Project Book

Eagle Eyes

Client: *Nissha Medical Technologies*

Project: *Q-Block Detection + OK/NG Gating*

Version/Date: *17/12/2025 (Handoff)*

Team: *Affida*

Members: *Mudit Nautiyal (Team Lead), Aditya Aryan, Swarat Sarkar, Aarti Jadhav*

What this project is:

This system is created to inspect images of the ticket web and decide OK or NG (No Good) based on whether the **big Q-blocks** are detected consistently and meet the quality rules. The decision is made by a rule based "gate" on top of YOLO detections.

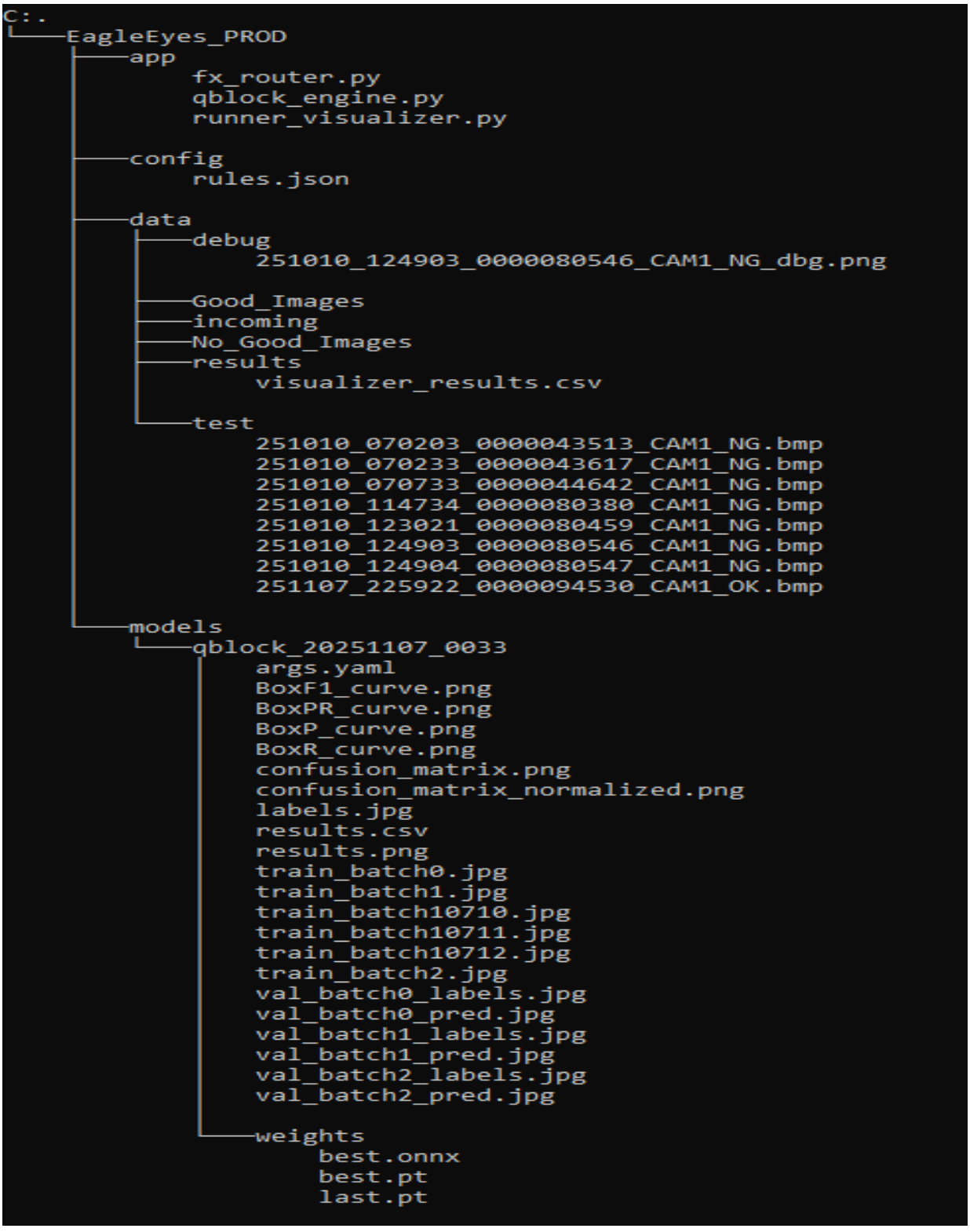
Primary goal in production:

- **High confidence gating** (avoid passing bad tickets).
- Consistent, repeatable decision logic with **clear failure reasons** per image.

Deliverables included

1. **Production Code**
    a. **app/qblock_engine.py:** code engine that loads model + rules, runs inferences, applies checks, returns a structured result dict.
    b. **app/runner_visualizer.py:** production runner that watches a folder (data/incoming/), process new images, shows an annotated dashboard window, appends a CSV log, and moves images to OK/NG respective folders i.e., Good_Images/No_Good_Images respectively.
    c. **app/fx_router.py:** Lightweight routing player that converts engine output into a unified action record (CSV logging + future PLC signal mapping). Not required for the visualizer run, but kept for integration.
2. **Configuration**
    a. **config/rules.json:** Single source of truth for model path + thresholds/ranges for count/visibility/density and inferences parameters.
3. **Runtime folders (I/O)**
    a. **data/incoming:** drop new images here.
    b. **data/Good_Images:** process OK images will appear here.
    c. **data/No_Good_Images:** processed NG images will appear here.
    d. **data/results/visualizer_results.csv:** per image results log (from the visualizer).
    e. **data/debug/:** debug render outputs (only if generated).
4. **Model artifacts**
    a. **Models/qblock_20251107_0033/weights/best.onnx:** production weight referenced in rules.json.
    b. **best.pt/last.pt + training plots:** training aritfacts kept for auditing/retraining reference.

Repository structure



Img 1: Repository Structure

How to run (production mode)

- **Environment assumptions**
  - Tested with **Python 3.11.**
  - Requires ultralytics, opencv-python, numpy.
  - For faster inferences, requires GPU.
- **Start the visualizer runner**
  - Run from the repo root (EagleEyes_PROD/): python app/runner_visualizer.py
  - **What happens:**
    - A window opens: “Eagle Eyes - Real-Time Visualizer”.
    - The program continuously scans **data/incoming/** every 0.5 seconds.
    - For each image found:
      - Engine evaluates it (OK/NG).
      - UI dashboard is displayed (status + boxes + timing).
      - A row is appended to data/results/visualizer_results.csv
      - The image is moved to data/Good_Images or data/No_Good_Images.

    To quit: press **q** in the OpenCV window.
- **Smoke test**
  - Copy any image to data/incoming/.
  - Confirm:
    - It gets moved to OK or NG folder.
    - CSV log gets a new row.
    - Dashboard displays the same decision.

System behavior and decision logic

- **What the engine returns (output contract)**

  QBlockEngine.evaluate_image(...) returns a dict with key fields:
  - **status:** OK or NG.
  - **big_count:** deduplicated big Q-block count used for gating.
  - **raw_big_count:** raw YOLO count before de-dup.
  - **count_ok, vis_big_ok, dens_big_ok, relpos_ok:** boolean check outcomes.
  - **failed_checks:** list of strings (reasons).
  - **For visualization:** xyxys, confs, clses, dedup_indices.

  This makes every decision auditable per image.
- **Inference strategy (two passes)**

  The engine runs **primary inference** first. It falls back to **recovery inference** in two cases:
  - If YOLO returns no boxes (empty detection).
  - If raw big deetctions are **below the minimum expected count**.

  From **rules.json** (current):
  - **Primary:** imgsz=1280, conf=0.05, iou=0.50, tta=False
  - **Recover:** imgsz=1280, conf=0.001, iou=0.45, tta=True

Why two passes are there:

- Primary is meant to be stable one.
- Recovery is meant to rescue the “hard” images (low contrast/blur etc..) but can increase false positives.

- **Class scope used in production**
  Even though the project has references to small_q_blocks in stats/rules history, the current engine runs YOLO with:
  - **Classes = [big_q_block only]**
  So production gating is based only on **big Q-block** detections.
- **Conservative de-dup (“hybrid stabilization”)**
  Before counting big blocks, the engine applies a conservative de-dup:
  - Sort by confidence (high → low).
  - Drop a box only if it is essentially the same block:
    - IoU >= 0.85
    - Area ratio within 0.8 to 1.25

  Purpose:
  - Reduce “double boxes on one Q-block” without deleting legitimate neighbors.
- **Checks used for OK/NG**
  Final decision:
  - OK only if all checks pass:
    - Count check
    - Visibility check
    - Density check
    - Relative position check
  - Otherwise NG and **failed_checks** lists the failing rules.

A) Count check (strict)

Current production policy: allow only these deduped big counts:

- **14** or **21** (configured in **rules.json** as **big_expected_any: [14, 21])**

This is a deliberate **hard fate** because count errors are strongly correlated with print/visibility issues.

B) Visibility check (confidence + area fraction)

For each detected big block, the engine checks:

- Confidence within the configured min/max range.
- Area fraction within the configured min/max range.

Tolerance exists:

- Base tolerance for big blocks = 1
- If the ticket is in the **21-block layout**, tolerance is relaxed to **3**.

This is meant to reduce false NG caused by 1-3 weak blocks when the overall set is consistent.

C) Density check (ink quality proxy)
For each detected block:

- Crop grayscale patch from bbox.
- Compute:
  - mean_gray (0 = black → 255=white)
  - dark_ratio = fraction of pixels darker than Otsu threshold.

The gate uses **median** acrossed detected blocks (robust against outliers), then checks whether the median falls inside the configured ranges.

D) Relative position check (layout sanity)
The current code computes:

- Median pairwise distance between detected centers (normalized by image diagonal).
- Orientation via SVD on the block centers.

Reports/Outputs produced

- **Visualizer log (main audit trail)**
  - **Path:** data/results/visualizer_results.csv

  Columns:

  - Timestamp
  - image_name
  - status
  - big_count
  - raw_big_count
  - count_ok
  - vis_big_ok
  - dens_big_ok
  - relpos_ok
  - failed_checks (semicolon separated)

This file is the easiest **handoff artifact** for performance review and debugging.

- **File routing (operator workflow)**
  - If status is OK → image is moved to data/Good_Images/
  - If status is NG → image is moved to data/No_Good_Images/
- **Router log (integration trail, optional)**
  If/when **FXRouter** is used:
  - Logs/results_log.csv with minimal fields:
    - Timestamp, image, status, big_count, failed_checks

Experiments tried this semester

Below is the working log of what was implemented and why it exists in the final system or removed from it.

ID	What was tried	Why	Where	Outcome	Why kept/dropped
E01	Single-image evaluation entrypoint	Fast debugging on one file	**qblock_engine.py** CLI usage	Worked	Kept as a minimal sanity tool
E02	Folder-watching runner prototypes	Needed automation on incoming images	Earlier runner variants (now removed)	Partial	Replaced by visualizer runner
E03	Real-time visualizer dashboard	Needed operator friendly live view + per-image audit	**runner_visualizer.py**	Worked	Kept as current production entrypoint
E04	Strict count gate (14 or 21 only)	Count is the strongest failure indicator	**rules.json** + engine logic	Worked	Kept to reduce false passes
E05	Recovery inference pass (low conf + TTA)	Handle “no-box / low contrast” cases	**rules.json** → **count_inference.recovery**	Mixed	Kept, but know to increase false positives
E06	Conservative de-dup (IoU 0.85 + area filter)	YOLO sometimes double-boxes the same block	**qblock_engine.py**	Worked	Kept; stabilizes count
E07	Visibility gate using conf + area fraction with tolerance	Reduce unstable passes and explain NG reasons	**visibility_thresholds**	Worked	Kept; supports explainable NG
E08	Ink density metric (mean_gray + dark_ratio)	Add print quality proxy beyond geometry	**density_thresholds** + engine crop	Partial	Kept but still a known weak point
E09	fx_router integration layer	Prepare PLC-ready mapping + unified action outputs	**fx_router.py**	Ready	Kept for future integration.

Table 1: List of Experiments

The experiments listed above are more of a relative ones, there were many more where a FasterViT (Vision based transformers) were tried, which are can be taken to a level further later.

How to safely change things (maintenance guide)

- **Updating the model**
  - Export/produce a new ONNX weight under **models/<new_run>/weights/best.onnx**
  - Update in **config/rules.json:**
    - **Meta.weights_path**
  - Run a validation pack (recommended) before deploying.
- **Updating thresholds**
  All gating thresholds live in **config/rules.json**. Safe workflow:
  - Change one group at a time (visibility OR density OR inference).
  - Run a known set of images (OK + NG).
  - Compare:
    - Pass rate changes.
    - Reason distribution (**failed_checks)**.
    - False pass risk vs false NG rate.

Appendix A - Quick “factory operator” instruction

1. Run: **python app/runner_visualizer.py**
2. Drop images into: **data/incoming/**
3. Watch status in the window.
4. OK images go to **data/Good_Images/**
5. NG images go to **data/No_Good_Images/**
6. Log is saved in **data/results/visualizer_results.csv**