AI FAE CV Technical Assessment – Edge AI Video Analytics Pipeline
Candidate: Bekir Oruk
Position: AI Field Application Engineer – Computer Vision
Date: 12 December 2025

---

1. Introduction

This report describes the design and implementation of an end-to-end Edge AI Video Analytics system developed for the AI FAE CV Technical Assessment.

The goal of the project is to build a **production-style pipeline** that covers the full lifecycle of a computer vision model:

- Model training with modern object detection architectures
- Model optimization and export (PyTorch → ONNX → TensorRT)
- A multi-backend inference engine
- Real-time video analytics with detector + tracker fusion
- FastAPI + Docker deployment
- Performance monitoring and basic unit tests

All code is organized under the repository cv-advanced-assessment following the folder structure provided in the assessment brief.

---

2. Model Training

2.1 Dataset

For this assessment I used the **COCO8** mini dataset provided by Ultralytics:

- Train images: 4
- Validation images: 4
- Total instances: 17
- Number of classes: 8
    - person, dog, horse, elephant, umbrella, potted plant, etc.

Even though the dataset is intentionally small, it is sufficient to demonstrate the full training, evaluation, and logging pipeline.

2.2 Architecture and Training Setup

The model is based on **YOLOv8n** (Ultralytics) with the following configuration:

- Backbone / head: standard YOLOv8n detection architecture
- Input resolution: 640 × 640
- Device: CPU (Intel i7-12650H)

- Epochs: 10
- Batch size: 8
- Optimizer: AdamW (automatically selected by Ultralytics)
- Learning rate schedule: cosine LR
- EMA: enabled via Ultralytics Trainer
- Mixed precision (AMP): enabled at the framework level, but effectively runs in FP32 on CPU

The custom training script is implemented in:

- training/train.py – orchestrates training and logging
- training/dataset.yaml or coco8.yaml – dataset configuration
- training/augmentations.py – lists the augmentation strategy

2.3 Data Augmentation and Training Features

The training pipeline uses **strong augmentations** to mimic a production-grade setup:

- Mosaic & multi-scale training (via YOLOv8 configuration)
- Albumentations transforms:
    - Blur, MedianBlur
    - ToGray
    - CLAHE
- Horizontal flip, color jitter, etc. as configured by Ultralytics

Additional training features:

- **Cosine learning rate schedule**
- **EMA** for more stable validation performance
- **Deterministic seed** for reproducibility
- Automatic logging of:
    - box loss, cls loss, dfl loss
    - mAP@0.5 and mAP@0.5:0.95
    - per-class metrics
    - confusion matrix and label plots

All logs and artifacts are stored under:

- training/logs/exp_coco8*/

2.4 Training Results

After 10 epochs on CPU, the trained model achieves the following validation scores on COCO8:

- Overall mAP@0.5: **~0.88**
- Overall mAP@0.5:0.95: **~0.62**
- Per-class performance is reasonable given the tiny dataset and CPU-only training.

Ultralytics also reports approximate runtime per image during validation:

- Pre-processing: ~2.2 ms
- Inference: ~75 ms (CPU, 640 × 640)
- Post-processing: ~1.4 ms

The best checkpoint from training is saved as:

- training/logs/exp_coco8*/weights/best.pt

and then copied into:

- models/latest.pt

for downstream optimization and deployment.

---

3. Model Optimization Pipeline

The **optimization** stage converts the trained PyTorch model into more deployable formats and prepares for TensorRT integration.

3.1 PyTorch → ONNX Export

Script: optimization/export_to_onnx.py

This script:

- Loads the trained PyTorch checkpoint from models/latest.pt
- Builds a dummy input tensor with dynamic dimensions
- Exports the model to **ONNX** with:
    - opset >= 12
    - **dynamic axes** for:
        - batch size (N)
        - image height (H)
        - image width (W)

The resulting file:

- models/model.onnx

To ensure correctness, the script runs a small consistency check:

- Feeds the same input through PyTorch and ONNX
- Compares outputs (class scores and boxes) within a numerical tolerance

This guarantees that the ONNX graph is compatible with its PyTorch counterpart.

3.2 TensorRT Engine Build (FP16 & INT8 – Code-level Implementation)

Scripts:

- optimization/calibrate_int8.py
- optimization/build_trt_engine.py

The assessment requires generation of both FP16 and INT8 TensorRT engines.

In this project:

- I implemented the complete **TensorRT pipeline in code**, including:
    - Entropy-based INT8 calibration using a subset of COCO8 training images
    - Dynamic optimization profiles for different input resolutions
    - Support for both FP16 and INT8 engine builds

Key design points:

- calibrate_int8.py:
    - Loads training images from:
        - C:/Users/Bekir/Desktop/object-detection-api/datasets/coco8/images/train
    - Preprocesses them to 640 × 640, normalizes to [0, 1], and converts to CHW
    - Uses IInt8EntropyCalibrator2 to generate a calibration cache
    - Writes out models/calibration.cache
- build_trt_engine.py:
    - Parses models/model.onnx with trt.OnnxParser
    - Sets **dynamic optimization profiles**, e.g.:
        - min: (1, 3, 480, 480)
        - opt: (1, 3, 640, 640)
        - max: (4, 3, 1280, 1280)

    - Builds:
        - FP16 engine → models/model_fp16.engine
        - INT8 engine → models/model_int8.engine (using the calibration cache)

Environment note:
The current development machine runs **Windows + Python 3.13 + CPU-only PyTorch** and does not have TensorRT or pycuda installed. For this reason, the TensorRT engines were **not built or executed locally**.
Instead, the pipeline is designed and implemented so that it can be executed on a proper **GPU + TensorRT** environment (e.g., Ubuntu, CUDA, Python 3.10) without code changes.

3.3 Benchmarking

Script: optimization/benchmarks.py

This script uses **ONNX Runtime** to benchmark the exported model:

- Warm-up iterations (e.g., 10)
- Multiple timed runs to compute:
    - average latency
    - p50 / p95 latency
    - effective FPS
- CPU utilization via psutil
- (Optionally) GPU metrics via pynvml in a GPU environment

Results are written to:

- optimization/benchmark_results.json

This gives a clear picture of the performance trade-offs between PyTorch and ONNX backends, and lays the groundwork for comparing future TensorRT engines.

---

4. Multi-Backend Inference Engine

Script: inference/detector.py

The **Detector** class abstracts over multiple inference backends:

- **PyTorch backend**:
    - Loads models/latest.pt via Ultralytics
    - Runs inference using the standard YOLOv8 API
- **ONNX backend**:
    - Loads models/model.onnx with ONNX Runtime
    - Preprocesses frames manually:

        - resize to 640 × 640
        - normalize to [0, 1]
        - convert to NCHW
    - Runs inference with InferenceSession on CPU
    - Applies custom post-processing (decoding predictions, non-max suppression)

- **TensorRT backend (planned)**:
    - The Detector is designed to support a future backend="tensorrt" mode, which would load:
        - models/model_fp16.engine or models/model_int8.engine
    - Due to the missing TensorRT runtime locally, this path is implemented at design level and can be activated later.

Common features across backends:

- Consistent preprocessing and post-processing
- Custom NMS implementation
- Support for batch inference
- Warm-up runs before timing
- Collection of latency statistics to feed into the monitoring dashboard

Example usage:

```
from inference.detector import Detector

detector = Detector(
  backend="onnx",
  model_path="models/model.onnx",
)

detections = detector(frame)  # returns list of boxes, scores, class_ids
```

5. Real-Time Video Engine and Tracking

Scripts:

- inference/tracker.py
- inference/video_engine.py
- inference/fusion.py

5.1 Tracker

The tracker is implemented as a simple **IoU-based tracker**:

- Class: SimpleIOUTracker
- For each new frame:
    - Assigns existing track IDs to new detections based on maximum IoU
    - Creates new tracks when no match is found
    - Removes stale tracks after a configurable number of frames

This is intentionally lightweight and easy to run in real time on CPU.

5.2 Detector + Tracker Fusion

video_engine.py implements a **hybrid detector + tracker pipeline**:

- The detector runs every **N frames** (e.g. every 5th frame)
- In the frames in between, only the tracker is updated
- For each track, the IoU between detection and tracker boxes is checked:
    - If IoU < 0.5 → the tracker is re-initialized with the new detection
    - Else → the tracker is trusted to reduce detector load

This strategy reduces the average inference cost while keeping tracking relatively stable, which is important for edge deployments with limited compute.

For simplicity, the reference implementation is single-threaded. In a production system, this can be extended to a queue-based multi-threaded design:

- Capture → Inference → Tracking → Visualization → Output

---

6. API Deployment (FastAPI + Docker)

Scripts:

- api/server.py
- api/schemas.py
- api/docker/Dockerfile

The model is wrapped as a **REST API** using FastAPI.

6.1 Endpoints

- GET /health
    - Basic health check endpoint.
- POST /detect
    - Accepts an image file (multipart/form-data).
    - Runs detection through the selected backend (PyTorch or ONNX).
    - Returns:
        - bounding boxes (x1, y1, x2, y2)
        - class IDs and labels
        - confidence scores
        - inference latency in milliseconds
- GET /metrics
    - Exposes runtime metrics collected by the monitoring module (see Section 7).

The API is started with:

uvicorn api.server:app --reload --port 8000

Interactive documentation is available at:

- http://127.0.0.1:8000/docs

6.2 Dockerization

A Dockerfile is provided under api/docker/Dockerfile to containerize the application. The base image is CPU-oriented in this environment, but the design is compatible with an NVIDIA TensorRT runtime image for GPU deployments.

In a real edge deployment, the Dockerfile would be adapted to:

- Use an NVIDIA-compatible base image
- Include TensorRT libraries
- Mount pre-built .engine files from the models/ directory

---

7. Monitoring and Observability

Scripts:

- monitoring/logger.py
- monitoring/fps_meter.py
- monitoring/dashboard.py

The monitoring layer is designed to make the system observable during real-time operation.

Key components:

- **FPSMeter**:
    - Tracks how many frames are processed per second
    - Provides a smoothed FPS value over a sliding window
- **Logger**:
    - Writes key events and metrics in JSON-friendly format
- **MetricsDashboard**:
    - Collects inference latencies (per backend)
    - Computes:
        - average latency
        - p50 / p90 / p95 latency
        - FPS
        - CPU utilization (via psutil)
        - Per-backend usage statistics (e.g. PyTorch vs ONNX)
    - Exposed to the outside world through /metrics endpoint

Example /metrics response (simplified):


```
{
  "inference_count": 10,
  "latency_ms": {
    "avg": 75.4,
    "p50": 70.1,
    "p90": 90.0,
    "p95": 100.0
  },
  "fps": 12.5,
  "cpu_percent": 35.0,
  "backend_usage": {
    "onnx": 10
  }
}
```


This design can be easily integrated with Prometheus and Grafana in a production environment.

8. Testing and Code Quality

The project includes a minimal but meaningful unit test suite under tests/:

- tests/test_inference.py
    - Verifies that the Detector runs end-to-end on a dummy image
    - Checks shapes of outputs (boxes, scores, class IDs)
- tests/test_onnx_shapes.py
    - Ensures that the ONNX model accepts dynamic shapes
    - Confirms that the exported model can run a forward pass

- tests/test_tracker.py
    - Checks that SimpleIOUTracker assigns IDs consistently
    - Tests basic drift and re-assignment scenarios

All tests pass:

pytest -q

-> 3 passed, 1 warning (pynvml deprecation warning)

This provides confidence that the core components behave as expected.

---

9. Limitations and Future Work

Although the current implementation demonstrates the full pipeline, there are several areas that can be extended in a real production system:

1. **Larger datasets**
    - COCO8 is a toy dataset used for demonstration.
    - In practice, a domain-specific dataset (e.g. VisDrone, UAVDT, or a custom industrial dataset) would be required.
2. **TensorRT execution**
    - The TensorRT FP16/INT8 pipeline is implemented at the code level but was not executed on this Windows + CPU setup.
    - On a GPU + TensorRT environment, the next step would be:
        - Run calibrate_int8.py → generate calibration.cache
        - Run build_trt_engine.py → produce model_fp16.engine and model_int8.engine
        - Add a backend="tensorrt" mode to the Detector to use these engines.

3. **More advanced trackers**
    - The current IOU-based tracker is simple and efficient.
    - For more challenging scenarios, DeepSORT or ByteTrack could be integrated.
4. **Multi-threaded video pipeline**
    - The reference implementation is single-threaded for clarity.
    - A production version would use a queue-based multi-threaded design:
        - Capture → Inference → Tracking → Visualization → Output

---

10. Conclusion

In this assessment, I implemented an end-to-end Edge AI Video Analytics pipeline that covers:

- Training a YOLOv8 model on the COCO8 dataset
- Exporting the model to ONNX with dynamic shapes
- Designing a full TensorRT FP16/INT8 optimization pipeline
- Building a multi-backend Detector abstraction
- Implementing a real-time detector + tracker video engine
- Exposing the model via FastAPI and containerizing it with Docker
- Adding monitoring, metrics, and basic unit tests

The code is structured to closely resemble a production-grade edge AI deployment, and is ready to be extended with GPU + TensorRT execution in a suitable environment.