YOLUX: Comments on the use of YOLO (You Only Look Once) to detect defects on luxury leather goods and stains on textiles

---

Olivier Vitrac, PhD., HDR | olivier.vitrac@adservio.fr – 2025-10-23

Summary

YOLO can work well for defect detection on luxury leather goods and stains on textiles, but you'll get the best results with a **hybrid pipeline** and some domain-specific care in data, optics, and training. Below is a technical plan that reflects what typically works in production for *small, subtle, highly variable* defects.

> *Access to all* files*, read this file in* PDF

---

1 | YOLO Overview

🧩 1.1 | Open-source status

The original **YOLOv1** was released by **Joseph Redmon et al.** in 2016 with full open-source code under a **GPL-style license**. Since then, many variants have been released by the community, all open-source under permissive licenses:

Version	Developer / Organization	Year	Framework	License / Repo
YOLOv1–v3	Joseph Redmon & Ali Farhadi (U. Washington)	2016–2018	Darknet (C/CUDA)	pjreddie/darknet
YOLOv4	Alexey Bochkovskiy	2020	Darknet	AlexeyAB/darknet
YOLOv5	Ultralytics	2020	PyTorch	ultralytics/yolov5
YOLOv6	Meituan	2022	PyTorch	meituan/YOLOv6
YOLOv7	WongKinYiu	2022	PyTorch	WongKinYiu/yolov7
YOLOv8	Ultralytics	2023	PyTorch	ultralytics/ultralytics

> *All these are* ***freely available*** *and can be retrained on custom datasets (COCO, Pascal VOC, your own images, etc.).*
>
> 📚 ***References***
>
> - *Redmon, J., et al. (2016). You Only Look Once: Unified, Real-Time Object Detection. CVPR.*
> - *Bochkovskiy, A., et al. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv:2004.10934.*
> - *Ultralytics (2023). YOLOv8 Documentation.* https://docs.ultralytics.com

---

⚙️ 1.2 | The method — "You Only Look Once"

YOLO introduced a **single-stage**, fully-convolutional approach to object detection.

Key idea

Instead of generating region proposals (like R-CNN), YOLO divides the input image into an $S \times S$ grid. Each grid cell directly **predicts**:

- B **bounding boxes** (center (x, y), width w, height h)
- A **confidence score** per box
- **Class probabilities** for C object categories

Output tensor

For example, for COCO ($C = 80$ classes):

$$\text{output shape} = S \times S \times (B \times 5 + C) \tag{1}$$

where 5 = $(x, y, w, h, \text{confidence})$.

Example (YOLOv3): $S = 13, B = 3, C = 80 \Rightarrow 13 \times 13 \times (3 \times 5 + 80) = 13 \times 13 \times 95$.

Network architecture

- **Backbone** (feature extractor): e.g. Darknet-53, CSPDarknet, EfficientNet.
- **Neck** (multi-scale fusion): e.g. FPN, PANet.
- **Head** (detection layers): predicts boxes + scores at 3 scales.

This enables **end-to-end** detection in one forward pass — hence the name.

🧠 1.3 | How the detection model was trained

Training YOLO is a standard supervised deep learning process using **annotated datasets** (bounding boxes + labels).

(a) Datasets

- **COCO** (Common Objects in Context, 118k train images, 80 classes)
- **Pascal VOC** (20 classes)
- **Open Images** (wider coverage)

(b) Loss functions

Earlier versions used a sum-of-squared-errors loss. Modern YOLOs use **compound losses** combining:

- **Bounding box regression loss:** IoU / GIoU / CIoU / DIoU loss
- **Objectness loss:** binary cross-entropy
- **Classification loss:** cross-entropy / focal loss

For example, in YOLOv5:

$$\mathcal{L} = \lambda_{box}\mathcal{L} * box + \lambda * obj\mathcal{L} * obj + \lambda * cls\mathcal{L}_{cls} \tag{2}$$

(c) Optimization

- Optimizer: SGD or Adam
- Data augmentation: mosaic, mixup, random scaling, hue/saturation jitter, etc.
- Batch normalization and residual connections
- Trained on multi-GPU setups with millions of iterations

(d) Anchors

YOLOv2+ introduced **anchor boxes** (like Faster R-CNN) to capture multi-scale shapes. Recent versions (YOLOv8) use **anchor-free** decoupled heads for simplicity and performance.

🚀 1.4 | Performance evolution

Version	mAP (COCO)	Speed	Notes
YOLOv1	63.4% VOC	~45 fps	First real-time detector
YOLOv3	57.9%	~30 fps	Multi-scale + anchors
YOLOv5s	37.4%	~140 fps	PyTorch + modern training
YOLOv8n	37.3%	~150 fps	Anchor-free, flexible tasks (detect/segment/pose)

🧩 1.5 | Training your own YOLO model

In PyTorch (YOLOv5+):

```
pip install ultralytics
yolo detect train data=coco.yaml model=yolov8n.pt epochs=100
imgsz=640
```

You can define your own dataset YAML and fine-tune pretrained weights.

🧩 1.6 | Invariance properties (details)

Let's analyze the invariances (and non-invariances) of **YOLO-type convolutional detectors** precisely and mechanistically.

🧩 1.6.1 | Translation invariance (✓ *approximate but effective*)

⚙️ Mechanism

- Convolutional layers are **shift-equivariant**: if you translate the input by $(\Delta x, \Delta y)$, the feature maps are translated by the same offset.
- Pooling and stride introduce **spatial subsampling**, yielding *translation invariance* to small shifts.
- The **grid structure** ($S \times S$ cells) assigns detections to cells based on object center coordinates.

✅ Consequence

YOLO exhibits:

- **Good invariance** to small translations — object shifts cause nearly identical detections.
- **Quantization sensitivity** at cell boundaries: an object crossing a grid line may flip its assignment to another cell and slightly change confidence/box regression.

> ⓘ **Note**
>
> Later YOLOs mitigate this with **multi-scale features** and **anchor boxes**; translation invariance is not *mathematically exact* but *empirically strong*.

🌀 1.6.2 | Rotation invariance (✗ *not inherent*)

⚙️ Mechanism

- Convolutions are **not rotation-equivariant**; kernels slide in fixed orientations (horizontal/vertical).
- Unless the dataset includes rotated examples or data augmentation (random rotations, flips), the model will not generalize well to rotated objects.

🧠 Remedies

To improve rotation invariance:

1. **Data augmentation**: random rotations, affine transforms (standard in YOLO training).
2. **Rotated bounding boxes**: variants like *Rotated-YOLO*, *Oriented-YOLO*, or *YOLOv8-OBB* explicitly predict orientation angles (θ).
3. **Group-equivariant CNNs (G-CNNs)**: theoretical frameworks using rotationally symmetric filters.

✅ Practical result

Modern YOLOs achieve **robustness** (through augmentation), not true **rotation invariance** (i.e., feature maps are not equivariant to rotation).

🪞 1.6.3 | Reflection / symmetry invariance (± *partial*)

⚙️ Mechanism

- Reflection along x or y axes is **not symmetric** for most object classes (e.g. "LEFT arrow" ≠ "RIGHT arrow").
- Convolutions can learn symmetry if the dataset includes mirrored samples.

✅ Remedies

- Horizontal flips are *standard augmentation*; this creates approximate **mirror invariance**.
- Vertical flips are often avoided (many objects are gravity-oriented).

🔍 1.6.4. Scale invariance (✓ *multi-scale, partial*)

Scale is a core invariance for detection.

⚙️ Mechanism

- Multi-scale feature fusion (FPN, PANet) gives **scale robustness**.
- Anchors (in v2–v7) or decoupled heads (v8) handle various object sizes.
- Still, scale invariance is *discrete* (depends on the pyramid levels, not continuous).

🧠 Summary Table

Invariance	Mechanism	Degree	Notes
Translation	Convolution, grid assignment	★★★★☆	Excellent but quantized by cell grid
Rotation	None intrinsic	★★☆☆☆	Needs augmentation or special heads

Invariance	Mechanism	Degree	Notes
Reflection / symmetry	Augmentation (flip)	★★★☆☆	Horizontal flip only
Scale	FPN / anchors / pyramids	★★★★☆	Robust across octaves
Illumination / contrast	Augmentation	★★☆☆☆	Not architectural

🧩 1.7 | Testing invariance properties

Formally, a CNN like YOLO is:

- **Translation-equivariant** (if no pooling, padding preserved)
- **Not rotation-equivariant** nor reflection-equivariant
- **Locally invariant** due to pooling and non-linearities

Invariance $\Rightarrow f(Tx) = f(x)$ Equivariance $\Rightarrow f(Tx) = T' f(x)$

YOLO's convolutions are **equivariant** to translation, not **invariant** (detections move consistently with the object).

🔬 Experimental verification

You can test this by inference on transformed images:

```
from ultralytics import YOLO
from PIL import Image
import torchvision.transforms as T

model = YOLO("yolov8n.pt")

img = Image.open("dog.jpg")
rot = T.functional.rotate(img, 90)
flip = T.functional.hflip(img)

results = model.predict(rot)
results.show()
```

You'll observe:

- Translation: boxes shift consistently.
- Rotation: detections often fail unless trained for rotations.
- Horizontal flips: mostly robust.

---

2 | When YOLO is a good fit (and when it's not)

2.1 supervised vs. unsupervised approaches

Good fit (supervised)

- You can **name** and **annotate** recurring defect classes (e.g., *scratch, cut, loose thread, stitch skip, edge wear, crease, hole, stain (oil/ink/water), discoloration*).
- You can capture **enough labeled examples** per class (≥200–500 instances/class as a rule of thumb; more for high intra-class variability).
- You need **real-time** or near-real-time inference on edge devices (YOLO excels at speed).

Less ideal (unsupervised/anomaly)

- True *rare anomaly* scenario: each defect is unique, labels are scarce/expensive, and background materials vary a lot.
- In those cases, add (or even start with) an **anomaly/novelty detector** (e.g., PaDiM, PatchCore, SPADE, DRAEM, teacher-student) to produce a heatmap of *"anything unusual vs golden"*, and then use YOLO (or a lightweight classifier/segmenter) as a second stage for categorization/severity.

> *Practical recommendation: **Two-stage hybrid** Stage A: Unsupervised anomaly heatmap (few hundred "good" images) → candidate regions. Stage B: Supervised YOLO (detect/segment) on mined ROIs for the defects you care about (taxonomy below). This yields high recall on unknowns + stable precision on known classes.*

2.2 | Data acquisition: optics & illumination (critical here)

Small, low-contrast surface defects are often **photometry-limited** more than model-limited.

- **Resolution & scale**
  - Target a ground sampling of ≈ **0.03–0.10 mm/pixel** for micro-scratches; for coarse stains you can relax to 0.2–0.4 mm/pixel.
  - If full product images are large (e.g., handbags), **tile** at inference: 640–1024 px tiles with **25–40% overlap**.
- **Illumination geometry**
  - **Raking light** (low grazing angle) to reveal relief (scratches, embossing defects).
  - **Cross-polarization** to suppress specular glare on glossy leather; **co-polarization** to enhance specular defects if needed.
  - **UV/near-UV** excitation for certain stains (organic residues, optical brighteners on textiles).
  - Keep **lighting fixed and repeatable** (integrate light domes or controlled bars; constant exposure/white balance).
- **Capture protocol**
  - Multi-view or **multi-illumination bursts** per area (e.g., two raking angles + one diffuse), then fuse (max/mean or learnable fusion).
  - Calibrate **mm/pixel**: later you'll convert bbox/seg areas to physical size for **severity grading**.

2.3 | Taxonomy & annotation strategy

Define a **controlled vocabulary** with **visual criteria** and **severity grades**:

- **Defect classes** (examples):
  - Leather: `scratch`, `cut`, `wrinkle/crease`, `edge_wear`, `loose_thread`, `stitch_skip`, `hole/puncture`, `dye_bleed`, `discoloration`, `emboss_mismatch`.
  - Textiles: `oil_stain`, `water_stain`, `ink_mark`, `weft_defect`, `warp_defect`, `pilling`, `snag`, `seam_defect`.
- **Shapes**
  - Use **instance segmentation** (e.g., YOLOv8-seg) when the *shape/area* matters (stains, irregular wear).
  - Pure detection (boxes) is fine for long, thin scratches if you also store an **oriented box** (see below) or post-fit a thin mask.
- **Oriented boxes**

    - For elongated scratches, oriented detection (angle θ) improves localization and robustness. Consider an **OBB head** (or regress θ downstream from the mask).
- **Annotation policy**
    - For stains: **mask** the actual stain extent; for scratches: either mask or oriented box.
    - Annotate **only visible defects** at the chosen illumination; keep a *notes* field for ambiguous/low-contrast cases (helps reviewer agreement).

2.4 | Model choices (YOLO variants & companions)

- **YOLOv8 (or v9) detect/seg** in PyTorch for ease and ecosystem.
    - **Detect head** for small & medium defects; **Seg head** for stain extent.
    - If many tiny defects: choose **higher-res backbones** and keep stride 8/16 heads active.
- **Small-object emphasis**
    - Use **larger input sizes** (1024–1536), **tiling**, and **autoanchor** (if anchors are used) tuned to your defect size distribution.
    - Consider **Anchor-free** decoupled heads (YOLOv8) which often simplify small-object training.
- **Hybrid anomaly front-end** (optional but recommended for luxury QA)
    - PaDiM/PatchCore/Teacher-Student to propose ROIs and heatmaps; pass ROIs to YOLO for classification/segmentation + severity.

2.5 | Training recipe (supervised YOLO)

Hyperparameters (starting point)

- Input size: **1024** (train multi-scale 0.8–1.2).
- Optimizer: **SGD** (momentum 0.937) or **AdamW** (weight decay 0.01).
- Batch: as large as VRAM allows (accumulate if needed).
- LR schedule: **cosine** or step decay; warmup 3–5 epochs.
- Epochs: **100–300**, stop on plateau of val **mAP@50** and **Recall** (recall matters in QC).

Losses & imbalance

- **Focal loss** or class weights if long-tail (rare defect classes).
- Box loss: **CIoU/DIoU**; Seg loss: BCE + Dice (for stain masks).
- Tune $\lambda_{\text{obj}}, \lambda_{\text{cls}}, \lambda_{\text{box}}$ to favor **recall** (catch all defects) and let **NMS** clean up.

Augmentations (be careful here)

- Photometric: moderate **brightness/contrast**, small **gamma**, light **color-jitter** (leather hue is a cue → don't overdo).
- Geometric: **small rotations** ($\pm 10°$), **H-flip** if class semantics allow; avoid V-flip for gravity-aligned cues.
- **Mosaic/CutMix**: useful for YOLO, but **limit** for micro-defects (they can get destroyed). Try: mosaic prob 0.2–0.3, cutmix 0–0.1.
- **Blur/Sharpen**: light Gaussian blur helps robustness; avoid heavy blur that erases hairline scratches.
- **Specular simulations**: if possible, add *synthetic specular streaks* or use domain-randomized highlights to encourage robustness to glare.

Validation protocol

- **Stratified splits by product line/material/batch** to measure domain generalization.
- Metrics: report **Recall@IoU=0.5**, **mAP@50**, **mAP@50–95**, and for segmentation **mIoU**. For QC, track **per-class miss rate** (false negatives) and **overkill** (false positives).

2.6 | Inference at scale

- **Tiling**: 640–1024 tiles with **25–40% overlap**, then merge boxes/masks (NMS across tiles).
- **Throughput**: YOLOv8n/s can hit **>60 FPS** on 640 px tiles on modern GPUs; for 1024 tiling expect 10–30 FPS depending on model size.
- **Batch inference** on images from multi-illumination bursts; fuse decisions (logical OR for recall, or learned fusion for precision).
- **Post-processing & severity**
  - Convert boxes/masks to **mm** via calibration; compute **length, width, area**.
  - Derive **severity score** S (example):

$$S = \alpha, \mathrm{length(mm)} + \beta, \mathrm{width(mm)} + \gamma, \mathrm{contrast} + \delta, \mathrm{location\_weight} \quad (3)$$

  Map S to **AQL levels** or pass/fail rules per SKU.

2.7 | Robustness & invariances for your case

- **Translation**: strong; tiling makes it effectively invariant across the field.
- **Rotation**: leather grain and stitches have orientation; use **small rotations** in augmentation. If scratches are arbitrarily oriented, consider **oriented heads** or post-fitting line segments to masks.
- **Reflection (mirror)**: typically fine for surface defects, but **text/logo** defects are *not* mirror-invariant—treat as separate classes if needed.
- **Scale**: handled via pyramid/tiling; ensure your **anchor sizes** (if used) match the small-defect priors.

2.8 | Practical pitfalls (seen often)

- **Under-resolved defects**: camera/lighting can make hairline scratches invisible—no model will recover what the sensor doesn't see. Fix optics first.
- **Annotation inconsistency**: different labelers draw *different extents* for the same stain → noisy training. Provide **tight guidelines** and **review samples** weekly.
- **Over-augmentation**: excessive mosaic/blur/color jitter can erase subtle cues.
- **Domain shift**: new leather finishes, colors, embossing → performance drop. Mitigate with **continual learning** and **periodic fine-tuning**.

2.9 | Suggested minimal POC plan (4–6 weeks)

1. **Design of experiments** for capture: 2–3 lighting setups, 1–2 cameras, tile size & overlap grid.
2. **Collect** ~1–2k images across **≥6 product SKUs**, annotate 8–10 defect classes + masks for stains.
3. Train **YOLOv8-seg (m or s)** at 1024, light augmentations; measure per-class Recall@0.5.
4. If recall on rare defects <90%, add **anomaly stage** (PaDiM/PatchCore) to propose ROIs, retrain YOLO on mined ROIs.

5. **Calibrate severity** in mm and integrate **AQL** thresholds; pilot on a **blind batch** of new items.

2.10 | Deliverables to industrialize

- **Model(s)**: YOLO detect/seg + optional anomaly front-end.
- **Labeling handbook** (PDF): rules, examples, edge cases.
- **Optics dossier**: exact light geometry, polarization, exposure, mm/pixel.
- **Acceptance tests**: per-class recall/precision on a locked validation set from future batches.
- **MLOps**: versioned datasets, periodic re-training, drift monitoring (CPC/InfoNCE or simple embedding drift), traceable model cards.

If you want, I can draft:

- a **dataset YAML** with a proposed defect taxonomy,
- a **YOLOv8 training config** tuned for small defects, and
- a **tiling & fusion** inference script (PyTorch) with severity scoring in mm.

---

3 | Templates

3.1 | Dataset spec (YAML) — `defects_luxe.yaml`

Intent: a controlled taxonomy for leather & textile defects, paths, and meta needed by Ultralytics (YOLOv8/9). *Note:* keep class names lowercase, _ separators; you can prune/extend later.

```yaml
# defects_luxe.yaml
# Dataset spec for YOLOv8/YOLOv9 (detect or seg)
# Replace [PATH]/... with your actual folders.

# ├── datasets/
# │   ├── defects_luxe/
# │   │   ├── images/
# │   │   │   ├── train/*.jpg|*.png
# │   │   │   ├── val/*.jpg|*.png
# │   │   │   └── test/*.jpg|*.png
# │   │   └── labels/
# │   │       ├── train/*.txt (YOLO format) or *.json for COCO
# │   │       ├── val/*.txt
# │   │       └── test/*.txt

path: [PATH]/datasets/defects_luxe

train: images/train
val: images/val
test: images/test  # optional

# If you use instance segmentation, labels must be in the YOLO-seg
format (polygons).
# If you use detection only, standard YOLO bbox format is enough.

# Classes (taxonomy v0.1)
names:
  0: scratch
  1: cut
```

```yaml
  2: wrinkle_crease
  3: edge_wear
  4: loose_thread
  5: stitch_skip
  6: hole_puncture
  7: dye_bleed
  8: discoloration
  9: oil_stain
  10: water_stain
  11: ink_mark
  12: weft_defect
  13: warp_defect
  14: snag
  15: pilling
  16: seam_defect

# Optional dataset-level metadata (consumed by your own code)
robustness:
  mm_per_pixel_nominal: 0.06     # example; set per SKU if needed
  capture_modalities: [raking_left, raking_right, diffuse]    #
multi-illum burst
  polarization: [cross, co]
  notes: >
    Images captured with fixed light geometry. If multiple bursts
exist,
    store them in parallel folders and fuse at inference.

# Splits can be stratified by SKU/material/batch in your data
prep;
# keep a CSV mapping if you need per-SKU metrics later.
```

For **segmentation**, annotate masks (polygons) for stains and irregular wear (e.g., `oil_stain`, `water_stain`, `discoloration`), optionally also for `scratch` if you want shape features; for **detection-only**, use tight bboxes. If you later need orientation, we can add an oriented-box head (OBB) or fit line segments post hoc.

3.2 Small-defect–oriented training config — `yolo_defects_train.yaml`

Intent: a starting recipe tuned for small, subtle defects (higher input size, gentle augs, recall-friendly losses). Use with:

```
pip install ultralytics
yolo task=segment mode=train data=defects_luxe.yaml model=yolov8m-
seg.pt \
     imgsz=1024 batch=16 epochs=200 optimizer=AdamW lr0=0.001 \
     project=runs_defects name=v0_1 cfg=yolo_defects_train.yaml
```

(Change `task=detect` and `model=yolov8m.pt` if you don't do segmentation.)

```yaml
# yolo_defects_train.yaml  (Ultralytics overrides)

# --- Model/Task ---
task: segment              # or 'detect'
model: yolov8m-seg.pt      # start from pretrained COCO; consider -l
(lite) for edge
imgsz: 1024                # small-defect emphasis; can try 1280
later
batch: 16                  # fit to VRAM; use accum if needed (e.g.,
accumulate=2)
```

```yaml
epochs: 200
patience: 50              # early stop patience
device: 0

# --- Optim & Sched ---
optimizer: AdamW
lr0: 0.001
lrf: 0.01                 # cosine final LR factor
momentum: 0.937
weight_decay: 0.01
warmup_epochs: 3.0
warmup_momentum: 0.8
warmup_bias_lr: 0.05

# --- Loss weights (favor recall on objects) ---
box: 7.5                  # λ_box
cls: 0.5                  # λ_cls  (keep low if many fine-grained classes)
dfl: 1.5                  # distribution focal loss (for v8)
fl_gamma: 1.5             # focal gamma (mitigate long-tail)
seg: 1.0                  # seg head λ (when task=segment)
# For seg, Ultralytics combines BCE + Dice internally.

# --- Augmentations (gentle; do not destroy micro-cues) ---
hsv_h: 0.01
hsv_s: 0.4
hsv_v: 0.2
degrees: 8.0
translate: 0.07
scale: 0.15
shear: 0.0
perspective: 0.0
flipud: 0.0               # vertical flip off (gravity cues)
fliplr: 0.5               # horiz flip ok if semantics allow
mosaic: 0.25              # keep low; can erase micro-defects if too high
mixup: 0.05
copy_paste: 0.0           # segmentation copy-paste tends to look unrealistic for stains
blur: 0.3                 # light Gaussian blur
noise: 0.02               # slight
erasing: 0.0

# --- Data ---
workers: 8
pretrained: true
val: true

# --- Anchors / Head ---
# Using YOLOv8 anchor-free decoupled head by default.
# If you switch to anchor-based, run autoanchor on your dataset.

# --- Multi-scale ---
multi_scale: true         # internal 0.8–1.2 by default

# --- Metrics of interest (tracked in your own eval) ---
# mAP50, mAP50-95, Recall are default; also track per-class miss rate.
```

Notes & knobs to try later

- If tiniest defects are still missed, try `imgsz=1280` or `yolov8l-seg.pt`.
- If you must keep 1024, increase **tiling/overlap** at inference (see script below).
- For severe class imbalance, use `--class_weights` (Ultralytics supports automatic class weighting) or curate batches with minority oversampling.

3.3 | Tiling + fusion inference with severity in mm — `infer_tiled_severity.py`

Intent: robust detection/segmentation on large images with small defects; overlap-tiling, cross-tile NMS, mm-unit severity scoring, multi-illumination fusion (optional).

```
# infer_tiled_severity.py
# Requirements: ultralytics, torch, torchvision, numpy, opencv-
python, shapely (optional), pillow
# Run:
#   python infer_tiled_severity.py \
#       --weights runs_defects/v0_1/weights/best.pt \
#       --source /path/to/large_images \
#       --save_dir outputs \
#       --imgsz 1024 --tile 1024 --overlap 0.35 \
#       --mm_per_pixel 0.06 \
#       --task segment   # or 'detect'

import os
import math
import argparse
import glob
import numpy as np
import cv2
import torch
from ultralytics import YOLO
from torchvision.ops import nms

# ----------------------------
# Helpers
# ----------------------------

def tile_coords(H, W, tile, overlap):
    """Yield (x0, y0, x1, y1) crop boxes covering the image with
given overlap."""
    stride = int(tile * (1 - overlap))
    xs = list(range(0, max(W - tile, 0) + 1, stride)) or [0]
    ys = list(range(0, max(H - tile, 0) + 1, stride)) or [0]
    if xs[-1] != W - tile: xs.append(max(W - tile, 0))
    if ys[-1] != H - tile: ys.append(max(H - tile, 0))
    for y in ys:
        for x in xs:
            yield (x, y, x + tile, y + tile)

def clip_box(box, W, H):
    x1, y1, x2, y2 = box
    return [max(0, x1), max(0, y1), min(W - 1, x2), min(H - 1,
y2)]

def merge_dets_across_tiles(boxes, scores, labels, iou_thr=0.5):
    """Global NMS across all tiles. boxes Nx4 (xyxy), labels Nx1
int."""
```

```
    if len(boxes) == 0:
        return boxes, scores, labels
    boxes_t = torch.tensor(boxes, dtype=torch.float32)
    scores_t = torch.tensor(scores, dtype=torch.float32)
    labels_t = torch.tensor(labels, dtype=torch.int64)

    keep_all = []
    # Class-wise NMS
    for c in torch.unique(labels_t):
        idx = (labels_t == c).nonzero(as_tuple=False).squeeze(1)
        if idx.numel() == 0:
            continue
        keep_idx = nms(boxes_t[idx], scores_t[idx], iou_thr)
        keep_all.append(idx[keep_idx])

    keep = torch.cat(keep_all).cpu().numpy() if keep_all else
np.array([], dtype=int)
    return [boxes[i] for i in keep], [scores[i] for i in keep],
[labels[i] for i in keep]

def poly_area_mm2(poly_xy, mm_per_pixel):
    """Polygon area in mm^2; poly_xy: Nx2 in pixels."""
    if poly_xy is None or len(poly_xy) < 3:
        return 0.0
    x = poly_xy[:, 0]; y = poly_xy[:, 1]
    area_px = 0.5 * np.abs(np.dot(x, np.roll(y, -1)) - np.dot(y,
np.roll(x, -1)))
    return (area_px * (mm_per_pixel ** 2))

def severity_score(box_xyxy, cls_name, contrast=1.0,
location_weight=1.0, mm_per_pixel=0.06):
    """Example severity function S; customize coefficients per
SKU/spec."""
    x1, y1, x2, y2 = box_xyxy
    length_px = max((x2 - x1), (y2 - y1))
    width_px  = min((x2 - x1), (y2 - y1))
    length_mm = length_px * mm_per_pixel
    width_mm  = width_px * mm_per_pixel

    # default coefficients; consider a per-class table
    alpha, beta, gamma, delta = 1.0, 0.5, 0.75, 0.5

    if cls_name in {"oil_stain", "water_stain", "ink_mark",
"discoloration"}:
        # stains: area matters more; treat width ~ diameter proxy
        S = 0.3 * length_mm + 0.7 * width_mm + gamma * contrast +
delta * location_weight
    elif cls_name in {"scratch"}:
        # scratches: length dominant
        S = 1.2 * length_mm + 0.3 * width_mm + gamma * contrast +
delta * location_weight
    else:
        # generic
        S = alpha * length_mm + beta * width_mm + gamma * contrast
+ delta * location_weight

    return float(S)
```

```
# ----------------------------
# Main inference
# ----------------------------

def run(weights, source, save_dir, imgsz=1024, tile=1024,
overlap=0.35, iou=0.5,
        conf=0.25, mm_per_pixel=0.06, task="segment"):
    os.makedirs(save_dir, exist_ok=True)
    model = YOLO(weights)

    exts = ("*.jpg", "*.jpeg", "*.png", "*.bmp", "*.tif",
"*.tiff")
    files = [p for e in exts for p in
glob.glob(os.path.join(source, e))]
    if not files:
        raise FileNotFoundError(f"No images in: {source}")

    names = model.names  # class id -> name

    for path in files:
        im = cv2.imread(path, cv2.IMREAD_COLOR)
        if im is None:
            print(f"[WARN] Cannot read {path}")
            continue
        H, W = im.shape[:2]

        all_boxes, all_scores, all_labels, all_polys = [], [], [],
[]

        # tile over image
        for (x0, y0, x1, y1) in tile_coords(H, W, tile, overlap):
            crop = im[y0:y1, x0:x1]
            # Predict on crop
            # Speed: use stream=True and iterate; here we call
predict directly for clarity
            results = model.predict(crop, imgsz=imgsz, conf=conf,
verbose=False)

            for r in results:
                if task == "segment" and r.masks is not None:
                    # Segmentation results
                    # r.boxes.xyxy: (N,4), r.boxes.conf: (N,),
r.boxes.cls: (N,)
                    for j in range(len(r.boxes)):
                        bx = r.boxes.xyxy[j].cpu().numpy()
                        sc = float(r.boxes.conf[j].item())
                        lb = int(r.boxes.cls[j].item())

                        # shift to full-image coords
                        bx_full = [bx[0] + x0, bx[1] + y0, bx[2] +
x0, bx[3] + y0]
                        bx_full = clip_box(bx_full, W, H)
                        all_boxes.append(bx_full);
all_scores.append(sc); all_labels.append(lb)

                        # polygon (in crop coords) → shift
                        poly = r.masks.xyn[j].cpu().numpy() *
np.array([crop.shape[1], crop.shape[0]])
```

```
                        poly[:, 0] += x0; poly[:, 1] += y0
                        all_polys.append(poly.astype(np.float32))
                else:
                    # Detection results
                    for j in range(len(r.boxes)):
                        bx = r.boxes.xyxy[j].cpu().numpy()
                        sc = float(r.boxes.conf[j].item())
                        lb = int(r.boxes.cls[j].item())
                        bx_full = [bx[0] + x0, bx[1] + y0, bx[2] +
x0, bx[3] + y0]
                        bx_full = clip_box(bx_full, W, H)
                        all_boxes.append(bx_full);
all_scores.append(sc); all_labels.append(lb)
                        all_polys.append(None)

        # global NMS across tiles
        mboxes, mscores, mlabels =
merge_dets_across_tiles(all_boxes, all_scores, all_labels,
iou_thr=iou)

        # draw & severity
        vis = im.copy()
        out_records = []
        for k, (bx, sc, lb) in enumerate(zip(mboxes, mscores,
mlabels)):
            x1, y1, x2, y2 = map(int, bx)
            cls_name = names[lb] if isinstance(names, dict) else
str(lb)

            # Contrast proxy (simple local stddev on gray crop)
            patch = cv2.cvtColor(vis[y1:y2, x1:x2],
cv2.COLOR_BGR2GRAY) if (y2>y1 and x2>x1) else None
            contrast = float(patch.std()) if patch is not None and
patch.size>0 else 1.0

            S = severity_score(bx, cls_name, contrast=contrast,
location_weight=1.0,
                               mm_per_pixel=mm_per_pixel)

            cv2.rectangle(vis, (x1, y1), (x2, y2), (0, 255, 0), 2)
            cv2.putText(vis, f"{cls_name} {sc:.2f} S={S:.1f}",
                        (x1, max(10, y1 - 6)),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (10, 240, 10), 1, cv2.LINE_AA)

            # segmentation area in mm^2 if available (optional)
            area_mm2 = None
            if task == "segment":
                # find any poly that fits within this box with
similar class (heuristic)
                # (For strict association, track indices during
merge; omitted for brevity.)
                area_mm2 = None

            out_records.append({
                "bbox_xyxy_px": [x1, y1, x2, y2],
                "class_id": int(lb),
                "class_name": cls_name,
                "confidence": float(sc),
```

```
                "severity_S": float(S),
                "mm_per_pixel": float(mm_per_pixel),
                "area_mm2": area_mm2
            })

        # save visualization
        base = os.path.splitext(os.path.basename(path))[0]
        out_img = os.path.join(save_dir, f"{base}_det.png")
        cv2.imwrite(out_img, vis)

        # save json of detections
        try:
            import json
            with open(os.path.join(save_dir, f"{base}_det.json"),
"w") as f:
                json.dump(out_records, f, indent=2)
        except Exception as e:
            print(f"[WARN] cannot write json: {e}")

        print(f"[OK] {path} -> {out_img}  (n={len(out_records)})")

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--weights", type=str, required=True)
    ap.add_argument("--source", type=str, required=True)
    ap.add_argument("--save_dir", type=str, default="outputs")
    ap.add_argument("--imgsz", type=int, default=1024)
    ap.add_argument("--tile", type=int, default=1024)
    ap.add_argument("--overlap", type=float, default=0.35)
    ap.add_argument("--iou", type=float, default=0.5)
    ap.add_argument("--conf", type=float, default=0.25)
    ap.add_argument("--mm_per_pixel", type=float, default=0.06)
    ap.add_argument("--task", type=str, default="segment",
choices=["segment", "detect"])
    args = ap.parse_args()

    run(args.weights, args.source, args.save_dir, args.imgsz,
args.tile, args.overlap,
        args.iou, args.conf, args.mm_per_pixel, args.task)
```

> ⓘ **Note**
>
> - **Tiling**: default `tile=1024`, `overlap=0.35` (increase overlap for micro-defects).
> - **Global NMS**: merges all tile detections per class (`IoU=0.5` default).
> - **Severity**: simple linear form using length/width (mm) + contrast proxy. Replace with your official AQL mapping:
>
> $$S = \alpha, \mathrm{length(mm)} + \beta, \mathrm{width(mm)} + \gamma, \mathrm{contrast} + \delta, \mathrm{location\_weight} \quad (4)$$
>
> - **mm/pixel**: pass `--mm_per_pixel` from your calibration; for multi-SKU, maintain a per-SKU table.
> - **Multi-illumination bursts**: run the script across each illumination folder and **fuse JSONs** with a rule like OR-on-presence or score pooling (can add a second pass to merge by IoU across illuminations).

Quick-start commands

Train (segmentation)

```
yolo task=segment mode=train data=defects_luxe.yaml model=yolov8m-seg.pt \
     imgsz=1024 batch=16 epochs=200 optimizer=AdamW lr0=0.001 \
     project=runs_defects name=v0_1 cfg=yolo_defects_train.yaml
```

Infer (tiled, severity)

```
python infer_tiled_severity.py \
  --weights runs_defects/v0_1/weights/best.pt \
  --source [PATH]/qa_batch_images \
  --save_dir outputs \
  --imgsz 1024 --tile 1024 --overlap 0.35 \
  --mm_per_pixel 0.06 \
  --task segment
```

---

4 | Possible iterations

- A **labeling handbook** (policy, examples, edge cases) with severity tables per class.
- **SKU-aware severity mapping** (location weighting by panel/zone).
- **Anomaly front-end** (PaDiM/PatchCore) that feeds ROIs into YOLO, with a fusion script.
- **Oriented scratch** head (OBB) or post-fit of line segments on masks for accurate length/width angle.
- A **metrics notebook** reporting per-class Recall@0.5, mAP, miss/overkill rates, and **size-wise** breakdown.