**CloudFactory Object Detection Pipeline:**

**Installation & Setup (GitHub)**

**Dependencies Installation**

# Create virtual environment

python -m venv venv

source venv/bin/activate # On Windows: venv\Scripts\activate

# Install dependencies

pip install -r requirements.txt

while using docker I had to take detectron2==0.6 out however in the docker I have wrote to pip install it should work however if it doesn’t need to pip install detectron2==0.6.

now pull the github repository file : https://github.com/Samyak44/ImageRecognition\_pipeline

**Important: Use numpy<2.0.0 to avoid compatibility issues with PyTorch.**

**Basic Execution**

# Run with default settings

python app.py

# Process images with custom confidence threshold

python app.py --confidence\_threshold 0.5

# Adjust NMS threshold for detection overlap

python app.py --nms\_threshold 0.3

# Enable debug logging

python app.py --debug

**To run according we to pass the parameters: this will create the output.**

python app.py --images\_dir ./Image --model\_path ./fish\_detector/model.pt --class\_mapping\_path ./fish\_detector/class\_mapping.json --output\_dir ./output

**When dockerized:**

docker pull mavrick444/fish-imagedetection:latest

Create the following folders on your local machine:

fish-detection-demo/

├── images/ # Put your fish images here (.jpg, .png)

├── models/ # Model files (see below)

│ ├── model.pth

│ └── class\_mapping.json

└── output/ # Results will appear here

**Running the pipeline:**

docker run -v $(pwd)/images:/app/images \ -v $(pwd)/models:/app/models \ -v $(pwd)/output:/app/output \ mavrick444/fish-imagedetection:latest \ python app.py \ --images\_dir /app/images \ --model\_path /app/models/model.pth \ --class\_mapping\_path /app/models/class\_mapping.json \ --output\_dir /app/output

**When you run the image :**

docker run image-detection python app.py \ --images\_dir /path/to/images \ --model\_path /path/to/model \ --class\_mapping\_path /path/to/class\_mapping \ --output\_dir /path/to/output

Fish-imagedDectector

This project is a Dockerized pipeline for detecting fish in images using PyTorch.

The pipeline reads images from the `Image/` folder, runs inference using a pre-trained model in `fish\_detector/`, and outputs results to the `output/` folder.

## Features

- Runs completely inside a Docker container.

- No installation of Python dependencies on the host required.

- Prepares outputs automatically in the `output/` folder.

## Known Dependency Warning

When running the pipeline, you may see a warning like:

**Failed to initialize NumPy: A module that was compiled using NumPy 1.x cannot be run in NumPy 2.x...**

This happens because:

- PyTorch was compiled with NumPy 1.x.

- Docker image currently has NumPy 2.2.6 (required for OpenCV 4.12+).

This is a warning only the pipeline has been tested and runs successfully. Outputs are correctly generated.

## Running the Docker Container

### Build Docker image (if needed):

```bash

docker build -t fish-imagedetector .

**Overview**

This project implements a production-ready ML pipeline for batch object detection using CloudFactory's TorchScript models. The pipeline processes images through the complete workflow from preprocessing to COCO format output generation.

**Solution Architecture**

**Core Components**

1. **Image Preprocessing Module** (CloudFactoryImageProcessor)
   * Loads and applies CloudFactory's serialized transforms
   * Handles image format conversions (RGBA → RGB)
   * Maintains preprocessing consistency between training and inference
2. **Object Detection Module** (CloudFactoryObjectDetector)
   * Supports both Faster R-CNN and FBNetv3 model formats
   * Implements CloudFactory's exact inference patterns
   * Applies Non-Maximum Suppression for duplicate removal
3. **Results Export Module** (COCOResultsExporter)
   * Generates industry-standard COCO format output
   * Includes detection metadata and confidence scores
   * Compatible with evaluation and visualization tools
4. **Pipeline Orchestrator** (MLPipelineRunner)
   * Coordinates all pipeline components
   * Handles batch processing and error recovery
   * Manages resource cleanup and logging

**Error Resilience**: The pipeline continues processing even if individual images fail, ensuring maximum throughput in production environments.

**Technical Implementation Details**

**Transform Handling**:

* Prioritizes CloudFactory's serialized transforms.json for consistency
* Falls back to standard 224x224 resize if transforms unavailable
* Handles various image formats (PNG, JPEG, RGBA) automatically

**Model Inference Pattern**:

* Follows CloudFactory's exact tensor preparation (channels-first, float conversion)
* Supports both dictionary and tuple output formats
* Implements proper batch dimension handling

**Memory Management**:

* GPU memory cleanup after processing
* Efficient tensor operations to minimize memory usage
* Resource cleanup in finally blocks

**File Structure**

project/

├── app.py # Main pipeline script

├── fish\_detector/

│ ├── model.pt # TorchScript model

│ ├── class\_mapping.json # Class definitions

│ └── transforms.json # Image transforms config

├── image/ # Input images directory

│ ├── image1.jpg

│ └── image2.jpg

├── output/ # Generated results

│ └── detection\_results\_TIMESTAMP.json

└── requirements.txt # Python dependencies

**Key Fields Explanation**

* **bbox**: [x, y, width, height] in pixels
* **score**: Model confidence (0.0 to 1.0)
* **area**: Bounding box area in square pixels
* **category\_id**: Maps to class in categories array

**Results Analysis**

Based on the provided detection results:

**Performance Summary**

* **Total Images Processed**: 5
* **Total Detections**: 12 fish detected
* **Detection Rate**: 80% (4 out of 5 images had detections)

**Confidence Distribution**

* **High Confidence (>80%)**: 3 detections
  + Best: 98.4% (great\_blue\_heron\_with\_fish)
  + Second: 93.9% (Koi\_fish\_in\_the\_water)
  + Third: 81.2% (Posidonia\_oceanica)
* **Medium Confidence (50-80%)**: 2 detections
* **Lower Confidence (<50%)**: 7 detections

**Image-by-Image Results**

1. **Koi\_fish\_in\_the\_water.jpg**: 5 fish (best: 93.9%)
2. **Madeira\_Fish.jpeg**: 5 fish (best: 61.6%)
3. **Malate\_Manila\_Aquarium\_Fish.jpg**: 0 fish (challenging image)
4. **Posidonia\_oceanica.jpg**: 1 fish (81.2%)
5. **great\_blue\_heron\_with\_fish.jpg**: 1 fish (98.4%)

**Testing**

**Running Unit Tests**

# Run all tests

python -m pytest test\_pipeline.py -v

# Run specific test class

python -m pytest test\_pipeline.py::TestCloudFactoryImageProcessor -v

# Run with coverage

python -m pytest test\_pipeline.py --cov=app --cov-report=html

**Test Coverage Areas**

* Image preprocessing with various formats
* Transform loading and application
* COCO format generation and validation
* Error handling for edge cases
* Mock model inference testing

**Troubleshooting**

**Common Issues**

**NumPy Compatibility Error**:

# Fix: Downgrade NumPy

pip install "numpy<2.0.0"

**Transform Loading Error**:

* Verify transforms.json format matches CloudFactory specification
* Check albumentations version compatibility
* Use default transforms if custom ones fail

**No Detections Found**:

* Lower confidence threshold: --confidence\_threshold 0.1
* Adjust NMS threshold: --nms\_threshold 0.3
* Verify image quality and class mapping

## Testing Methodology

**Real-world reasoning:**

1. **Unit testing** should focus on isolated components, not the entire end-to-end system.
   * Test preprocessing, inference, and postprocessing separately.
2. **Mocking inputs**: Instead of using large datasets, use small dummy images or a few sample fish images.
3. **Check outputs rigorously**:
   * Type (numpy.ndarray, list, dict)
   * Shape ((N, 4) for bounding boxes)
   * Values (confidence scores between 0 and 1, class labels within valid classes)

**Overall Design Philosophy**

The script implements a **modular ML pipeline**:

1. **Image preprocessing** (CloudFactoryImageProcessor)
2. **Model inference** (CloudFactoryObjectDetector)
3. **COCO-format output generation** (COCOResultsExporter)
4. **Infrastructure and resource management** (PipelineInfrastructure)
5. **Batch orchestration** (run\_pipeline + main CLI)

**Reasoning**:

* Modularity makes the pipeline **easier to test**, debug, and extend.
* Each class is **single-responsibility**, which is a best practice in software engineering.
* Using **debugging/logging in every function** ensures we can monitor each step and catch errors quickly.

**2. Image Preprocessing with Augmentations**

* **Why**:
  + Preprocessing must match the training transforms to maintain model performance.
  + Using albumentations ensures transformations are **deterministic, fast, and configurable** via transforms.json.
  + Default fallback (resize to 224×224) allows robustness if config is missing.
* **Benefit**:
  + Ensures that even new images or batch runs behave consistently.
  + Debuggers in process\_image let you **inspect preprocessed arrays**, catch shape or normalization errors early.

**3. Model Inference**

**Why**:

* Handles **both Faster RCNN and FBNetv3 formats**, so the pipeline is model-agnostic.
* Uses **TorchScript**, which is optimized for production and avoids Python-only dependency issues.
* Includes **NMS** (Non-Maximum Suppression) as per CloudFactory guidelines.

**Benefit**:

* Predictable output: boxes, scores, classes.
* GPU/CPU device handling allows flexible deployment.
* Debugging ensures that **tensor shapes, types, and NMS filtering** are correct.

**Coco-format output :**

**Why**:

* COCO is **industry-standard** for object detection evaluation.
* JSON structure is compatible with evaluation tools and visualization frameworks.
* Allows later integration with dashboards or model evaluation metrics.

**Benefit**:

* Easy to audit predictions.
* Enables reproducible results for reporting or competitions.
* Debug logs help verify that **each annotation** is valid (positive bbox, correct class ID).

**5. Infrastructure and Resource Management**

* **Why**:
  + GPU memory must be **cleared after batch processing** to avoid leaks.
  + Temporary files are cleaned automatically.
  + Keeps track of runtime statistics for performance monitoring.
* **Benefit**:
  + Ensures clean and safe execution in **automated pipelines** or CI/CD deployments.
  + Helps identify bottlenecks via logged total time.

**6. Batch Pipeline Orchestration**

* **Why**:
  + Central function that **orchestrates all steps**: preprocess → predict → save results.
  + Uses **try-except** around individual images, so one corrupt file does not crash the entire batch.
  + Generates **timestamped COCO outputs**, ensuring reproducibility.
* **Benefit**:
  + Handles large batches reliably.
  + Provides clear logging for **each step and each image**.
  + Supports optional transforms.json and configurable nms\_threshold for flexibility.

**7. Command-Line Interface (CLI)**

* **Why**:
  + Provides a **simple way to run the pipeline** without editing the code.
  + Validates file existence before processing.
  + Accepts arguments for batch images, model, class mappings, output directory, and NMS threshold.
* **Benefit**:
  + Usable by non-programmers or in **automation scripts**.
  + Debugging in each function ensures errors are caught **before the pipeline fails silently**.

**8. Why This is a Best Practice**

1. **Modularity** → Each step is isolated, testable, and debuggable.
2. **Logging and Debugging** → Every function has logging/debugger hooks to inspect **what is happening internally**.
3. **Error Handling** → Pipeline continues even if one image fails, preventing batch crashes.
4. **Compatibility & Flexibility** → TorchScript + NMS + COCO standard + transforms.json make it portable and reproducible.
5. **Production Ready** → Resource management, GPU cleanup, and CLI make it robust for deployment.
6. **Scalable & Maintainable** → Easy to add new models, transforms, or postprocessing steps.

**Summary**

Designed **like a production-grade ML pipeline**:

* **Preprocessing** ensures inputs match training.
* **Inference** supports multiple model outputs robustly.
* **Postprocessing** exports COCO for standardized evaluation.
* **Infrastructure** manages resources safely.
* **Debugging/logging** in every function allows **real-time monitoring** and fast issue resolution.

In short: this is a **scalable, maintainable, and testable approach** that aligns with industry best practices.