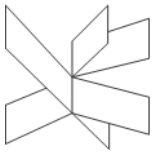


Object Classification in Logistics PoC – Project Report

Jack Thomas Spinola, 326238

Guilherme Caldeira Miranda, 326304

Michael Viuff

61,545 characters

Software Technology Engineering

6th Semester

27.05.2025

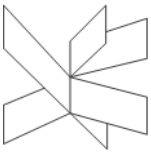
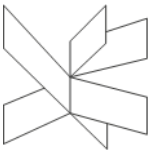
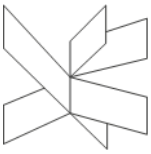


Table of content

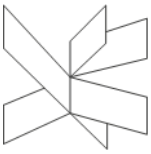




1. Abstract

This project addresses inefficiencies in logistics operations caused by the absence of automated systems for detecting fragile labels and classifying objects in real time. The primary objective is to develop a proof-of-concept (PoC) solution based on image recognition that can reliably identify fragile labels and distinguish between various package types within a logistics environment. YOLO (You Only Look Once) is employed as the core object detection framework due to its balance between accuracy and processing speed.

The system integrates annotated image data, a dedicated labeling tool, and deployment pipelines via Kubernetes, with outputs stored and visualized using Elasticsearch and Kibana. The development process follows Kanban methodology to ensure iterative progress and flexibility. As a PoC, the solution is designed to demonstrate the technical feasibility of achieving accurate, sub-second detection performance, suitable for future integration into large-scale logistics systems. Preliminary results show substantial improvements in detection accuracy and operational responsiveness, supporting the viability of real-time detection in dynamic warehouse environments.



2. Introduction

The efficient and timely flow of goods across vast networks, dependent on the logistics and transportation industries, is the center of trades. These infrastructure components are essential because they manage millions of packages under strict deadlines and high-volume expectations. (Group, 2023) Distribution centers and logistics depots are two key components of this infrastructure. Accurate and effective package management remains challenging despite automation advancements, particularly for fragile items.

A primary issue is the absence of a reliable and consistent mechanism for fragile label detection (Products, n.d.). Most current methods rely on manual visual inspection, which is laborious and prone to human error. This often leads to the improper handling of fragile products (Smith, 2022). Further complicating the uniform detection process and affecting overall operational accuracy is the range of label formats, which include differences in size, placement, design, and partial occlusion (Andersen, 2023).

Real-time object classification and label detection are taxing the logistics industry. Conventional techniques still have a poor ability to differentiate among tires, containers, and other irregular cargo types. Lack of real-time identification compromises handling, monitoring, and sorting processes, influencing operational effectiveness and customer satisfaction.

This work uses real-time image recognition to create an automated system that detects fragile labels and classifies objects, addressing these problems. The YOLO (You Only Look Once) (DataScientest, n.d.) object detection framework guarantees constant and quick performance in demanding surroundings. Strong development pipelines spanning labeled datasets, Kubernetes (Kubernetes, n.d.) deployment pipelines, ElasticSearch (Elastic, n.d.), and Kibana (Elastic, n.d.) visualizing tools support this system.

The significance of this undertaking is not limited to its technical capabilities. Academically, it contributes to the expanding corpus of research on applied machine learning in industrial environments. It provides a practical, scalable solution for improving logistics efficacy in the real

world. This endeavour is pertinent to various domains due to the economic pressure for faster and more accurate delivery systems and the social and technological trend toward more intelligent automation.

3. Main section

3.1 Methods

3.1.1 Knowledge Acquisition

The project's first stage was extensive knowledge acquisition intended to direct object detection system design and implementation. A comparative study of several YOLO (You Only Look Once) models was done using official documentation and repositories provided by Ultralytics. This report considered performance metrics, including detection accuracy, inference speed, and implementation support. Following this study, the YOLOv8 architecture (Torres, 2024) was selected as the most suitable framework because of its balance of accuracy, speed, and extensive documentation, which is especially practical for real-time logistics applications.

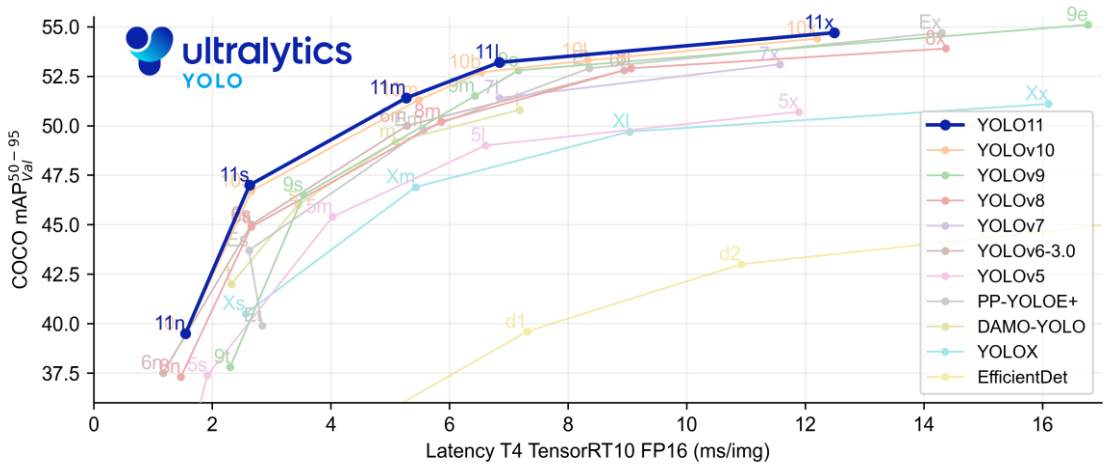


Figure 1 Yolo models

Additionally, this examined important technical components required for efficient model training. This included knowledge of annotations applied in object detection—more especially, the YOLO form based on bounding box coordinates scaled to image size (Ultralytics, 2023). Image augmentation techniques were also examined with an eye towards rotational, hue, and

other synthetic variations meant to improve generalisation during training, using Albumentations (Buslaev, 2020). When handling label occlusion, uneven lighting, and variation in package presentation, these methods are indispensable for improving the model.

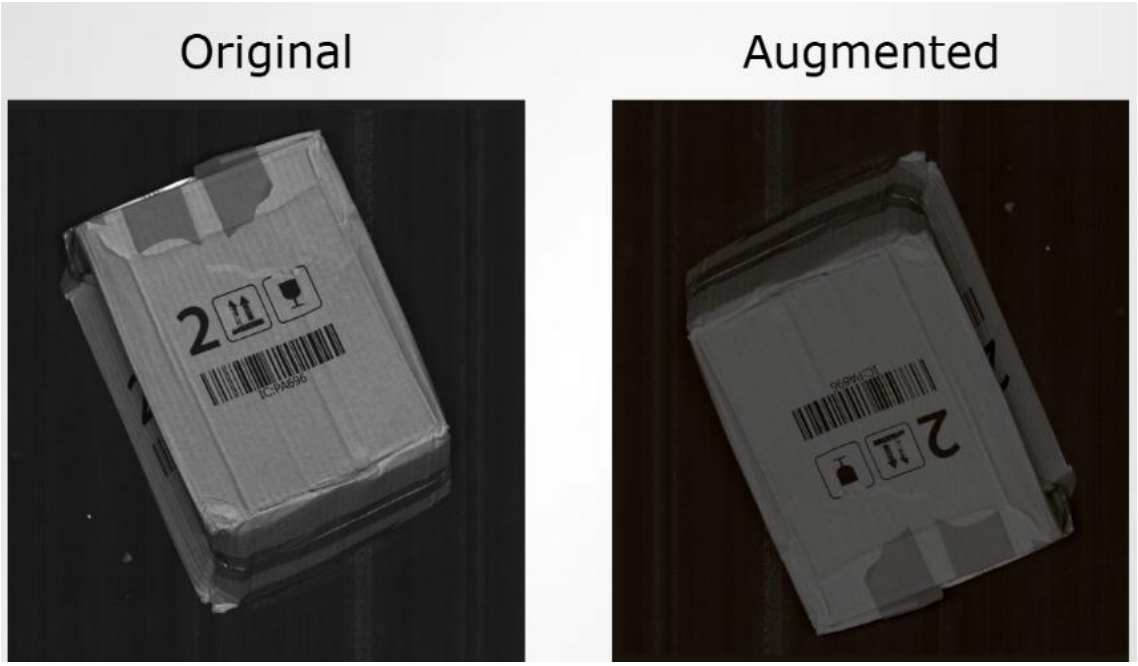


Figure 2 Example of image before and after augmentation

Active learning (GeeksForGeeks, n.d.) was another practical approach, particularly in saving the time required to mark vast volumes of data manually. Verifying the promise of active learning

to quickly refine datasets and more efficiently over iterative cycles improves model performance through informal feedback. This diagram demonstrates the active learning process:

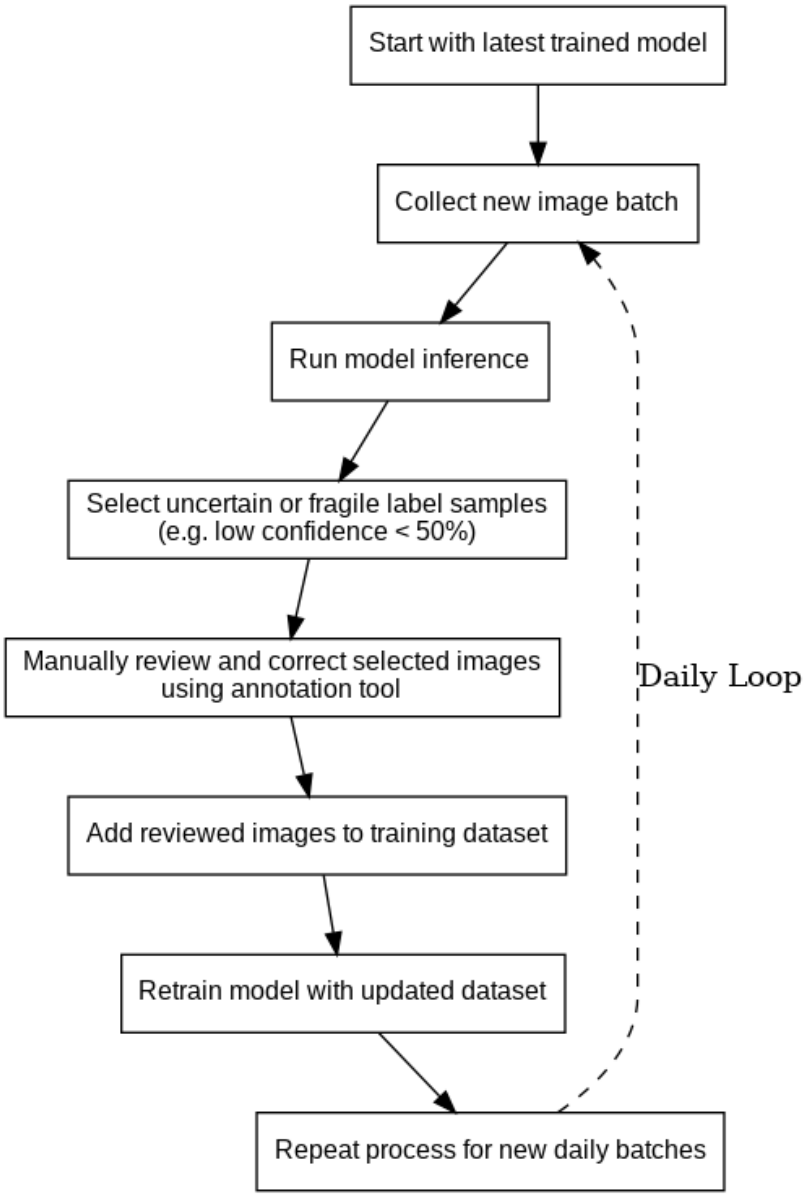


Figure 3 Active learning flow

Formal technical documentation combined with pragmatic knowledge from domain experts assured a strong foundation for the next phases of system development. This knowledge base significantly influenced the implementation strategy and design decisions made in the project.

3.1.2 System Analysis and Solution Design

3.1.2.1 Requirements

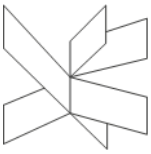
The system aims to enable real-time object detection and classification based on image inputs from a live feed camera, such as parcels, tires, and fragile labels. It is built to automate and standardize the recognition of important logistical elements to increase accuracy, lower manual inspection efforts, and support decision-making in high-volume operational environments.

Fundamentally, the system is a file-watching script located in a Kubernetes cluster set to track a directory getting images from a live image feed. When a new image is detected, the script sends it to an inference service located in a Kubernetes cluster driven by the YOLOv8 model. Object detection in the inference module finds fragile label, parcel, and tire instances.

Therefore, when the inference service receives an image, it outputs the results back to the file-watching script, where the output is serialized into a structured JSON object with metadata including the id of inference, image path, timestamp, if it is a parcel or fragile label, coordinates of the bounding box, the class id of the detected object, and confidence scores.

Once the output is serialized the file-watching script sends all the information to an Elasticsearch instance linked with Kibana for real-time data visualization.
Acting as the system's end users, warehouse workers interact with the data only via the Kibana dashboard. On this interface, key operational statistics include visual overlays for quality assurance, object quantities, and detection trends. Also needs to run entirely in real-time under a Kubernetes-managed infrastructure, this system guarantees scalability, fault tolerance, and effective use of resources.

The following requirements were established during the initial planning phase of the project, prior to the determination of specific tools, frameworks, or infrastructure described in subsequent sections. At that stage, the selection of technologies, such as the object detection



model, deployment platform, and data visualization tools, had not yet been finalized. Consequently, the requirements have been formulated in a way that emphasizes the system's intended functionality independently of implementation-specific details introduced later in the development process.

Functional requirements:

- FR1: The system should receive, and process images captured from a logistics warehouse environment.
- FR2: The system should detect when new images become available for processing.
- FR3: The system should classify detected objects into predefined categories, including parcels, tires, and fragile labels.
- FR4: The system should assign bounding boxes and confidence scores to each detected object.
- FR5: The system should serialize the detection output into a structured format containing image path, timestamp, object class, bounding box, confidence, and flags
- FR6: The system should transmit structured detection results to an Elasticsearch index for storage.
- FR6: The system should visualize detection statistics in real-time via a Kibana dashboard.
- FR7: The system should maintain and store the latest trained detection model used for inference.
- FR8: The system should log all inference operations, including errors, processing times, and system status.

Non-functional requirements:

- NFR1: The system should process each image and complete inference within a maximum latency of 1 second.

- NFR2: The entire system should be containerized and deployed in a Kubernetes cluster to support orchestration, scalability, and reliability.
- NFR3: The system should log performance metrics and operational events, including processing time and system status, for monitoring and traceability.
- NFR4: The Kibana dashboard should reflect newly received detection results in real time, without requiring manual refresh or user input.

3.2 Project Implementation

3.2.1 Data Preparation and Annotation

From a camera placed in an operational warehouse, the basis of the object detection system is a custom dataset consisting of real-world images of goods and tires. This camera configuration was set to send data to Beumer's centralized data center in real time, producing an ongoing stream of images. Along with tire units, the captured images show various parcels in a range of sizes and materials, including cardboard boxes, plastic wraps, paper coverings, and plastic bags.

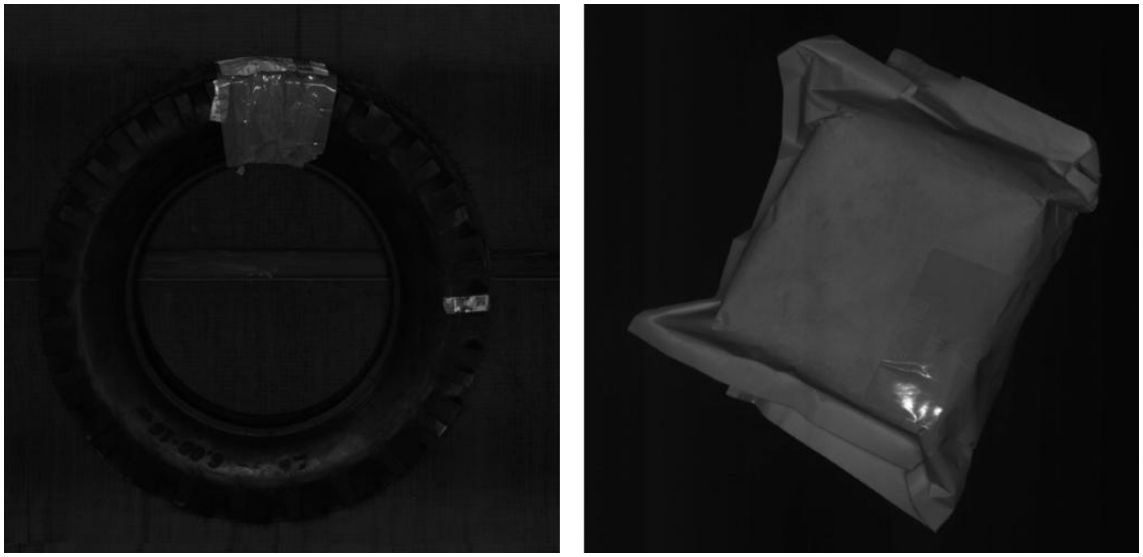


Figure 4 Tire & Parcel Image

All collected images were manually annotated using Label Studio, an open-source data labeling tool. This tool was selected due to its ease of integration and ability to simplify custom object identification processes. Each of the four categories, label, label_icon, parcel, and tire, is designated as a unique class identification, 0,1,2,3, respectively.

Bounding boxes were manually drawn around relevant objects in raw photos that were uploaded into Label Studio as part of the annotation process. This process, although it seemed simple, involved a lot of work because any bounding boxes not placed correctly would negatively impact the model in the future. Keyboard shortcuts allowed annotators to switch between classes and carefully adjust bounding boxes to ensure accurate labeling. After annotations were finished, Label Studio exported the labeled data into YOLO format, which was appropriate for the training pipeline. Specifically, the text files contained class IDs and normalized bounding box coordinates (*center_x, center_y, width, and height*) in relation to image dimensions.

dataset > train > labels > 0f5e0dae-03_0000000062861363_1_aug.txt

```
1 0.37073755264282227 0.24386918544769287 0.02750873565673828 0.04904627799987793
0 0.37039369344711304 0.2431880235671997 0.04195082187652588 0.07220709323883057
2 0.5340707078576088 0.515667587518692 0.4504556208848953 0.8215258717536926
```

Figure 5 YOLO format

This setup allowed for a consistent and efficient annotation process, especially when handling large amounts of warehouse photos. As the model was retrained in later development cycles, Label Studio made it possible to annotate and incorporate new images into the dataset through iterative dataset improvements.

All thought this process was time-consuming, it was mandatory to develop the initial models. Later, this process was replaced with active learning once the model could help label the data itself.

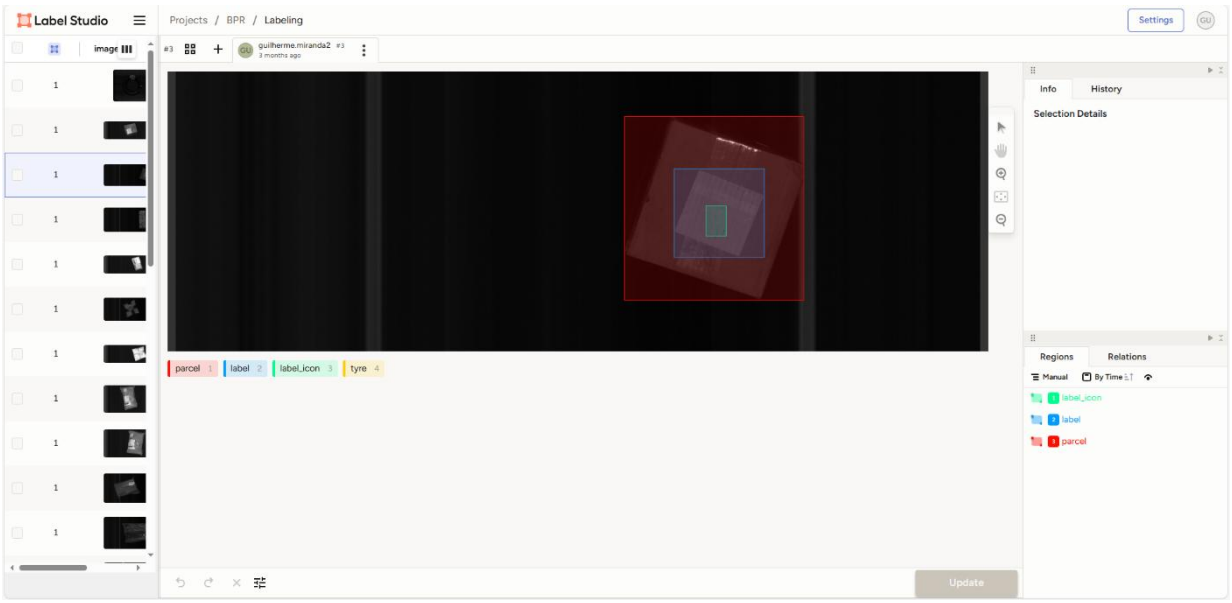


Figure 6 Label Studio

Preprocessing included data augmentation techniques to balance the dataset, particularly about the low frequency of fragile labels, which showed up roughly 1 in every 100 images. These augmentations aim to increase training sets fragile label variability by artificial representation. Images with fragile labels varied using geometric transformations, hue corrections, brightness changes, and image rotation. Despite their low frequency in the original dataset, this approach was crucial for raising the generalizing capacity and precise identification of fragile labels.

```
# Define the augmentation pipeline
transform = A.Compose([
    A.Rotate(limit=(180, 180), p=1.0),
    A.RandomBrightnessContrast(brightness_limit=(0.05, 0.1), contrast_limit=(0.05, 0.1), p=1.0),
    A.RandomGamma(gamma_limit=(90, 110), p=1.0),
], bbox_params=A.BboxParams(format='yolo', label_fields=['category_ids']))
```

Figure 7 Data augmentation code snippet

The dataset was divided into three subsets: 70% for training, 20% for validation, and 10% for testing. This split was performed with attention to class balance, ensuring that the model could

learn from a diverse and representative distribution of object types while being effectively validated and tested on unseen data.

The data preparation phase was critical to ensure the integrity and applicability of the resulting object detection model, laying the groundwork for accurate measures. The dataset was continuously updated and refined after each model version, incorporating new samples and corrections to address errors identified during evaluation, thereby supporting progressive improvements in detection accuracy.

3.2.2 Model Development and Training

3.2.2.1 Model Selection and Early Prototypes

The model development process began with the selection and comparison of various object detection frameworks suitable for real-time classification tasks.

The initial architecture examined was YOLOv8 from Ultralytics, which had good performance metrics, strong documentation, and rapid inference capabilities. However, due to licensing restrictions, evaluating alternative open-source solutions was the next step.

Detectron2 from *Meta AI Research* (Research, n.d.) and *YOLOv9* from *MultimediaTechLab* (MultimediaTechLab, n.d.) (an unofficial open-source implementation) were the two primary alternatives assessed. The absence of community support and documentation presented significant usability challenges for *YOLOv9* even though it was license compatible. Its practicality was further limited by suboptimal accuracy and integration challenges. *Detectron2*, despite its recognition in research contexts, was deemed unsuitable for this application. It demonstrated training rates up to five times slower than those of *YOLOv8* on the same dataset and generated less dependable predictions in visual validation and test metrics.

The evaluation criteria used to assess each framework included:

- **Detection accuracy** across each object class
- **Inference speed** and suitability for real-time operation
- **Training performance** on the available hardware
- **Ease of integration** with third-party tools (e.g. *MLFlow*)
- **Documentation quality** and community support

YOLOv8 is built upon a convolutional neural network architecture optimized for speed and accuracy, aligning with our project goals. It divides each input image into a grid and directly predicts bounding boxes and class probabilities in a single forward pass. This single-shot design reduces computational overhead, allowing for real-time inference. The model's backbone extracts, processes, and interprets visual features at multiple scales. This multi-scale detection capability is critical in logistics environments where objects like parcels and labels can vary significantly in size and visibility.

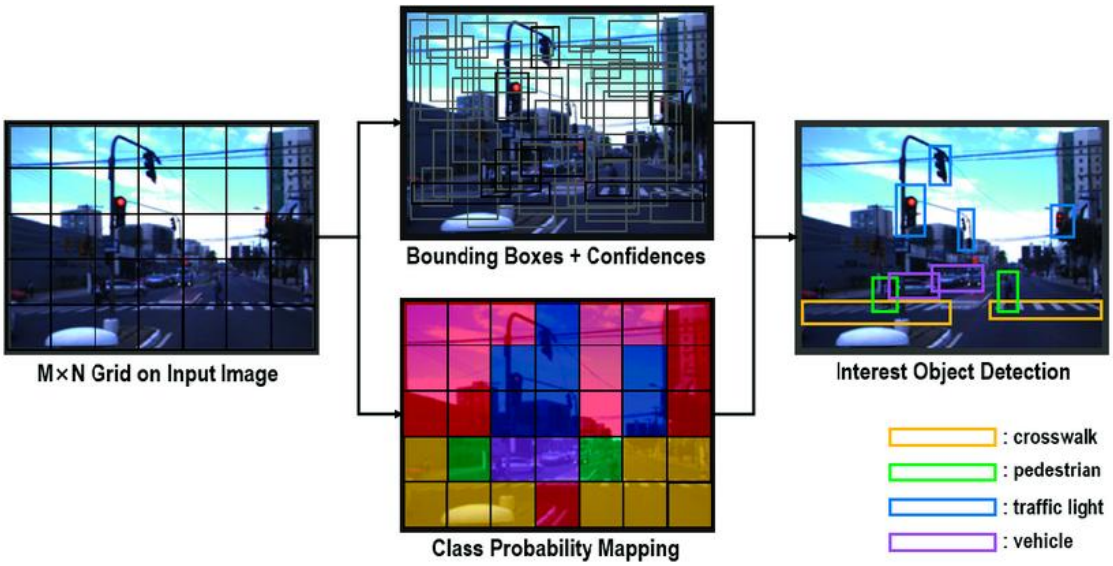


Figure 8 YOLO model

A total of six prototype models were built and tested. Alterations to hyperparameters, augmentation techniques and training data were implemented in each iteration. The models were assessed quantitatively using performance metrics such as precision and mean average precision (mAP) and qualitatively through visual inspection of consistent test images.

Mean Average Precision (mAP) is a widely accepted metric for evaluating the performance of object detection models. It quantifies both the classification accuracy and the spatial localization quality of predicted objects across all target classes. The Intersection over Union (IoU), which quantifies the overlap between a predicted bounding box and the matching ground truth bounding box, serves as the basis for mAP evaluation. The definition of IoU is:

$$IoU = \frac{Area\ of\ Union}{Area\ of\ Overlap}$$

A detection is considered correct, a true positive, if its IoU with the ground truth box exceeds a predefined threshold (commonly 0.5). Otherwise, the prediction is classified as a false positive or false negative, depending on context. This spatial criterion ensures that detections are not only present but also sufficiently accurate in position and scale.

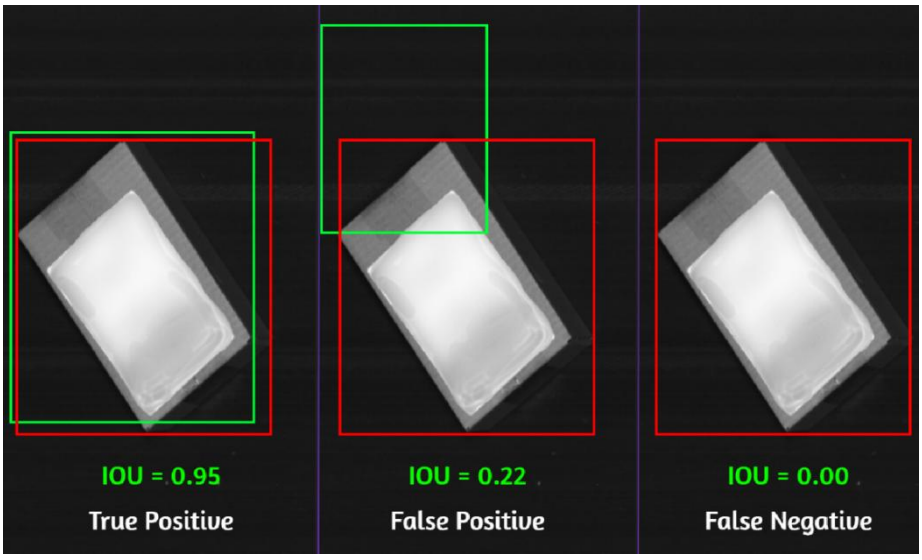


Figure 9 IOU example

Once detections are classified, the model's performance is evaluated using precision and recall:

- Precision quantifies the ratio of correctly predicted objects to all predicted objects

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

- Recall measures the ratio of correctly predicted objects to all ground truth objects

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

By adjusting the confidence threshold of predictions, multiple precision-recall pairs can be computed, forming a precision–recall (PR) curve.

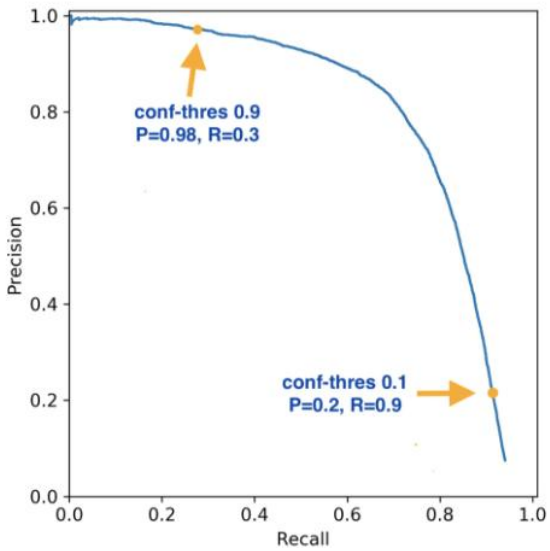
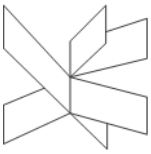


Figure 10 Precision–recall curve

For each object class, an Average Precision (AP) score is calculated as the area under the precision-recall curve. This value provides a representation of how well the model balances Precision and recall across varying thresholds. Average Precision reflects both the quality and consistency of detections for a specific class.



Two approaches were used in this project for model evaluation:

- *AP@0.5:* Uses a fixed IoU threshold of 0.5 for evaluation
- *AP@[0.5:0.95]:* Averages AP scores across multiple IoU thresholds (from 0.5 to 0.95 in increments of 0.05), providing a more strict and complete metric.

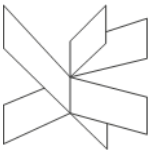
After computing Average Precision scores for each object class, the final mean Average Precision (mAP) is calculated as the arithmetic mean across all classes:

$$mAP = \frac{1}{N}\sum_{i=1}^{N} APi$$

Where N represents the total number of object classes, in this project, mAP was used to monitor detection performance across the *parcel, tire, label, and label_icon* classes.

The use of mAP was necessary in this project to quantify improvements during each model iteration. Evaluation using *mAP@0.5* and *mAP@[0.5:0.95]* allowed comparison across model versions, showing weaknesses in detection (e.g., missed or imprecise bounding boxes) and guiding dataset augmentation and model tuning. Consequently, mAP served not only as a performance metric but also as a feedback mechanism for the iterative refinement of the object detection system.

Even though mAP was a good metric to see while the model was training for around 5 hours, actually testing with images and visualizing the predictions was always favored. But this was only possible after training was complete. The mAP allowed, in some cases, to stop training early if it noted that the metrics were not meeting standards, implying that some issue was occurring.
An example of this was a new model on a recently prepared dataset. Despite proceeding with standard training procedures, the mAP values remained significantly the same as those of the previous model, showing no signs of improvement. Upon further investigation, it was discovered that transfer learning had not been applied, and the model was being trained from scratch. Transfer learning was correctly configured, and training was restarted. This observation saved



approximately 3 hours and demonstrated the practical utility of mAP as a diagnostic tool during model training and development.

This evaluation strategy made possible to monitor regression trends and incremental improvements. The model's performance was significantly enhanced when trained on a progressively diverse dataset, particularly in recognizing fragile labels almost imperceptible to the human eye.

The evolution of the model’s detection capabilities can be clearly observed by comparing the output of early prototypes with that of the final model. In the first prototype, the model consistently failed to detect fragile labels, particularly in cases where they were partially obscured, low contrast, or located at the periphery of the image. For example, in the image below (Figure 11), the fragile label is clearly visible to the human eye but was entirely missed by the model. This early limitation highlighted the need for more diverse training data and better label representation across different visual conditions.

Figure 11 First model prediction

In contrast, the final model, trained using over 1,200 images, including over 1,000 enhanced through active learning, demonstrated a much more refined ability to detect fragile labels, even in complex or degraded visual contexts. As shown in Figure 12, the same fragile label, which was missed in the earlier version, is now successfully detected with high confidence. This improvement underscores the effectiveness of the training process, particularly the contribution of manual re-labeling and real-time feedback from the active learning cycle. The contrast in model behavior before and after the full training loop illustrates not only improved accuracy but also the model's improved generalization to real-world package variability.

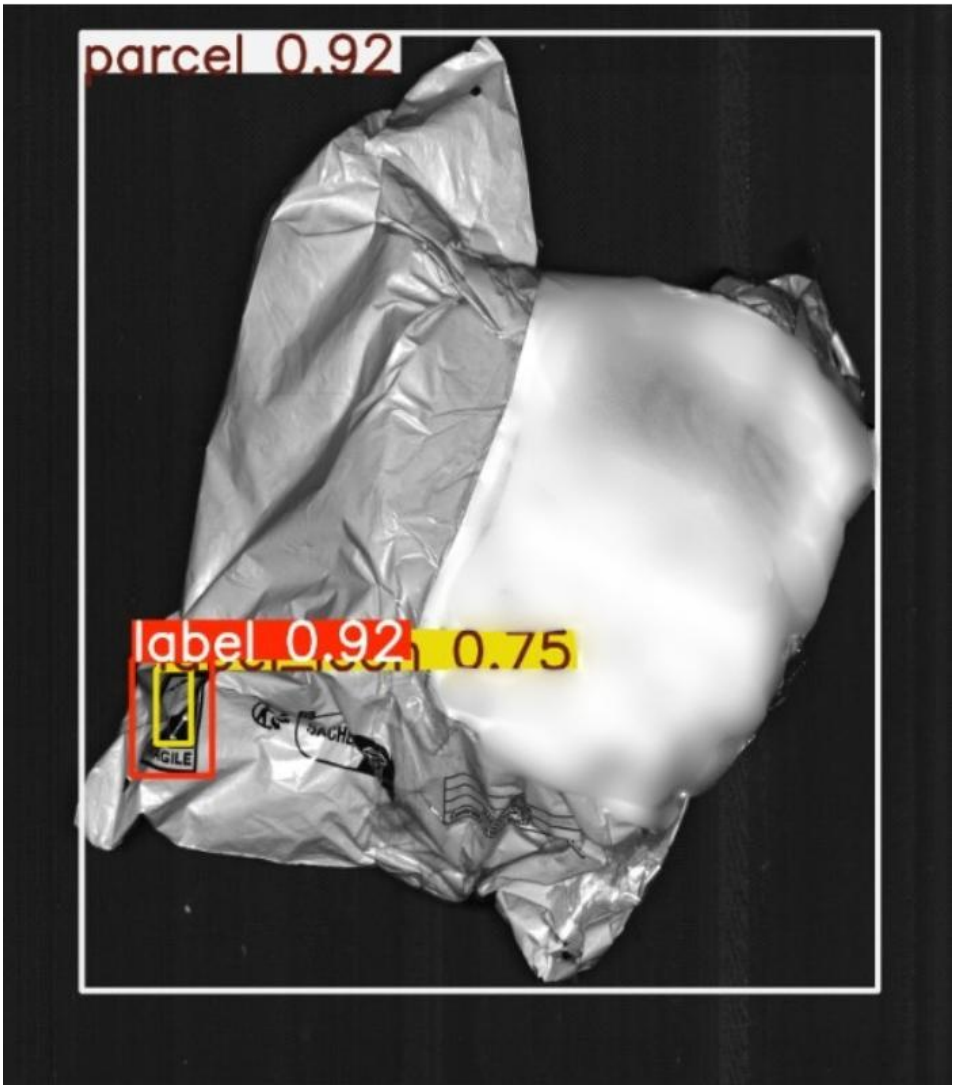
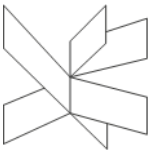


Figure 12 Final model predictions

3.2.2.2 Dataset Expansion and Iterative Retraining

The training dataset was consistently expanded and refined during the project to meet the model's learning requirements and enhance detection performance.



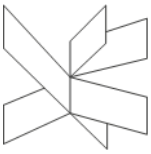
Daily, new images were obtained from a live feed from the logistics warehouse, the real-world samples were examined and labeled with bounding boxes, emphasizing fragile labels under-represented related to the non-fragile parcels.

Dataset upgrades occurred weekly, during which newly acquired image batches were examined for training. Due to non-fragile parcels comprising around 99% of the data, the data collection prioritized images featuring fragile labels. This was predicated on the premise that any fragile label would necessarily be linked to a parcel, where numerous parcels would lack fragile labels. Starting with the third model iteration, an active learning method was implemented to facilitate sample selection (elaborated in Section 3.2.2.4), enabling the automatic identification of potentially relevant training samples.

The model's performance was assessed numerically and via manual inspection. During each evaluation phase, results were visually examined to identify detection failures, particularly overlooked fragile labels. In instances where active learning proposed samples, human validation was constantly employed to guarantee that only correctly labeled and contextually pertinent data were incorporated into the training set. The human verification procedure was crucial for preserving a clean and efficient dataset. While corrupted images from the live feed occasionally appeared, they were identified and not included in training.

The retraining process was developed via successive project iterations. Initially, models were developed from the ground up with less extensive datasets. In the subsequent phases, transfer learning was utilized by training new iterations based on previously optimized models. This enabled the system to preserve fundamental knowledge while integrating ideas from supplementary data.

The ongoing data augmentation, assessment, and retraining cycle significantly enhanced the model's accuracy and generalization, especially in identifying fragile labels across diverse visual contexts.



3.2.2.3 Hyperparameter Tuning

Hyperparameter tuning played a key role in optimizing the detection model's performance, particularly during the early development stages when the dataset was still limited in size. Several parameters were actively modified to maximize detection accuracy and improve the model's capacity to identify small, hard-to-detect fragile labels.

The most critical hyperparameters (Keylabs, 2023) tuned included the learning rate, optimizer type, weight decay, and image input size. Due to hardware constraints, the batch size and number of training epochs were held constant throughout the experiments. Specifically, batch size remained fixed to avoid memory overflows, and the number of epochs was consistently set to 100, which results showed optimal convergence without overfitting. The optimizer was set to *Adam*, and the system was configured with *weight decay*, *patience* of 10 epochs and *max_det* set to 10, limiting the number of objects detected per image. These hyperparameters are explained bellow Figure 13.

Notable enhancements were detected upon modification of the image input dimensions. The original images obtained were frequently of exceptionally high resolution (up to 12,000 × 12,000 pixels). The default image input size of 640 x 640 caused a loss of essential detail, particularly in smaller fragile label areas. Increasing the image input size to 2048 × 2048 significantly enhanced the model's capacity to identify smaller items, such as fragile labels, without overflowing the available GPU resources.

The performance improvements from hyperparameter tuning were assessed by a combination of quantitative metrics, including mean Average Precision *(mAP)*, validation loss, and accuracy, as well as qualitative testing on the images. Challenging test images featuring partially obscured or minuscule fragile labels were used as visual benchmarks to assess practical advancements throughout model iterations.

```
model.train(
    data="dataset/dataset_custom.yaml",  # Path to your data.yaml file (make sure paths in the YAML are correct)
    epochs=100,                 # Increase the number of epochs
    imgsz=2048,                 # Adjust image size for better performance
    batch=3,                    # Use a smaller batch size
    workers=5,                  # Number of data loading workers
    lr0=0.001,                  # Set initial learning rate
    weight_decay=0.0005,        # Set weight decay
    amp=True,                   # Enable AMP for better performance
    project="custom_output6",   # Set the base directory for saving results
    name="experiment_1",        # Set the subdirectory name for this experiment
    optimizer="Adam",           # Set the optimizer
    max_det=20,                 # Set the max detections in one image allowed
    device=0,                   # Set the device to use for training (0 = GPU CUDA)
    patience=10                 # Set the patience
)
```

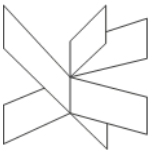
Figure 13 Hyperparameters

data="dataset/dataset_custom.yaml", was used to define the path to the dataset configuration file. This YAML file sets the location of the training and validation images as well as the class labels. Providing a well-structured and correctly referenced dataset file was essential for successful training. Once the hyperparameters were set, this data was the most common change parameter.

epochs=100, was selected to allow the model sufficient time to converge across multiple passes through the training data. This value was determined through experimentation and provided a balance between training duration and performance gains. This parameter defines how many complete iterations the model makes over the entire training dataset, updating its weights with each pass.

imgsz=2048, set the input image size for training. The use of high-resolution images proved beneficial for detecting small objects. The default size was 640, but this proved to be insufficient, removing details from the images that were highly required for identifying fragile labels.

batch =3, defined as the number of images processed simultaneously during training. Due to the increased memory demands introduced by the large input image size, the batch size was kept small to prevent GPU memory overflow, which was a problem in the beginning.



workers=5, specified the number of parallel data-loading threads. This allowed the training process to efficiently preload and preprocess image batches, minimizing idle time between training steps. With high-resolution images, setting this number of workers was required to maintain training throughput and prevent issues caused by I/O latency.

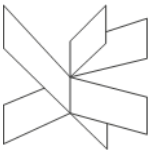
lr0=0.001, sets the initial learning rate for the optimizer. This value is for controlling how quickly the model adjusts its weights in response to the loss function. A learning rate that is too high can lead to unstable training, while one that is too low can slow convergence. The chosen value was found to be idle after some training and provided a stable update rate across iterations and was well-suited to the optimizer selected.

weight_decay=0.0005, was applied as a regularization mechanism to penalize large weight values during training. This technique helps reduce overfitting. It supported generalization by encouraging simpler, more robust representations.

amp=True, enabled Automatic Mixed Precision training, allowing the model to utilize both 16-bit and 32-bit floating-point calculations. This reduced GPU memory usage and training time without compromising model performance. It was especially useful when training, where memory constraints could otherwise limit model capacity or batch size.

project= "custom_output6" and *name="experiment_1"* were used to structure the output directory to save model weights, logs, and evaluation results. This organization facilitated systematic experimentation and easy retrieval of previous model versions for comparison and analysis.

optimizer="Adam" defined the optimization algorithm used to update model parameters during training. The Adam optimizer mixes the advantages of momentum and adaptive learning rates,



making it effective for object detection tasks. Its use contributed to stable and efficient convergence.

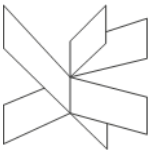
max_det=20, limited the number of detections the model could return per image. This parameter helped control the number of bounding boxes, reducing noise from excessive or low-confidence predictions and ensuring computational efficiency during both training and inference.

device=0, set the GPU device to be used for training. Running the model on a CUDA-enabled device allowed for accelerated computation than would be possible on a CPU.

patience=10, defined the early stopping condition by monitoring validation loss. If no improvement was observed for ten consecutive epochs, training was terminated automatically. This prevented overfitting and lowered the computational cost of running unnecessary epochs.

Tuning was particularly focused on the first phases of the model, where the limited dataset needed optimization to derive dependable learning signals. As the dataset expanded, the influence of additional hyperparameter modifications lessened, leading to more cautious adjustments. Each training session and its associated setup were recorded in a structured spreadsheet, facilitating comparative analysis and informed decision-making for future training cycles.

However, as the project progressed, MLflow was introduced into the workflow. This platform improved how training data and model metrics were tracked and visualized. Unlike static spreadsheets, MLflow provided a dynamic interface with interactive plots, comparison dashboards, and version control for each experiment. These capabilities enhanced interpretability, enabled more efficient model evaluation, and supported faster iteration by offering real-time insight into performance trends across multiple model configurations.



3.2.2.4 YOLO Model

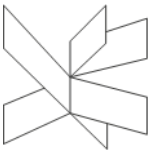
The chosen model (YOLO) is a high performance and well-known model within real-time object detection because of its speed and accuracy. Traditional object detection models perform object localization and classification in separate stages, but YOLO does these at the same stage, making it significantly faster than others.

In YOLO, the image is divided into a grid, and in each cell of the grid the model tries to predict bounding boxes, and the corresponding class based on probability (Figure 8).
YOLOv8 is a version developed by Ultralytics, built on top of earlier versions by improving accuracy, and processing speed and adding features like dynamic input scaling and an updated backbone.

The backbone of a neural network is the part of the model responsible for extracting features from the input data. Therefore, in YOLO, the backbone plays a crucial role, as it enables the model to accurately identify and localize objects in images.

In YOLOv8 weights are very important, because of the anchor-free architecture, and these weights are responsible for detecting objects' bounding boxes and classes (like fragile labels, tires, and parcels). The model's performance relies very much on how well these weights are learned and optimized through training.

Anchor-free refers to a method where the model doesn't rely on predefined anchor boxes, like older object detection models that use anchor-based methods, to predict object bounding boxes. Instead, the model directly predicts the position and size of the objects in an image, by directly predicting the center of the object and its bounding box size. Therefore, the model learns how to detect objects based on the features within the image, making the model more generalized and flexible.



The traditional anchor-based methods used in older versions, like YOLOv3 and YOLOv4, used anchor boxes to define a set of bounding box templates that the model would try to match with objects within the image. These anchor boxes were designed based on the average object sizes and aspect ratios from the training data.

YOLOv8 loss function is a custom combination of different losses designed to balance the tasks of detecting objects and classifying them, consisting of the following losses:

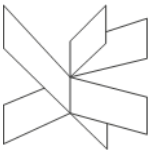
- **Localization Loss:** Measures how accurate the predicted bounding boxes are compared to the ground truth.
- **Confidence Loss:** Measures how confident the model is in the detected object in a predicted bounding box. This evaluates whether the predicted object score aligns with the actual object.
- **Classification Loss:** Measures how well the model classifies the detected object.

Together, these losses are combined into a single loss value, which *YOLOv8* uses during training to improve accuracy. The model goal is to predict precise bounding boxes, high confidence scores, and accurate object classification.

As *YOLOv8* has dynamic input scaling, it allows the model to use different input sizes without sacrificing performance, which is very important in a project that has very high-quality images.

While model pruning, quantization and adding additional layers is possible to the current architecture, there was no need to change it. Pruning could reduce the size of the model, but *YOLOv8* is already very optimized to process objects efficiently and some pruning could compromise detection accuracy, specifically to small objects like the fragile labels.

Quantization could have been implemented to make the model perform better in the hardware available, but quantization could reduce the precision of the weights, potentially impacting performance and accuracy.



The same happens with adding layers, as *YOLOv8* has already an efficient architecture, adding more layers could increase computational complexity, potentially slowing down the inference speed that is crucial to real-time detection in a fast-paced environment like logistics. Therefore, it was chosen not to change or add anything to the default architecture.

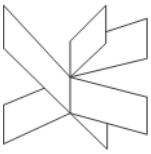
3.2.2.5 Implementation of Active Learning

Active learning was implemented following the third model iteration when manual data selection became a considerable constraint in the development process. The stream of approximately 10,000 images per day, with only around 1% containing fragile labels, made the manual identification process highly time-consuming and unsustainable.

Active learning is a machine learning approach in which the model actively selects the most uncertain samples from an unlabeled dataset for annotation. It is referred to as "active" because the model plays an active role in deciding which data should be labeled next, often based on uncertainty metrics, to improve learning efficiency while reducing annotation costs.

In this context, an active learning strategy was used to prioritize surface images that the model found difficult to classify with high confidence, particularly those likely to contain fragile labels, so that the images could be manually reviewed, re-annotated if necessary, and included in the training dataset.

The active learning process started after the training of each subsequent model iteration. The latest and most efficient model was used to analyze newly obtained image batches. The algorithm was initially set to highlight any discovered fragile label with a confidence score under 50%, indicating instances of model uncertainty. The flagged images underwent manual inspection and annotation, precise bounding boxes were preserved, while wrong detections were rectified and re-labeled to enhance label quality.



This labeling process was conducted using a custom annotation interface, shown in the keybinding configuration image (Figure 14). The interface was developed using OpenCV (cv2), a Python-based computer vision library.

The tool enabled fast, keyboard-based annotation through a set of custom-defined keybindings: "c" to cycle between classes, "g" to manually draw bounding boxes, "z" to undo the last annotation, "d" to delete a selected box, "s" to save labeled images, "n" to skip images without labels, and "q" to mark images as skipped. This solution was designed to handle high image quantities efficiently, minimizing annotation time and maximizing accuracy for the bounding boxes.

The custom implementation significantly reduced manual overhead and played a major role in supporting the scalability of the data labeling process throughout the project.

```
if key == ord("c"):
    current_class = (current_class + 1) % len(cfg["classes"])
    print(f"🔁 Class toggled: {current_class} ({get_class_name(current_class)})")

elif key == ord("g"):
    drawing_manual = True
    print("🖱️ Manual draw mode enabled. Click and drag to create a box.")

elif key == ord("z"):
    if boxes:
        removed = boxes.pop()
        print(f"↩️ Undo: removed last box {removed}")
    else:
        print("No boxes to undo.")

elif key == ord("d"):
    if selected_box_index != -1:
        print(f"🗑️ Deleted box {selected_box_index}")
        boxes.pop(selected_box_index)
        selected_box_index = -1
    else:
        print("⚠️ No box selected.")

elif key == ord("s") and boxes:
    save_label(image_path, boxes, w_orig, h_orig)
    break

elif key == ord("n"):
    save_label(image_path, [], w_orig, h_orig)
    break

elif key == ord("q"):
    print("⏭️ Skipped image.")
    break
```

Figure 14 Active learning key binds code snippet

As model accuracy got better, the focus of active learning transitioned from pinpointing "uncertain" samples to effectively augmenting the dataset with additional fragile label instances. The confidence threshold was reduced to 0%, resulting in the algorithm returning all images containing any object identified as a fragile label. The bounding boxes were manually

checked to rectify their accuracy, hence maintaining a high degree of label precision before their inclusion in the training dataset.

The procedure was used to a cyclical pattern: following inference and evaluation, all validated and re-labeled images were stored and later utilized to retrain the model based on the preceding version. This loop was reiterated for every batch of freshly obtained data. The active learning cycle was performed daily, after working hours, in order to have all the images taken during the working hours.

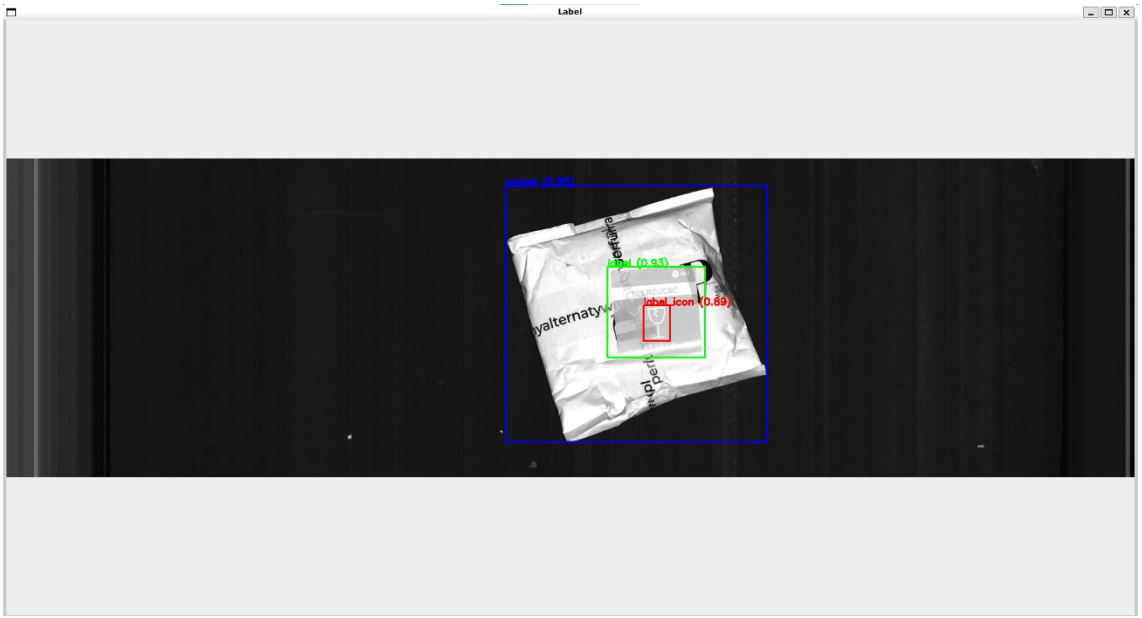


Figure 15 Example of image after active learning

The influence of active learning on model performance was significant. Of the over the 1,200 images utilized in training, more than 1,000 were acquired and enhanced via the active learning process. The enhanced dataset allowed the model to consistently identify fragile labels in progressively harder situations, including instances where labels were partially concealed by tape. This strategy greatly enhanced the system's capacity to generalize and sustain high detection accuracy among real-world scenarios.

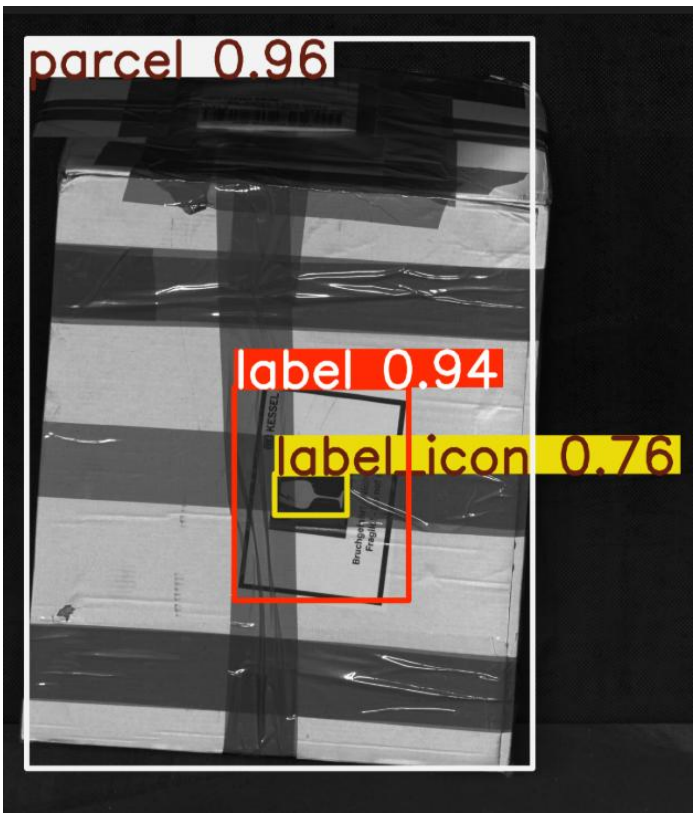


Figure 16 Fragile label detected with tape on top

Another common challenge addressed through active learning was the presence of poorly lit or low-contrast images. In logistics environments, image conditions are not always optimal—some images are captured under inconsistent lighting or from shadowed angles. These darker images often obscured fragile labels or reduced their contrast with surrounding surfaces, making detection difficult. By incorporating such cases into the training dataset and reviewing them manually for bounding box accuracy, the model became more resilient to changes in illumination. As a result, the final model was capable of detecting fragile labels even in suboptimal lighting conditions, enhancing its robustness for deployment in uncontrolled warehouse environments.

Figure 17 Dark image sample

Additionally, the dataset was augmented with images featuring damaged or partially destroyed labels. These cases included labels that were crumpled, scratched, or torn—often from rough handling or being placed on irregular packaging surfaces. Initially, the model struggled with these scenarios due to the high visual variability. However, through iterative labeling and

exposure via the active learning process, the system learned to identify key label features despite imperfections. This increased its ability to detect fragile labels with high confidence, even when visual indicators were partially missing or distorted, demonstrating the value of real-world data diversity in training generalizable detection systems.

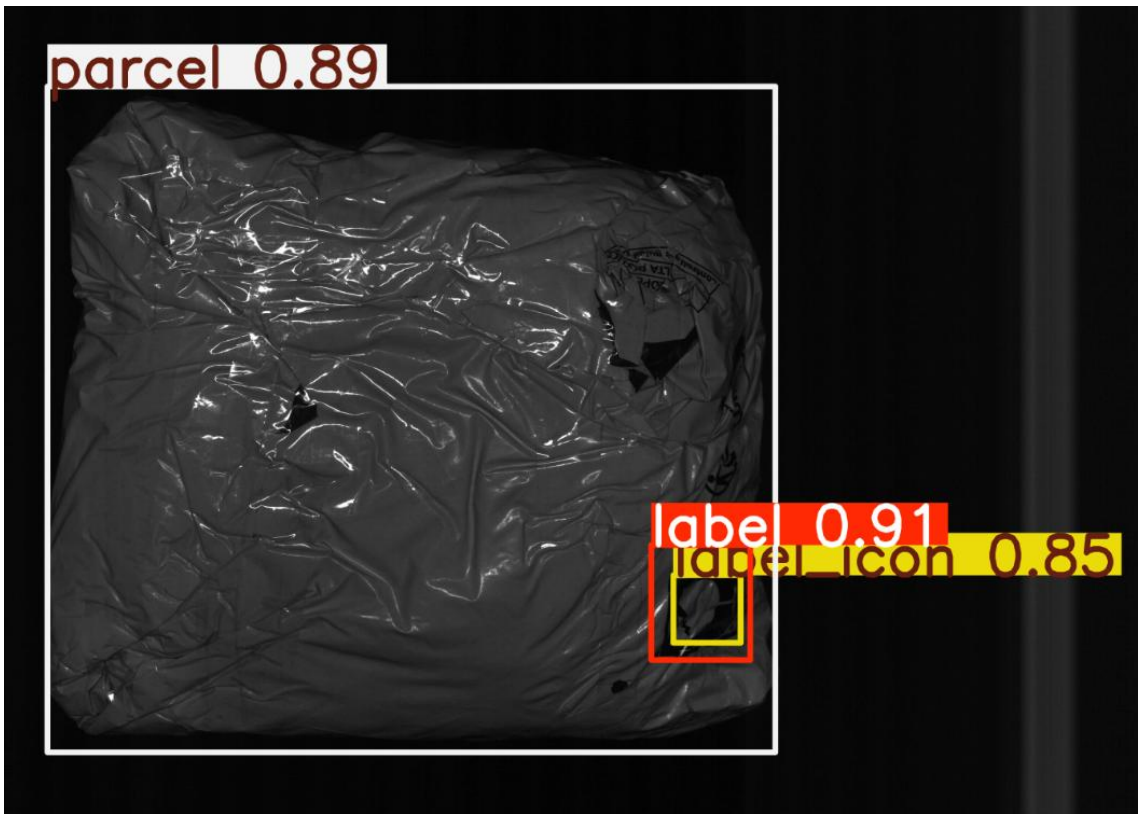
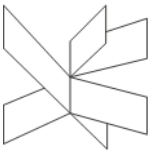


Figure 18 Destroyed fragile label sample

3.2.2.6 Final Model Configuration

The final object detection system was developed using the *YOLOv8n* architecture provided by Ultralytics. This model was selected for its trade-off between inference efficiency and detection accuracy, making it suitable for real-time logistics applications. No custom modifications were applied to the architecture, and the standard implementation was used without model pruning, quantization, or additional layers.



The training dataset consisted of more than 1,200 annotated images sourced from the logistics warehouse environment, consisting of a mix of parcels, fragile labels, and a small subset of tire instances. All images were resized to 2048 × 2048 resolution, which represented the maximum allowed size under the hardware constraints of the CUDA-enabled GPU used for training. The batch size was restricted to three due to memory limitations, and training was conducted over 100 epochs. Each full training cycle required approximately five hours to complete.

The final hyperparameter configuration included a learning rate of 0.001, weight decay set to 0.0005, and the use of the Adam optimizer. Mixed precision training was enabled to optimize performance. The model was also configured with a detection limit of 10 objects per image and an early stopping patience of 10 epochs to prevent overfitting.
Upon completion of training, the final model achieved a precision score of 0.9296, recall of 0.8168, and a mean average precision (mAP50) of 0.84788.

In addition to the quantitative metrics, the model's qualitative performance was evaluated on visually challenging test samples, including images containing fragile labels that were partially obstructed by tape, placed on corners, or distorted in appearance. The model consistently demonstrated an ability to identify these difficult cases, confirming its robustness and generalization to real-world conditions.

3.2.3 Inference Pipeline and System Architecture

The system is designed with a real-time inference pipeline that analyses incoming picture data from a live feed in a logistics warehouse. Images are continuously streamed and stored in a designated directory at Beumer's data centre. A custom file-watcher service is deployed in the Kubernetes environment to recursively monitor the live stream and detect the creation of new .jpg files. The file-watcher disregards folder hierarchy and identifies new files across all subdirectories.

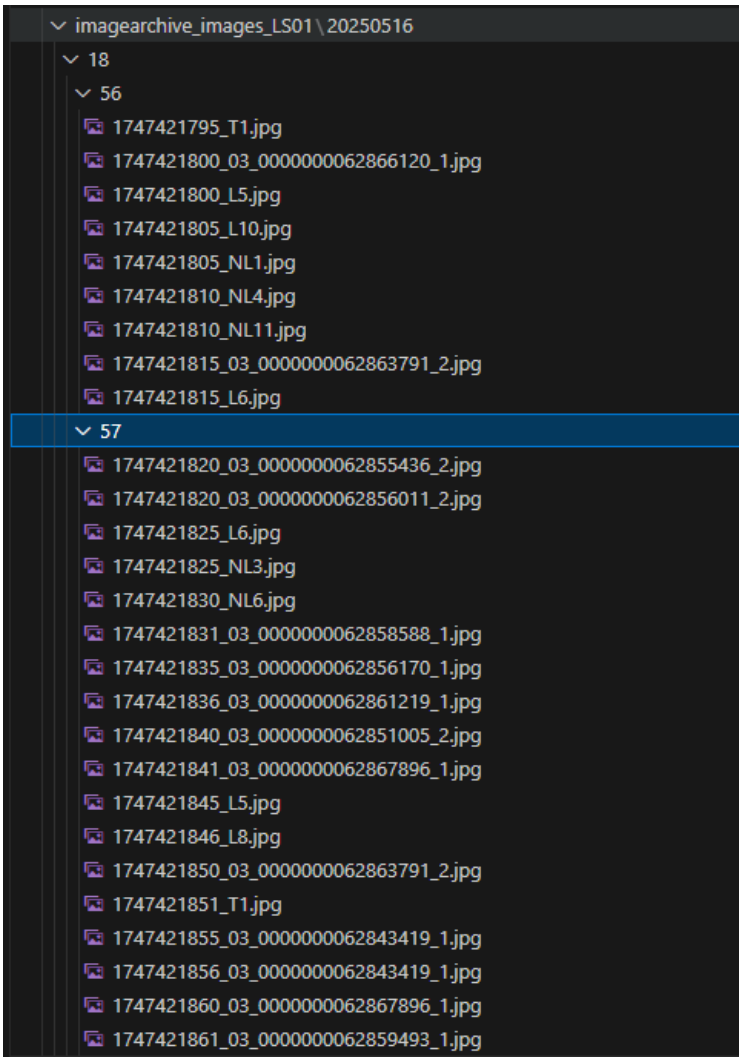


Figure 19 Folder Structure

The images acquired from the live feed camera were stored in a hierarchical directory structure. At the top level, images were stored inside a designated data folder, which was structured by date, with each date represented as a subfolder (e.g., 20250516). Within each date folder, there were 24 subfolders corresponding to the 24 hours of the day (e.g., 18 for 6:00 PM). Each hourly folder was further subdivided into 60 subfolders, each representing a specific minute (e.g., 56 for minute 56 of the hour). Inside each minute-level folder, separate parcel images were stored.

Each parcel was captured using a dual-camera layout, resulting in two images per item, one from the top and one from the bottom.

Upon detecting a new image, the file-watcher adds the file path to a task queue. This queue ensures the system can handle high volumes of incoming images without exceeding memory limitations, which could otherwise lead to system failure under concurrent load. A worker loop within the same service continuously monitors this queue.

```
def on_any_event(self, event):
    if event.is_directory:
        return
    if event.event_type in ("created", "modified") and event.src_path.endswith(".jpg"):
        log.info(f"📥 Queuing image for processing: {event.src_path}")
        task_queue.put(event.src_path)

def worker_loop(self):
    while True:
        path = task_queue.get()
        try:
            self.executor.submit(self.process_image, path)
        except Exception as e:
            log.exception(f"❌ Failed to submit task for {path}: {e}")
        finally:
            task_queue.task_done()  # Ensure the queue doesn't get stuck
```

Figure 20 Queue code snippet

When a task becomes available, it extracts the image path and attempts to process it. If an exception is raised during this stage (e.g., invalid file or read error), the error is logged, and the task is marked as completed to avoid blocking subsequent operations.

When a valid image path is retrieved, the service constructs a payload, in the format that the deployed inference service expects (Figure 21). This payload includes the keys "name", that represents the input that is being sent (in our case “image_path”), "shape" how much information this payload contains (as it only processes one image at the image, the “shape“ is 1 because only one image_path string is being sent), "datatype" and "data", where "data" is the path to the image, not the image file itself.

```
payload = {
    "inputs": [
        {
            "name": "image_path",
            "shape": [1],
            "datatype": "BYTES",
            "data": [path]
        }
    ]
}
try:
    response = requests.post(
        url=MLSERVER_URL,
        headers={"Content-Type": "application/json"},
        json=payload,
        timeout=10
    )
```

Figure 21 Payload format

This lightweight exchange prevents the need to send image files over the network and significantly reduces latency. Because all services, including the inference engine, run within the same Kubernetes cluster, image paths are accessible via shared mounted volumes.

The inference service is deployed using *MLServer*, which was chosen over the default *MLFlow Flask-based* serving engine due to scalability concerns in production settings (MLFlow, n.d.). The model is registered and managed via *MLFlow*, which tracks experiments, stores artifacts, and serves the model metadata. *MLServer* integrates seamlessly with *KServe*, a Kubernetes-native serving framework that provides efficient model hosting, autoscaling, and interface standardization for high-performance environments.

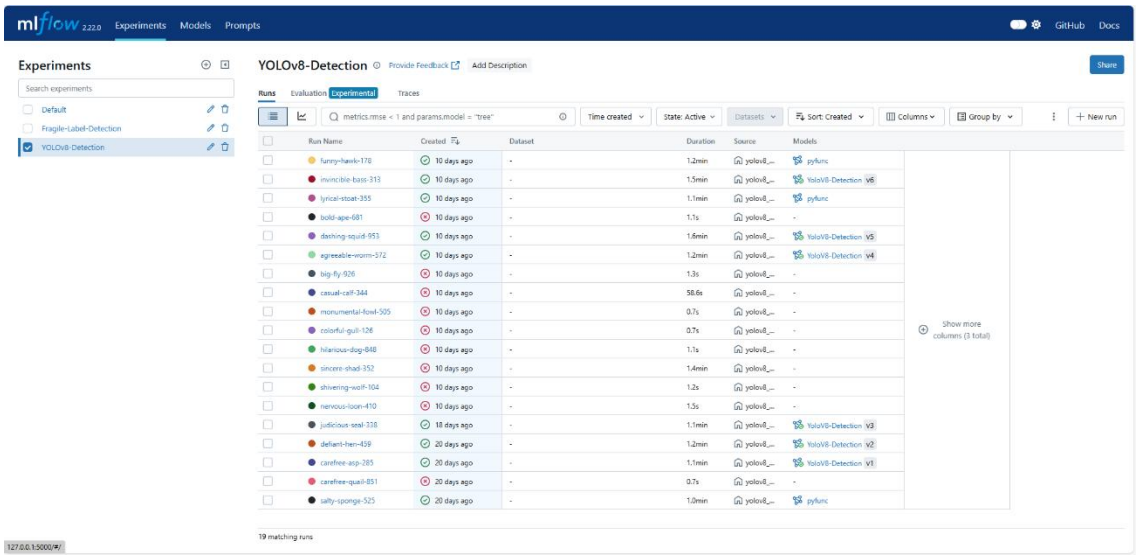


Figure 22 MLFlow UI

The model was logged in *MLFlow* using a Python script that contains an input and output example that the model expects, to do so, creating a real input example was needed, therefore we could infer that input and get an output example after getting the output example generated, a signature was created using an *MLFlow* function with the input and output example, because to log the model, *MLFlow* requires a signature with both examples, therefore the inference service knows what to expect to receive and output, consequently doing data validation whenever a request is made to the inference service. In case there is a bad input to the inference service, the inference service will return a Bad Request code.

```python
# Input example
input_example = pd.DataFrame({"image_path": ["dataset/test/03_0000000063617346_1.jpg"]})

# Get example output from model
yolo = YOLO("custom_output6/experiment_12/weights/best.pt")
result = yolo(input_example["image_path"][0])[0]

example_output = pd.DataFrame([{
    "confidence": float(conf),
    "class_id": int(cls),
    "x": float(box[0]),
    "y": float(box[1]),
    "w": float(box[2]),
    "h": float(box[3]),
} for box, conf, cls in zip(result.boxes.xywh, result.boxes.conf, result.boxes.cls)])

signature = mlflow.models.infer_signature(input_example, example_output)

with mlflow.start_run():
    mlflow.pyfunc.log_model(
        artifact_path="model",
        python_model=YOLOv8Model(),
        artifacts={"model_path": "custom_output6/experiment_12/weights/best.pt"},
        input_example=input_example,
        signature=signature,
        code_path=["yolov8_mlflow_model.py"]
    )
```

Figure 23 Log model to MLFlow

To log the model *MLFlow* requires a custom Python model to be created with at least two functions, *load_context* and *predict, the load_context* basically loads the model file with the model path mentioned in the log model script, and the *predict* function it's the logic the model will use to predict the image given. This function will use *cv2* (OpenCV, n.d.) to read the image and use the model to predict the objects in the image, therefore if anything is detected the function will return an array of detections, otherwise it will return an empty array.

```python
class YOLOv8Model(mlflow.pyfunc.PythonModel):
    def load_context(self, context):
        self.model = YOLO(context.artifacts["model_path"])

    def predict(self, context, model_input: pd.DataFrame) -> pd.DataFrame:
        detections = []

        for image_path in model_input["image_path"]:
            img = cv2.imread(image_path)
            if img is None:
                raise ValueError(f"Could not load image at: {image_path}")

            result = self.model(img)[0]

            for box, conf, cls in zip(result.boxes.xywh, result.boxes.conf, result.boxes.cls):
                detections.append({
                    "confidence": float(conf),
                    "class_id": int(cls),
                    "x": float(box[0]),
                    "y": float(box[1]),
                    "w": float(box[2]),
                    "h": float(box[3]),
                })

        return pd.DataFrame(detections)
```

Figure 24 Custom model

The file-watcher sends inference requests to the *MLServer* with a strict 10-second timeout to prevent resource exhaustion in the event of corrupted or problematic files. If the request fails, an exception is logged, and the task exits cleanly. When successful, the inference output is returned in structured form. These results are serialized into JSON, and a secondary function organizes the data for ElasticSearch ingestion. Each detected object within the image is transformed into a JSON object containing a unique inference ID, the original image path, timestamp, confidence score, class ID (*parcel*, *tire*, *label*, or *label_icon*), bounding box coordinates (x, y, width, height), and *boolean* fields *is_parcel* and *is_fragile*, derived from the object class.

These *JSON* objects are then published to an ElasticSearch index named image-inference-*<currentDay>*, allowing for structured storage and efficient querying. Each detection is sent via a POST request. Kibana, integrated with ElasticSearch, reads from these indices and provides real-time dashboards for monitoring inference activity.

Warehouse staff can interact with these dashboards to analyze the number and type of detected objects, track fragile labels, and evaluate detection accuracy visually over time.

The entire system is containerized and orchestrated within a Kubernetes environment. Shared volumes ensure that image files do not need to be transmitted over the network, significantly improving performance and scalability. The architecture supports near-real-time, large-scale object detection with minimal overhead by decoupling components and leveraging standard ML infrastructure tools such as MLFlow, MLServer, KServe, ElasticSearch, and Kibana.

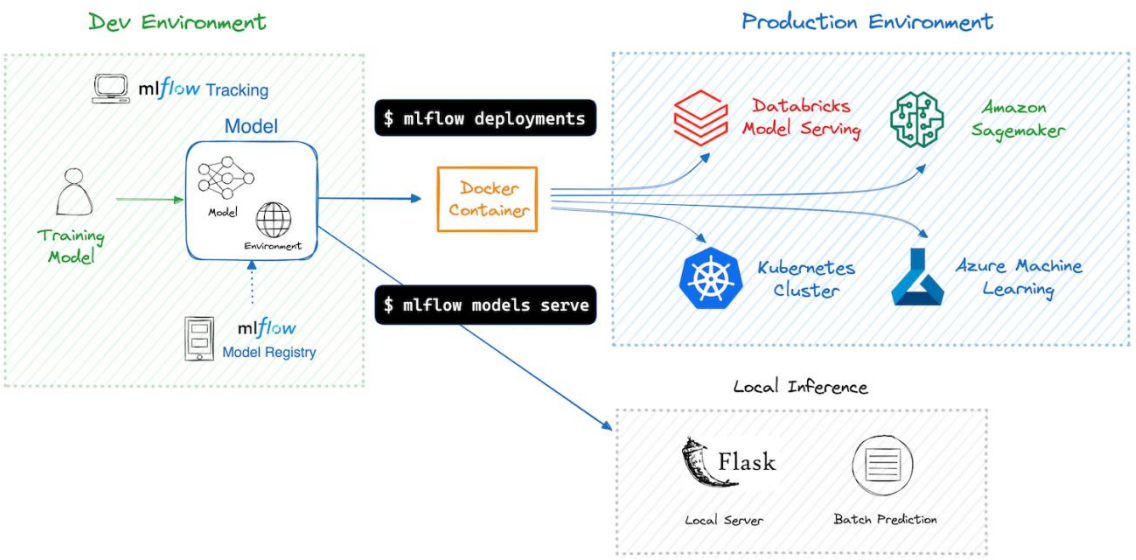
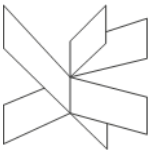


Figure 25 MLFlow Serving

This figure (Figure 25) illustrates the architecture used to manage, track, and deploy the models using MLflow. This project directly adopted MLflow to facilitate scalable and reproducible object detection workflows. The workflow is divided into three functional zones: the Development Environment, the Production Environment, and Local Inference.

Model training and experiment tracking were conducted using MLflow Tracking in the Development Environment. During this stage, each object detection model iteration, including training parameters, evaluation metrics such as mAP and precision, and versioned artifacts, were logged.



After satisfactory performance, the model was registered into the MLflow Model Registry, allowing consistent version control and traceability throughout the project's lifecycle.

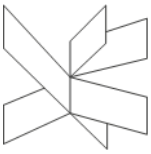
For testing, the trained model was served using "mlflow models serve", enabling local inference through a Flask-based interface. This was essential for rapid testing and validation before transitioning to production.

Although the architecture supports broader cloud-based deployment targets such as Amazon SageMaker and Azure Machine Learning, this project focused on serving within a Kubernetes environment for real-time inference. The trained model was wrapped into a Docker container, allowing it to be deployed via MLServer and integrated into the real-time image processing pipeline.

This architecture provided modularity and reproducibility, enabling integration between model development and deployment phases. It also ensured that the object detection system remained scalable and maintainable, which was particularly important given the continuous data inflow from the live warehouse image feed.

3.2.4 Dashboard Integration

A Kibana dashboard was incorporated into the system as a real-time visualization and analytical layer to provide the inference results' transparency, monitoring, and usability (Figure 27). This dashboard was linked to ElasticSearch indices generated by the inference pipeline, offering a structured and interactive interface for operational insights.
The dashboard presents various aggregated data from the detection results, including total counts for parcels, fragile parcels (determined by the presence of label or label_icon classes), and tires. It additionally provides the percentage of fragile labels relative to the entire parcel count, a metric of significant relevance to warehouse personnel. Furthermore, average



confidence ratings are computed and illustrated for each object class—namely label, label_icon, package, and tire—allowing stakeholders to assess the model's performance over time.
Two principal time-series graphs were employed to facilitate operational monitoring. The initial graph *(Figure 27, bottom left graph)* illustrates the frequency of identified object classes with time, providing insights into class distribution patterns throughout the day. The second graph *(Figure 27, bottom right graph)* illustrates the number of inferences generated over time, indirectly indicating system throughput and availability. This enables users to detect anomalies, such as service unavailability or latency spikes, by monitoring gaps or declines in activity.

The ElasticSearch backend structures inference data into daily indices, each designated with the image-inference-<currentDay>. A data view in Kibana is established using the index pattern image-inference-*, enabling the dashboard to query all pertinent indices. The timestamp feature in each inference JSON acts as the principal temporal axis, facilitating detailed filtering and temporal analysis.

```
for i in range(num_items):
    document = {
        "id": base_id,
        "path": path,
        "timestamp": timestamp,
        "confidence": data_map["confidence"][i],
        "class_id": data_map["class_id"][i],
        "x": data_map["x"][i],
        "y": data_map["y"][i],
        "w": data_map["w"][i],
        "h": data_map["h"][i],
        "is_parcel": data_map["class_id"][i] == 2,
        "is_fragile": data_map["class_id"][i] in (0, 1),
    }
    log.info(f"📦 Sending detection {i+1}/{num_items}")
    self.send_to_elasticsearch(document)
```

Figure 26 Json to elasticsearch code snippet

The dashboard is designed for warehouse operations personnel, offering a detailed overview of the categories and quantities of packages processed, including fragile items. This information can facilitate operational decisions, such as modifying staff allocation for sensitive commodities or assessing the viability of supplementary automation systems based on detection data. The dashboard is a proof-of-value instrument, illustrating how real-time detection models can facilitate logistics optimization and the prospective enhancement of automated workflows.

Despite the absence of automatic alerts or anomaly detection capabilities, the dashboard facilitates dynamic filtering by class type and timestamp. Filters allow users to isolate detection events, whereas time-based controls facilitate the dashboard's adjustment to any specified interval (e.g., last hour, current day, or week). The dashboard is set to refresh automatically every five seconds, ensuring new inference results are displayed in near real-time.

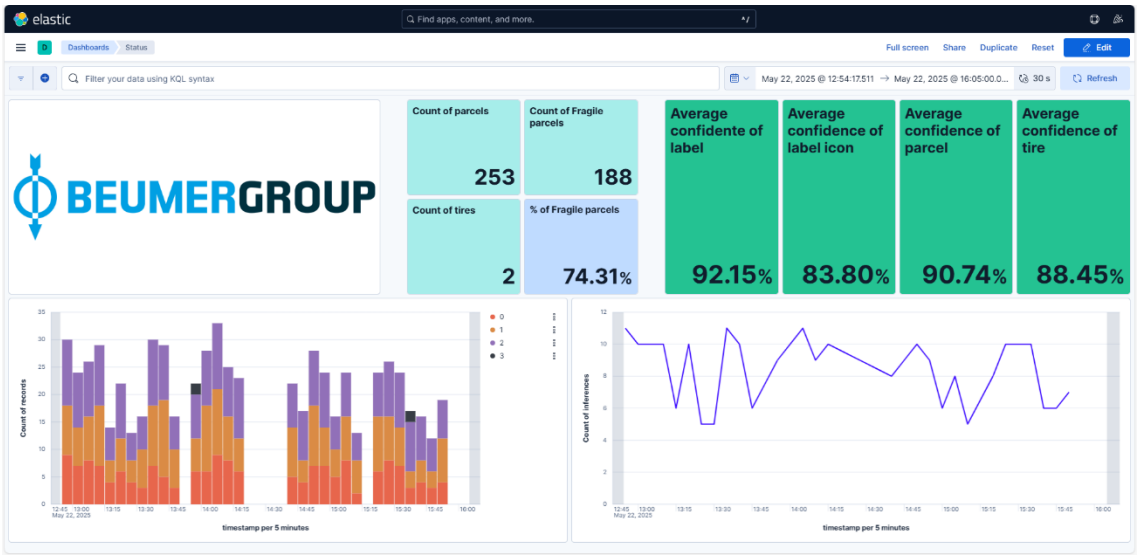
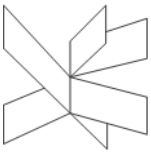


Figure 27 Kibana dashboard



3.3 Results

3.3.1 Model Performance

The superior object detection model, employing the YOLOv8n architecture, demonstrated significant effectiveness in real-time parcel classification and fragile label recognition. The model, trained on a dataset including 1,200 images, achieved a mean average precision (mAP50) of 0.84788, a precision of 0.9296, and a recall of 0.8168. The mean average precision at different IoU thresholds (mAP50–95) was 0.64025. The results were further validated by consistent convergence in validation losses. Qualitative evaluations confirmed the model's capability to accurately detect fragile labels, even in difficult situations, such as when labels were partially obscured or located in non-central areas of high-resolution images. The training process included hyperparameter optimization, active learning integration, and iterative data augmentation, all of which together improved the model's performance.

3.3.2 System Functionality

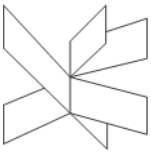
The developed system functioned as a complete, real-time inference pipeline designed for deployment in logistics environments. Live feed images from a warehouse were continuously monitored by a file-watching service, which queued and forwarded image paths to an inference engine. The inference engine, deployed using MLServer and KServe within a Kubernetes cluster, returned detection results in JSON format. These outputs were serialized and sent to ElasticSearch and subsequently visualized through a Kibana dashboard. The system demonstrated the ability to handle incoming images while maintaining stability, responsiveness, and a clean modular architecture. All components—including data ingestion, inference, serialization, and dashboard integration—functioned together reliably under real operating conditions.

3.3.3 System Integration and Validation

The end-to-end pipeline underwent continuous testing with actual data, validating seamless integration among all components. Each image processed by the system triggered a comprehensive inference cycle, producing organised outputs displayed on the dashboard within seconds. This real-time responsiveness was achieved by careful system design, utilising lightweight image management with shared volume mounts and asynchronous job execution via a queue. The output consistency, along with visual input from Kibana, enabled the verification of system integrity and inference accuracy. The manual evaluation of detections on the dashboard confirmed consistency with ground-truth expectations, so further validating the system's operational readiness and reliability.

3.3.4 Ethical Considerations

Ethical and privacy considerations were addressed throughout the procedure. The system neither processed nor stored any personally identifiable information (PII); all data consisted exclusively of images of parcels and tires within the warehouse environment. Image data was not transmitted via external networks but accessed locally using shared volumes within the Kubernetes cluster. Additionally, data annotation and augmentation protocols were implemented to maintain the integrity and representativeness of real-world settings, avoiding any synthetic modifications that could introduce bias into the model. From an ethical AI standpoint, the system creates a foundation for responsible automation in logistics, aiming to enhance operational efficiency while maintaining worker duties, safety, and transparency.



4. Discussion

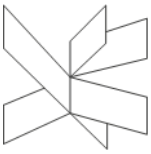
The project's outcomes primarily align with the initial objective: to develop a real-time image recognition system adept at spotting fragile labels and classifying shipments inside logistical environments. Implementing the YOLO-based detection model demonstrated strong performance across many conditions, including partial occlusion, variability in label design, and irregular lighting. The findings confirm the initial hypothesis that modern object recognition systems can be effectively customized to tackle industry-specific challenges.

4.1 Model Benchmarking

Several state-of-the-art models were evaluated on the custom dataset to determine the optimal object detection framework for the task. The evaluation primarily focused on three models: YOLOv8, Detectron2, and YOLOv9.

The initial framework tested was YOLOv8 from Ultralytics, which demonstrated excellent performance in real-time object detection. YOLOv8 offered a strong balance between high detection accuracy and rapid inference speed, making it particularly suitable for time-sensitive tasks. Its advanced architecture, which includes an updated backbone and dynamic input scaling, allowed it to excel in detecting small and irregular patterns, including fragile labels. This model also showed improved generalization and fewer false positives in complex or cluttered visual environments. For these reasons, YOLOv8 emerged as a strong contender for deployment.

In contrast, YOLOv9, an unofficial open-source implementation from MultimediaTechLab, presented challenges during the evaluation. Despite being license-compatible, it suffered from a lack of community support and poor documentation, which created significant usability issues. Moreover, its performance was not consistent with that of YOLOv8, especially regarding accuracy on partially obscured labels. The model showed suboptimal results on complex datasets and integration difficulties with third-party tools like MLFlow. These limitations hindered its practical application for the project.



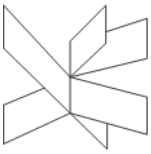
Detectron2, developed by Meta AI Research, was the third framework evaluated. While it is widely recognized in research contexts, its performance in the current application was unfavourable. During testing, Detectron2 demonstrated training rates up to five times slower than *YOLOv8* on the same dataset. Additionally, its prediction accuracy was less reliable, particularly in visual validation and test metrics. Despite its strength in specific research scenarios, *Detectron2* proved unsuitable for real-time detection tasks, where fast training and inference are critical.

	YOLO	**DETECTRON2**
Overview	Object detection	Object detection
Architecture	Single stage	Multi-stage
Speed	Fast	Slower than YOLO in real time
Built on	C and Darknet	Python and Pytorch
Use Cases Example	Real time detection	Medical Diagnosis

Based on this comparative analysis, *YOLOv8* was selected as the primary detection model for deployment. It demonstrated the most favorable trade-off between computational efficiency and detection robustness, making it highly suitable for operational logistics scenarios.

4.2 Interpretation of Results

The model's real-time performance under challenging conditions confirmed the technical feasibility of implementing the system in logistics environments. Notably, it significantly improved detecting fragile labels, even when they were partially occluded or exposed to suboptimal lighting conditions. These enhanced capabilities provide a marked advantage over traditional manual inspection methods, which are labor-intensive and prone to human error.



Furthermore, the amalgamation of ElasticSearch with Kibana created a robust foundation for the storing and visualization of predictive data. This integration facilitated real-time monitoring of critical performance metrics, offering stakeholders substantial insights into system dynamics. It ensured comprehensive traceability of detection events, crucial for continuous performance evaluation and operational transparency.

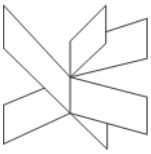
4.3 Limitations and Future Work

The system exhibited robust performance. Nevertheless, numerous limitations were discovered that could be rectified in further generations. Significant illumination fluctuations and extensively deteriorated or damaged labels occasionally affected detection accuracy. To address these problems, incorporating multi-modal sensors, such as infrared or depth sensors, should be considered to enhance the model's resilience under challenging situations.

The system was predominantly assessed in a simulated logistics environment, restricting its engagement with the complete complexity and variety of actual operations. Future endeavors will entail implementing the system in various real-time logistics environments and doing longitudinal assessments to verify its scalability, stability, and dependability across several facilities and operational contexts.

In addition to fragile label detection, the system's functionalities could be enhanced to encompass the identification of additional essential labels, such as hazardous material indicators, priority handling labels, or customs declarations. This would expand the system's applicability and substantially enhance its utility across diverse logistics operations, providing a more thorough automated labelling and package management solution.

From a business standpoint, the data generated by this system could be leveraged in several impactful ways. One application would be direct integration into automated sortation systems, where parcels identified with fragile labels are automatically rerouted to designated fragile-handling conveyors or compartments. Alternatively, the system could function as a metadata



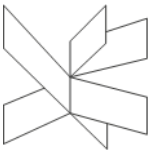
enrichment layer, supplying logistics companies and customers with actionable insights into parcel contents and trends, such as the frequency and distribution of fragile items, thereby informing packaging standards, customer handling protocols, or even predictive damage risk models. The system adds value in either use case by improving decision-making and traceability in high-throughput logistics environments.

4.4 Broader Implications

This project makes significant contributions to both the technical and contextual advancement of intelligent automation in logistics. It offers a repeatable framework for deploying machine learning-based detection systems within operational environments, with a methodology that spans dataset creation, model evaluation, API integration, and continuous monitoring. This thorough methodology can act as a model for analogous automation initiatives across other industrial sectors.

The successful implementation of YOLOv8 reinforces its potential as a versatile object detection model for real-time, resource-limited applications. The study highlights the rising utilisation of lightweight, anchor-free structures, which are gaining prominence for industrial applications, especially in settings where speed and efficiency are paramount.

This project effectively advocates for the shift to automated, intelligent handling technologies in logistics. The system's strong performance, scalability, and adaptability position it as a fundamental component for future smart logistics systems, dependent on further improvement, real-world implementation, and continuous validation.

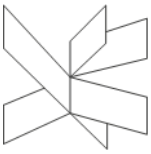


5. Conclusion and Recommendations

This project effectively created and validated a real-time object detection system that identifies fragile labels and classifies shipments in logistics settings. Utilizing the YOLOv8 framework, the system exhibited robust performance under diverse visual circumstances, attaining high precision and recall while maintaining rapid inference speed.

The model benchmarking phase demonstrated that YOLOv8 surpassed other assessed architectures, including YOLOv9 and Detectron2, in detection accuracy and processing efficiency. These findings underscore the advancing sophistication of anchor-free object detection models, establishing YOLOv8 as a leading contender for industrial automation applications, especially those necessitating rapid performance and precision in real-time contexts.

This experiment underscored the vast possibilities for implementing intelligent computer vision systems in high-throughput logistics operations. The effective automation of fragile label identification illustrates that these systems can markedly diminish dependence on manual inspection, decrease human error, and enhance operational efficiency. The modular architecture of the detection pipeline facilitates seamless adaption to other applications, including the identification of hazardous compounds and the management of sensitive shipments.



Recommendations

Extension to Additional Label Types

While the primary focus of this work was on fragile label detection, the system's architecture and capabilities can be expanded to accommodate a wider range of critical logistics labels.

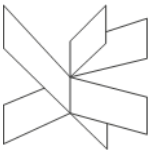
Future work should explore the integration of labels indicating hazard types (flammable, biohazard, etc.), priority handling instructions. This extension would increase the system's value, making it a more comprehensive solution for diverse logistical operations and ensuring safety, compliance, and better handling of sensitive shipments.

Environmental Adaptation

A key limitation encountered during development was the model's performance in extreme environmental conditions, such as poor lighting or damaged labels. To enhance robustness and improve detection accuracy under varying conditions, future iterations of the system should explore the integration of additional sensory inputs. Those additional sensory inputs, could provide a more comprehensive understanding of object characteristics, ensuring higher accuracy across diverse conditions.

Real-World Validation and Deployment

Although the system was rigorously tested in simulated logistics environments, its real-world performance remains untested at scale. A critical next step in the project is to deploy the system to a live logistics site. This would help validate the model's ability to generalize across different types of packages, varying lighting conditions, and diverse warehouse configurations. Furthermore, it would allow for the identification of any operational constraints or system behavior that may not have been captured in the controlled, simulated environment. Real-world deployment will be essential to fine-tune the system for practical, large-scale use.



Continuous Learning Mechanisms

To ensure that the system remains adaptable over time, it is crucial to incorporate continuous learning capabilities. Currently, the model operates with a fixed dataset and does not dynamically adjust to new data post-deployment.

An online learning pipeline could be introduced, where the model continues to learn from new incoming data, such as newly introduced package types, label variations, and unforeseen anomalies. This would allow the model to adapt to evolving logistics environments, improving its performance and accuracy over time. Such a mechanism would be particularly beneficial in handling rare or unique package types that may not be well-represented in the original training set.

Ethical and Regulatory Considerations

As the scope of automation in logistics expands, addressing the ethical and regulatory implications becomes increasingly important. Future work should consider the broader societal impacts of automation, particularly in terms of data privacy**,** surveillance, and labor displacement.

Ethical frameworks should be established to ensure that the deployment of intelligent systems in logistics adheres to privacy laws and standards while minimizing negative effects on the workforce.

Additionally, as logistics systems become more automated, there will be a growing need to align with evolving regulations, such as those related to AI transparency**,** accountability**,** and liability. Future research should explore these dimensions in-depth to ensure that automated systems are developed responsibly and in line with societal expectations.

6. References

Andersen, L. &. Z. M., 2023. Logistics Insight.. *Industry trends in automation for package handling..*

Buslaev, A. a. I. V. I. a. K. E. a. P. A. a. D. M. a. K. A. A., 2020. *Albumentations: Fast and Flexible Image Augmentations.* [Online] Available at: https://www.mdpi.com/2078-2489/11/2/125

DataScientest, n.d. *Yolo - You Only Look Once.* [Online] Available at: https://datascientest.com/en/you-only-look-once-yolo-what-is-it

Elastic, n.d. *ElasticSearch.* [Online] Available at: https://www.elastic.co/elasticsearch

Elastic, n.d. *Kibana.* [Online] Available at: https://www.elastic.co/kibana

GeeksForGeeks, n.d. *ML | Active Learning.* [Online] Available at: https://www.geeksforgeeks.org/ml-active-learning/

Group, W. B., 2023. *Trending Data.* [Online] Available at: https://www.worldbank.org/en/topic/trade-facilitation-and-logistics

Keylabs, 2023. *Maximizing Object Detection: YOLOv8 Performance Tips.* [Online] Available at: https://keylabs.ai/blog/maximizing-object-detection-yolov8-performance-tips/

Kubernetes, n.d. [Online] Available at: https://kubernetes.io/

MLFlow, n.d. *Introduction: Scalable Model Serving with KServe and MLServer.* [Online] Available at: https://mlflow.org/docs/latest/deployment/deploy-model-to-kubernetes/tutorial#introduction-scalable-model-serving-with-kserve-and-mlserver

MultimediaTechLab, n.d. *YoloV9.* [Online] Available at: https://github.com/MultimediaTechLab/YOLO

NISO, 2010. *Scientific and Technical Reports -,* Baltimore: National Information Standards Oganization.

OpenCV, n.d. *OpenCV.* [Online]
Available at: https://pypi.org/project/opencv-python/

Products, I., n.d. *impact-o-graph.* [Online]
Available at: https://impactograph.com/fragile-shipping-labels-do-they-really-work/

Research, F., n.d. *Detectron2.* [Online]
Available at: https://github.com/facebookresearch/detectron2

Smith, J., 2022. A Review. Journal of Logistics Research. *Human Error in Cargo Handling.*

Torres, J., 2024. *YOLOv8 Architecture: A Deep Dive into its Architecture.* [Online]
Available at: https://yolov8.org/yolov8-architecture/#YOLOv8_Architecture_Just_Overview

Ultralytics, 2023. *Ultralytics YOLO format.* [Online]
Available at: https://docs.ultralytics.com/datasets/detect/#ultralytics-yolo-format

Ultralytics, 2023. *YoloV8.* [Online]
Available at: https://docs.ultralytics.com/models/yolov8/

VIA Engineering, in preparation. *Confidential Student Reports,* s.l.: s.n.