Ok let's focus on these two parts:

pose_estimationa and object_detection - classification.

Everything that's done right now and everything left and everything else needed. give me directly.

Thought for 1m 6s

Thought for 1m 40s

Got it—here's the straight status for the two pieces, no fluff.

Pose estimation

What's already in the repo

- **Model assets folder**: `aixavier/models/pose/`
  - `download.sh` (grabs `yolov8n-pose.pt`).
  - `export_trt.sh` (exports ONNX and builds **FP16** TensorRT: `engines/yolov8n-pose-fp16.engine`).
  - `calibrate.sh` (INT8 calibration **stub**, not complete).
  - `README.md` with target perf on Xavier.
- **Detector config**: `configs/detectors/pose_velocity.yaml` points to `models/pose/engines/yolov8n-pose-fp16.engine`, input 640×640, `interval: 3`.
- **Runtime skeleton**: event-bus, runners/trackers/rules services exist and expect lines in `artifacts/*`, but there's **no actual pose inference service**.

What's missing / left to do

- **Inference service** (P0): No `detect_pose` process that:
  - Loads the **TensorRT** engine,
  - Runs keypoint inference,
  - Decodes YOLOv8-pose outputs (17 keypoints) to skeletons,
  - Publishes detections to the FileEventBus (e.g., `artifacts/detections/events.log`).

- **Velocity/analytics hook** (P0): `pose_velocity.yaml` implies velocity computation, but there's **no code** that:
  - Tracks keypoints per ID (needs tracker integration),
  - Computes per-joint velocity and emits rule-friendly events/attributes.
- **INT8 calibration** (P1): `calibrate.sh` is incomplete (… in command), needs dataset path & full `trtexec` flags.
- **Dependencies** (P1): Export scripts use `ultralytics` and `trtexec`, but `ultralytics` **isn't in** `requirements*.txt`; TensorRT runtime libs aren't wired in a detector image/Dockerfile.
- **Docker/service wiring** (P1): No `src/detectors/pose` code, Dockerfile, or `docker-compose` service for pose.
- **Tests** (P2): No unit/integration tests for pose decode, accuracy, or latency.

What else you'll need to ship

- **Tracker**: ByteTrack (or KPT-aware tracker) to maintain person IDs -> stable keypoint streams.
- **Schema**: Event JSON for pose (keypoints, conf, bbox, track_id, ts) and velocity features.
- **Perf targets**: Baseline FPS/latency on target SKUs; watchdog + metrics (Prometheus).
- **Edge packaging**: A detector container with CUDA/TensorRT, healthcheck & logs.

Fastest path to green (pose)

1. **Create detector service** `src/detectors/pose/main.py`:
   - Load `models/pose/engines/yolov8n-pose-fp16.engine` with TensorRT.
   - Pre/postprocess to emit: `{type:"pose", camera_id, track_id, bbox, keypoints:[(x,y,conf)*17], ts}` to `artifacts/detections/events.log`.
2. **Add tracker**:
   - Start with bbox-level ByteTrack on person boxes inferred from keypoints (or run a light person detector at `interval: N` to seed IDs).
3. **Velocity module**:
   - Compute per-joint Δ and magnitude per frame; publish `{type:"pose_velocity", joints:{...}, track_id, ts}`.
4. **Finish INT8**:

  - Fix `calibrate.sh` with full `trtexec` flags and `DATASET_PATH_POSE_CALIB`, produce `engines/yolov8n-pose-int8.engine`.
5. **Wire it up**:
   - Add Dockerfile, service in `docker-compose.yml` (`detect_pose`), envs, volumes.
6. **Add tests**:
   - Unit test ONNX → decoder on a small clip; golden JSON for 10 frames.

Object detection + classification

What's already in the repo

- **YOLO detector assets**: `aixavier/models/yolo/`
  - `download.sh` (Ultralytics weights), `export_trt.sh` (builds **FP16**), `calibrate.sh` (INT8 done properly), `README.md`.
- **Detector config**: `configs/detectors/object.yaml` → `models/yolo/yolov8n-fp16.engine`, classes include `person, bag, suitcase, backpack, fire, smoke`.
- **Use-case recipe**: `configs/usecases/object_identification_classification.yaml` (PPE logic: `hard_hat`, `safety_vest`, `tool`, `missing_ppe` rule).
- **Runtime skeleton**: event bus, rules, tracker scaffolding (no real ByteTrack), exporter/recorder services.

What's missing / left to do

- **Inference service** (P0): No `detect_object` process that runs YOLO TensorRT, decodes boxes/classes, emits events.
- **Class coverage mismatch** (P0): Use-case expects **PPE classes** (`hard_hat`, `safety_vest`, `tool`) but default `yolov8n` doesn't contain those → need **fine-tuned weights** or a second-stage classifier.
- **Tracking** (P0/P1): `tracker: bytetrack` is referenced but not implemented; `src/trackers` is a stub.
- **Classification stage** (P1): If you keep general YOLO, you'll need ROI **classifier** for PPE/tools from crops:

  - No classifier model, code, or config present.
- **Dependencies & Docker** (P1): Same as pose: `ultralytics` not in requirements; no detector Dockerfile/service.
- **INT8 dataset pathing** (P1): `DATASET_PATH_OBJECT_CALIB` placeholder; needs real path and sample images.
- **Tests** (P2): No decode/accuracy tests; no end-to-end rule trigger tests tied to detections.

What else you'll need to ship

- **Either**:
  - A **fine-tuned YOLO** (PPE classes) → simplest single-stage pipeline, or
  - **Two-stage**: General YOLO (person/tool-ish) → crop → lightweight classifier (e.g., MobileNetV3/ViT-Tiny) to label `hard_hat/safety_vest/tool/none` .
- **Label schema + thresholds**: Confidence, dwell seconds (already in use-case), NMS/IoU sane defaults.
- **ByteTrack** integration: associate boxes across frames for dwell/missing-PPE logic.
- **Evaluation harness**: mAP@.5:.95 for detector; top-1 for classifier; rule precision/recall on a small validation clip set.

Fastest path to green (object + classification)

1. **Create detector service** `src/detectors/object/main.py` :
   - Load `models/yolo/yolov8n-fp16.engine` , preprocess 640×640, decode, publish:

     ```
     {type:"object", camera_id, ts, detections:[{bbox, cls, conf, id(op
     ```
2. **Add ByteTrack**:
   - Port a minimal ByteTrack impl; assign `track_id` to detections; publish `type:"object_tracked"` .
3. **Unblock PPE logic** (pick one):
   - **Option A (preferred for time)**: drop-in **fine-tuned YOLO PPE engine**; update `configs/detectors/object.yaml` classes to `hard_hat/safety_vest/tool/person` .
   - **Option B**: add `src/classifier/ppe/` :
     - Crop person head/torso/tool ROIs,

    - Run classifier ONNX/TRT,
    - Enrich detection with `attributes:{ppe:"hard_hat"/"none", ...}`.
4. **Rules wiring**:
    - Ensure emitted event fields match `configs/usecases/object_identification_classification.yaml` (e.g., `classes_required`, `actor_class:"person"`, `dwell_seconds`).
5. **Finish INT8** (optional but good):
    - Point `DATASET_PATH_OBJECT_CALIB` to a few hundred varied frames and run `calibrate.sh` → `engines/yolov8n-int8.engine`.
6. **Shipables**:
    - Detector Dockerfile + `docker-compose` service `detect_object`.
    - Add `ultralytics` to `requirements-dev.txt` (export only) and document TensorRT/JetPack for runtime.
    - Add unit tests for post-proc decode + a 30-sec demo E2E that triggers `missing_ppe`.

Quick P0 checklist to start today

- Implement `src/detectors/object/main.py` + Dockerfile + compose service.
- Implement `src/detectors/pose/main.py` + Dockerfile + compose service.
- Add minimal ByteTrack and return `track_id`.
- Decide PPE path: **fine-tuned YOLO PPE** (swap engine) **or** add ROI classifier; wire to rules.
- Remove `…` placeholders in `models/pose/calibrate.sh`; verify both calibrations run.
- Add `ultralytics` to dev reqs for export; document `trtexec`/TensorRT runtime.
- Add 2 tiny integration tests: one sample video → expected JSON events for both services.

If you want, I can stub both detector services (TensorRT runtime + post-proc) and a minimal ByteTrack in one go so you can drop in engines and run.

These are the models I am thinking of using. can you browse and tell me if there exists any better models. latest and better performance.

pose_estimation → RTMPose-m.
object_detection and classification → YOLO11n or RT-DETRv2-R50.

Thought for 58s >

Here's the straight take, after checking the latest benchmarks and repos:

Pose estimation

Your pick: RTMPose-m
Call: Still a strong default. It's ~75.8 AP on COCO keypoints with excellent throughput; among real-time top-down methods it's hard to beat for accuracy/speed on 17-kpt human pose. arXiv +1

If you want something *newer* that can scale better with many people per frame:

- **RTMO (one-stage, CVPR'24):** ~74.8 AP (RTMO-l) and very high FPS on V100; one-stage design avoids the top-down per-person cost, so it shines in crowded scenes. Accuracy is a hair under RTMPose-m but throughput in multi-person settings can be superior. CVF Open Access
- **Whole-body (133 kpts) focus: DWPose** distillation lifts RTMPose-L on COCO-WholeBody to ~66.5 AP (beats RTMPose-X teacher). Use if you truly need hands/face/feet. arXiv +1
- **YOLO-pose (YOLO11-pose):** convenient, export-friendly, but there isn't clear evidence it beats RTMPose-m on COCO-keypoints AP; pick it mainly for simpler tooling. Ultralytics Docs

Bottom line (pose):

- Keep **RTMPose-m** for best AP with typical person counts.

- Switch to **RTMO** if you're bottlenecked by many simultaneous people and want one-stage scaling. CVF Open Access

Object detection (and your "classification" stage)

Your picks: YOLO11n **or** RT-DETRv2-R50

What's better/newer right now:

- **YOLO12n (2025):** successor to YOLO11n; reported ~40.6 mAP (vs ~39.5 for YOLO11n) at similar tiny-model latency. If you want the *same nano footprint* but a bit more accuracy, YOLO12n is the cleanest upgrade. Replicate +2
- **RT-DETRv2-R50:** strong real-time transformer baseline around **53.1 AP @ ~108 FPS (T4)**; **~55.3 AP** with Objects365 pretraining. Great if you can afford a midsize model and want a big accuracy jump over nano YOLO. GitHub +1
- **RF-DETR (2025):** new open-source real-time DETR variant; **first real-time model to exceed 60 AP on COCO** (base sizes). If you prioritize accuracy while staying real-time on decent GPUs, this is currently the ceiling among real-time models. GitHub +1

Classification add-on:
If your pipeline needs attributes (e.g., PPE) beyond detector classes, keep a lightweight ROI classifier head (any small CNN/ViT). If you stay in the Ultralytics stack, the **YOLO12 classification** head is available; otherwise use a tiny EfficientNet/MobileNet. (YOLO12 family + export paths are documented and actively maintained.) Ultralytics Docs

Bottom line (detection):

- Need **nano** speed/size? Pick **YOLO12n** over YOLO11n. Small, simple upgrade. Replicate +1
- Need **higher AP but still real-time**? Use **RT-DETRv2-R50**; if you can go bigger and want the best real-time AP reported, evaluate **RF-DETR**. GitHub +2

TL;DR picks

- **Pose:** Stay **RTMPose-m** unless you're crowd-heavy → then **RTMO**. arXiv +1

- **Detection:** Upgrade **YOLO11n → YOLO12n** (same tiny tier, better mAP). For accuracy at real-time, go **RT-DETRv2-R50**, or **RF-DETR** if you can budget more compute. Replicate +3

If you want, I can map these to your target hardware (Jetson/CPU/T4/Orin) and suggest exact variants + TensorRT export flags and expected FPS.

Will this object detection model also do the classification??

Thought for 17s

Short answer: **yes, but only for the classes it was trained on.**

- **YOLO/RT-DETR** are *detectors with a classifier head per box*. Every detection comes with a class label and score (e.g., `person` , `car` , `dog`).
- If by "classification" you mean **PPE attributes** like `hard_hat` , `safety_vest` , `tool` , the **default COCO weights won't do it**—those labels aren't in COCO. You'll get `person` and generic objects, but **not** "wearing a hard hat vs not."

What to do for PPE "classification"

You have two clean paths:

A) Single-stage (recommended if you can get data/weights)

Fine-tune **YOLO / RT-DETR** on a PPE dataset so the detector's class set is:
`person, hard_hat, safety_vest, tool` (and whatever else you need).
Pros: simple runtime, fastest E2E, one engine.
Cons: needs labeled boxes (or ready weights).

How you'd wire it: the detector already "classifies" each box → your events directly contain PPE classes, no second pass.

B) Two-stage (no custom detector training required)

1. Run a general detector (YOLO/RT-DETR) to get `person` boxes (and optionally head/torso crops).

2. Run a **light ROI classifier** (MobileNet/EfficientNet/ViT-Tiny or a YOLO “cls” head) on those crops to predict:
    - `ppe.hard_hat: yes/no` , `ppe.safety_vest: yes/no` , `tool: yes/no` (multi-label).

      Pros: flexible, easy to add/change attributes.

      Cons: extra compute and code path.

Quick recommendations for your picks

- **If you stay tiny:** use **YOLO(11→)12n** fine-tuned with PPE labels (**A**), or pair **YOLO12n** with a tiny ROI classifier (**B**).
- **If you want higher AP and can afford it: RT-DETRv2-R50** fine-tuned (**A**). If you can’t fine-tune now, use **B** with RT-DETRv2 for detection.

Event schema tip (works for both paths)

For single-stage PPE detector:

```
{ "type":"object", "track_id":42, "cls":"hard_hat", "conf":0.91,
"bbox":[x,y,w,h], "ts":... }
```

For two-stage:

```
{ "type":"object_tracked", "track_id":42, "cls":"person", "conf":0.89,
"bbox":[x,y,w,h], "ts":..., "attributes": { "ppe": { "hard_hat": true,
"safety_vest": false }, "tool": true } }
```

Bottom line

- **Yes**, the detector “does classification” for **its** class list.
- For **PPE** or other attributes, you must **(A) train/use a detector with those classes** or **(B) add an ROI classifier**.

  Tell me which way you want to go and your target device (Jetson/T4/CPU), and I’ll pin exact model variants, export flags, and expected FPS.

Alright full implementation pipeline. How do i implement how do everything

Thought for 22s

Awesome—here's a complete, end-to-end implementation plan you can follow today. I'm giving you a **battle-tested default** plus variants so you can swap pieces without re-architecting.

0) TL;DR architecture

- **Cameras** → `ingest` (RTSP → frames)
- **Detectors:**
  - `detect_object` (YOLO/RT-DETR TensorRT) → boxes + class + conf
  - `detect_pose` (RTMPose or YOLO-pose TensorRT) → keypoints + conf
- **Tracker:** `bytetrack` → adds `track_id` to object streams
- **(Optional) ROI classifier:** crops from tracked boxes → PPE attributes (multi-label)
- **Analytics:** dwell time / missing PPE / velocity on keypoints
- **Event Bus:** newline-delimited JSON (NDJSON) files or Kafka/Redis (start with files)
- **Exporter/Recorder/Alerts:** consume events → UI, DB, webhooks

1) Repos & layout

```
aixavier/
  src/
    detectors/
      object/
        main.py              # TRT runtime + postproc + publish
        postproc.py          # decode, NMS
        docker/Dockerfile
      pose/
        main.py
        postproc.py
        docker/Dockerfile
    trackers/
      bytetrack.py
    classifier/
      ppe/
        infer.py            # multi-label ROI classifier (optional)
        export.md
    analytics/
      ppe_rules.py
      pose_velocity.py
    bus/
      file_event_bus.py
  configs/
    detectors/object.yaml
    detectors/pose.yaml
    classifier/ppe.yaml
    analytics/ppe_rules.yaml
    analytics/pose_velocity.yaml
  models/
    object/
      engines/...
      onnx/...
    pose/
      engines/...
      onnx/...
    classifier/
      ppe_cls.onnx / .engine
  docker-compose.yml
  tests/
    data/ (images, short clips)
    e2e_object.yaml
    e2e_pose.yaml
```

2) Object detection (+ classification)

2.1 Choose your path

- **Path A (single-stage, recommended if you can fine-tune):** Train detector with your domain labels
  e.g., `person`, `hard_hat`, `safety_vest`, `tool` (+ anything else).
  → One engine, simplest runtime.
- **Path B (two-stage, fastest to ship without training):**
  General detector (`person` , etc.) **+** lightweight ROI classifier for PPE attributes

(`hard_hat yes/no`, `safety_vest yes/no`, `tool yes/no`).
→ Add a small ONNX to classify crops.

I'll detail **both**; your wiring stays almost identical.

2.2 Model export

Option 1: YOLO (nano/tiny friendly)

- **Train / load** (Ultralytics): `yolo detect train` or `yolo detect predict` as needed.
- **Export ONNX:**
`yolo export model=best.pt format=onnx imgsz=640 opset=12 dynamic=False`
- **Build TensorRT (FP16):**

```
trtexec --onnx=models/object/onnx/yolo.onnx \
        --saveEngine=models/object/engines/yolo-fp16.engine \
        --fp16 --workspace=4096 --shapes=input:1x3x640x640
```

- **(Optional INT8):** prepare 500–1000 calibration images, then:

```
trtexec --onnx=.../yolo.onnx --int8 --fp16 \
        --saveEngine=.../yolo-int8.engine \
        --calib=max --calibCache=.../yolo.calib
```

Option 2: RT-DETRv2 (for higher AP, still realtime on T4/Orin)

- Export via the repo's tools to ONNX, then `trtexec` as above (batch=1, 640 or 800).

ROI classifier (Path B)

- Train a **multi-label** head (MobileNetV3 / EfficientNet-B0) on crops:
  - labels: `ppe.hard_hat`, `ppe.safety_vest`, `tool` (each ∈ {0,1})
- Export ONNX: `torch.onnx.export` with `input: 1x3x224x224`
- Build TRT FP16: `trtexec --onnx=ppe_cls.onnx --saveEngine=ppe_cls-fp16.engine --fp16`

2.3 Detector service (`src/detectors/object/main.py`)

Responsibilities:

- Read frames (shared queue / RTSP reader)

- Preprocess (letterbox to 640×640, FP16)
- TRT enqueue → outputs
- Decode (logits → boxes, scores, classes, NMS)
- Publish NDJSON events

NDJSON event (Path A – trained PPE detector):

```
{"type":"object","camera_id":"cam1","ts":1730227201.123, "detections":
[{"bbox":[x,y,w,h],"cls":"hard_hat","conf":0.93}]}
```

NDJSON (Path B – general detector):

```
{"type":"object","camera_id":"cam1","ts":1730227201.123, "detections":
[{"bbox":[x,y,w,h],"cls":"person","conf":0.89}]}
```

Minimal post-proc knobs

- `score_thresh` : 0.25–0.4
- `nms_iou` : 0.45
- max detections per frame: 100

2.4 Tracking (`src/trackers/bytetrack.py`)

- Input: per-frame detections (x,y,w,h, score, class)
- Output: add `track_id` with stable IDs
- Publish:

```
{"type":"object_tracked","camera_id":"cam1","ts":..., "detections":
[{"track_id":7,"bbox":[...],"cls":"person","conf":0.89}]}
```

2.5 ROI classifier (Path B)

- Subscribe to `object_tracked`
- For each `person` (or tool-like) box: crop ROI(s)
  - Head/torso crop: use bbox heuristics or a tiny head-finder
- Run TRT engine `ppe_cls-fp16.engine`
- Attach attributes:

```
{"type":"object_attributed","camera_id":"cam1","ts":..., "detections":
[{"track_id":7,"bbox":[...],"cls":"person", "attributes":{"ppe":
```

```
{"hard_hat":true,"safety_vest":false},"tool":true}}]}
```

3) Pose estimation

3.1 Choose your model

- **RTMPose-m** (best accuracy/speed balance, top-down)
- **RTMO** (one-stage, great for crowded scenes)
- **YOLO-pose** (easiest export path if you're already in Ultralytics)

3.2 Export → TensorRT

A) RTMPose via MMDeploy (canonical)

1. Convert to ONNX with the official MMPose/MMDeploy toolchain
2. `trtexec --onnx=rtmpose.onnx --saveEngine=rtmpose-fp16.engine --fp16 --shapes=input:1x3x640x640`

B) YOLOv8-pose (fastest path)

- `yolo pose export model=yolov8n-pose.pt format=onnx imgsz=640`
- `trtexec ... --saveEngine=yolov8n-pose-fp16.engine --fp16`

3.3 Pose service (`src/detectors/pose/main.py`)

- Preprocess → TRT → decode heatmaps/offsets or direct kpts
- Output schema:

```
{"type":"pose","camera_id":"cam1","ts":..., "detections":[ {"bbox":
[x,y,w,h],"keypoints":[[x,y,conf],... 17], "conf":0.91, "person_id":12}
]}
```

Tracking for pose: reuse `object_tracked` IDs by associating each skeleton with the nearest tracked `person` bbox; fallback to simple KPT-IoU matching if needed.

3.4 Pose velocity analytics (`src/analytics/pose_velocity.py`)

- For each `person_id` , keep a ring buffer of last N frames (kpts)
- Velocity per joint `v = ||p_t - p_{t-1}|| / Δt`
- Publish:

```
{"type":"pose_velocity","camera_id":"cam1","ts":..., "track_id":12,
"joints":{"wrist_r":0.42,"ankle_l":0.17}, "v_mean":0.21}
```

4) Analytics: PPE rules (works for Path A or B)

Config (`configs/analytics/ppe_rules.yaml` **)**

```
actor_class: person dwell_seconds: 3 required: hard_hat: true
safety_vest: true tool_allowed: false
```

Rule engine (`src/analytics/ppe_rules.py` **)**

- Subscribe to `object_tracked` (Path A: class already `hard_hat`, etc.) or `object_attributed` (Path B)
- Maintain per `track_id` timers for dwell
- Emit violations:

```
{"type":"alert","subtype":"missing_ppe","track_id":7,"camera_id":"cam1",
"missing":["hard_hat"],"dwell":3.2,"ts":...}
```

5) Event bus (file-based to start)

Write: append NDJSON to e.g. `artifacts/detections/object.ndjson`

Read: tail-like stream per consumer

Common pub helper:

```
def publish(path, obj): with open(path, "a") as f:
f.write(json.dumps(obj) + "\n")
```

(When scaling, swap to Kafka/Redis without touching producers/consumers—just replace the adapter.)

6) Docker & Compose

6.1 Detector Dockerfiles (GPU)

`src/detectors/object/docker/Dockerfile`

```
FROM nvcr.io/nvidia/cuda:12.2.0-runtime-ubuntu20.04 RUN apt-get update
&& apt-get install -y python3-pip libglib2.0-0 libsm6 libxrender1
libxext6 && rm -rf /var/lib/apt/lists/* COPY requirements.txt
/app/requirements.txt RUN pip3 install --no-cache-dir -r
/app/requirements.txt WORKDIR /app COPY . /app ENV
TRT_LOGGER_SEVERITY=ERROR CMD ["python3",
"src/detectors/object/main.py", "--config",
"configs/detectors/object.yaml"]
```

(Use `nvcr.io/nvidia/l4t-jetpack` base on Jetson.)

6.2 docker-compose.yml

```
version: "3.8" services: detect_object: build: { context: .,
dockerfile: src/detectors/object/docker/Dockerfile } runtime: nvidia
environment: [ NVIDIA_VISIBLE_DEVICES=all ] volumes: -
./models:/app/models - ./configs:/app/configs -
./artifacts:/app/artifacts tracker: build: { context: ., dockerfile:
src/trackers/docker/Dockerfile } runtime: nvidia volumes: [
./artifacts:/app/artifacts ] detect_pose: build: { context: .,
dockerfile: src/detectors/pose/docker/Dockerfile } runtime: nvidia
volumes: [ ./models:/app/models, ./configs:/app/configs,
./artifacts:/app/artifacts ] analytics: build: { context: ., dockerfile:
src/analytics/docker/Dockerfile } volumes: [ ./configs:/app/configs,
./artifacts:/app/artifacts ]
```

> Make sure host has `--gpus all` (Compose v2 uses `runtime: nvidia` or `deploy.resources.reservations.devices`).

7) Configs (detectors)

`configs/detectors/object.yaml`

```
engine: models/object/engines/yolo-fp16.engine imgsz: 640 score_thresh:
0.35 nms_iou: 0.45 max_det: 100 publish:
artifacts/detections/object.ndjson
```

`configs/detectors/pose.yaml`

```
engine: models/pose/engines/rtmpose-fp16.engine imgsz: 640
score_thresh: 0.3 publish: artifacts/detections/pose.ndjson
```

`configs/classifier/ppe.yaml` (Path B)

```
engine: models/classifier/ppe_cls-fp16.engine input_size: 224 labels:
["ppe.hard_hat","ppe.safety_vest","tool"] thresholds: [0.5, 0.5, 0.6]
publish: artifacts/detections/object_attributed.ndjson
```

8) Tests (ship-blocking)

8.1 Unit

- **Postproc decode**: raw logits → 5 known boxes (golden)
- **NMS**: deterministic results for overlaps
- **Pose decode**: kpt heatmaps → 17 points within tolerance
- **Classifier**: logits → multi-label thresholding

8.2 Integration (short clips)

- `tests/e2e_object.yaml` : run `detect_object` (+ tracker) on a 10-sec clip
  → assert at least one `person` with stable `track_id`
  → if Path B, assert attributes JSON keys exist.
- `tests/e2e_pose.yaml` : run `detect_pose` on same clip
  → assert 17 kpts per detected person
  → if velocity enabled, assert `pose_velocity` events emitted.

9) Observability & ops

- **Health endpoints:** each service exposes `/:health` (HTTP 200)
- **Metrics:** Prometheus counters: FPS, latency (p50/p95), dropped frames, publishes/sec
- **Logs:** structured JSON; at least: `camera_id` , `frame_id` , `model` , `lat_ms`

10) Performance checklist

- Pre-allocate device buffers; reuse CUDA streams
- Convert to **NHWC/NCHW** as model expects once; avoid per-frame reallocations
- Use **FP16** first; INT8 later if you hit a wall (ensure representative calibration)
- Pin host memory for H2D copies; batch size = 1 for live streams
- Run **NMS on GPU** if available; otherwise optimize CPU NMS (NumPy → torch/onnx runtime kernels)
- Turn on **affinity**: dedicate cores to decode vs infer on Jetson

11) What to do today (P0 checklist)

1. **Pick path** for PPE:
   - If you **have labeled PPE data** → Path A (single-stage).
   - If you **don't yet** → Path B (two-stage with ROI classifier).
2. **Build engines**
   - Detector (YOLO or RT-DETRv2) → ONNX → TRT FP16
   - Pose (RTMPose or YOLO-pose) → ONNX → TRT FP16
   - (Path B) ROI classifier → ONNX → TRT
3. **Implement services**
   - `src/detectors/object/main.py` + `postproc.py`
   - `src/trackers/bytetrack.py`
   - (Path B) `src/classifier/ppe/infer.py`
   - `src/detectors/pose/main.py` + `pose_velocity.py`
4. **Wire configs & compose**
   - Fill `configs/*` as shown; mount `models/` & `artifacts/` volumes
5. **Run e2e tests** on short clips; fix thresholds until stable
6. **Add health + metrics**; set alerts on FPS < target or publish stalls

12) Minimal code stubs (drop-in)

`src/detectors/object/main.py` (skeleton)

```
import json, time import numpy as np import tensorrt as trt,
pycuda.driver as cuda, pycuda.autoinit from postproc import decode def
load_engine(path): logger = trt.Logger(trt.Logger.ERROR) with open(path,
"rb") as f, trt.Runtime(logger) as rt: return
rt.deserialize_cuda_engine(f.read()) def allocate(engine): ctx =
engine.create_execution_context() bindings, host, device = [], {}, {}
for i, name in enumerate(engine): dtype =
trt.nptype(engine.get_binding_dtype(name)) shape =
tuple(engine.get_binding_shape(i)) size = np.prod(shape) host[name] =
cuda.pagelocked_empty(size, dtype) device[name] =
cuda.mem_alloc(host[name].nbytes) bindings.append(int(device[name]))
stream = cuda.Stream() return ctx, bindings, host, device, stream def
infer(ctx, bindings, host, device, stream, inp):
np.copyto(host["images"], inp.ravel())
cuda.memcpy_htod_async(device["images"], host["images"], stream)
ctx.execute_async_v2(bindings, stream.handle)
cuda.memcpy_dtoh_async(host["output0"], device["output0"], stream)
stream.synchronize() return host["output0"] def publish(path, obj): with
open(path, "a") as f: f.write(json.dumps(obj)+"\n") def main(cfg): eng =
load_engine(cfg["engine"]) ctx, bindings, host, device, stream =
allocate(eng) while True: frame = read_frame_somehow() # BGR HxWx3 img,
scale, pad = preprocess_letterbox(frame, 640) out = infer(ctx, bindings,
host, device, stream, img) dets = decode(out, scale, pad,
cfg["score_thresh"], cfg["nms_iou"]) evt =
{"type":"object","camera_id":cfg.get("camera_id","cam1"),
"ts":time.time(),"detections":dets} publish(cfg["publish"], evt) if
__name__ == "__main__": import yaml, sys cfg =
yaml.safe_load(open(sys.argv[-1])) main(cfg)
```

(Pose is analogous; `postproc` differs.)

If you want, I can generate **ready-to-run stubs** for `detect_object`, `detect_pose`, `bytetrack`, the ROI classifier, and sample configs exactly in your tree so you can `docker compose up` and see events flowing.