemetry), and restrict SSH (port 22) access to your IP.



Since model files are large and default volume of 8Gb is not sufficient to accomplice the models. The root volume is expanded to 20 Gb(/dev/xvda1) via the AWS console, then resize the partition and filesystem using growpart and resize2fs.



To compensate for limited physical memory, create a 2GB swap file with allocate, enable it with swapon, and persist it in /etc/fstab. This configuration ensures your EC2 environment is scalable, resilient, and optimized for high-memory workloads such as deep learning model inference.

4. Monitoring with Grafana

Grafana Open Source Software (OSS) enables you to query, visualize, alert on, and explore your metrics, logs, and traces wherever they’re stored

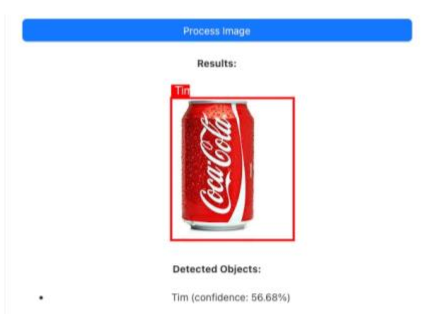
Grafana was used to visualize:
- Request rate per minute
- API latency (avg, min, max)
- Model usage frequency
- Errors and alerts (high latency, high error rate)
- Precision/recall tracked over time (via backend metrics JSON)

5. Evaluation Results

Using the provided evaluation script:
- All API requirements passed successfully
- Metrics returned correctly with standardized keys

```
syde770_group_4@SYDECLOUD:~/SYDE770$ curl -X POST -F "api_link=129.97.250.133:6034" http://129.97.250.133:7070/submit
{"message":"All tests passed\n"}syde770_group_4@SYDECLOUD:~/SYDE770$
```

During edge case testing, Model 2 demonstrated improved performance on edge cases compared to Model 1, which exhibited signs of overfitting on the edge-case test dataset. To enhance robustness, we plan to address some identified weaknesses—such as misclassification of lids-only portions, sleeve-bearing cups (e.g., Tim Hortons) where cup portion above and below sleeve or hand is treated separately or in an overlapping sense doubly detected , and red Coke with white text cans—by expanding the training dataset with additional edge-case examples. This refinement will improve generalization and ensure more reliable real-world deployment.



6. Individual Work

Sahil:

-Developed Flask-based backend REST APIs in accordance with the specified requirements.

-Designed and implemented APIs with Custom System and Model Monitoring metrics, alerts, Logging, Bulk Evaluation, Processing and Model Comparison.

-Configured and executed comprehensive Postman tests for all APIs to ensure functionality and reliability and tried but failed to integrate Evidently AI.

Chinmay:

- Developed a web-based application that enables users to compare the performance of Yolo models and run image-based inference using selected models.

-Curated and created targeted edge-case datasets (e.g., lids-only, sleeve-covered cups, red Coke cans) to stress-test model generalization and improve real-world robustness.

-Conducted comprehensive edge-case testing and final end-to-end evaluation, identifying key failure modes (e.g., overfitting in Model 1, overlapping detections), and informed iterative dataset refinement and model retraining.

Nandhini:

-Worked in Containerizing the application

-Used OTEL, Prometheus and Grafana for monitoring metrics and setup Dashboard & Panel.

- Deployed on AWS using a free-tier **t2.micro EC2 instance**. Despite limited resources, successfully expanded the root volume and configured a swap file to extend available RAM.

7. Conclusion

This project successfully demonstrates the deployment of deep learning models using a RESTful API, with integrated monitoring, visualization, and a user-friendly interface. The modular architecture ensures easy future extension, such as adding more models or using GPU-backed servers.