

Design Report

Development of an AI-Powered Camera-Based Wagon Identification System

Group 11

Alexandr Gidikov – s3200205

Alexandra Jakovleva – s3199495

Wiktor Brach – s3193691

Tudor Calin – s3185281

Michał Dac – s3210774

Danil Badarev – s3210928

Bachelor Computer Science

University of Twente

Company:	Strukton
Company Supervisor:	Roel Westenberg
UT Supervisor:	Alexia Briasouli
Date:	April 22, 2026

Introduction

The project chosen by Team 11 for this Design Project Module is a real-world engineering challenge provided by Strukton Systems, focusing on intelligent wagon identification in railway marshalling yards. The goal of the assignment is to design and implement a computer vision-based system pipeline that processes side-view video recordings of passing freight wagons and extracts structured information such as wagon identification numbers (UIC), wagon length, axle count, GEVI/UN hazard codes, and container identifiers. The system developed in this project forms the software analysis component of a larger envisioned platform that integrates camera data, IoT sensing, and cloud-based visualization tools.

This report describes the full design and implementation of the proposed solution, including data collection and preprocessing strategies, model selection and training, system architecture, and key engineering decisions. It also discusses challenges encountered during development, evaluates model performance, and reflects on design trade-offs and future improvements.

Contents

Introduction	1
Summary	1
1 Project Description	5
1.1 Assignment Overview	5
1.1.1 Requirements	5
1.1.2 Project Scope and Objectives	5
1.1.3 Data Provided	6
1.2 Timeline	7
1.3 Task Division	7
2 Project Design	8
2.1 Technology Stack and Model Selection	8
2.1.1 Development Tools and Frameworks	8
2.1.2 Machine Learning Model Selection	8
2.1.3 Data Management	10
2.1.4 Coupler Detection Dataset	10
2.1.5 Wheel Detection Dataset	11
2.1.6 Hazard Sign Detection	12
2.2 Pipeline Design	13
2.3 Major Design and Implementation Decisions	15
2.3.1 Using a Frames-per-Wagon Approach	15
2.3.2 Wagon Separation Approach	15
2.3.3 Region of Interest for Couplers and Wheels	16
2.3.4 Sequential vs. Combined Coupler and Wheel Detection	16
2.3.5 Tracking Mechanisms	17
2.3.6 Full-frame OCR versus YOLO-cropped regions	18
2.3.7 Motion Detection	18
3 Project Development	19
3.1 Data Acquisition	19
3.2 Training Execution and Testing	19
3.2.1 Wheel Detection Model	19

3.2.2	Coupler Detection Model	21
3.3	Preprocessing	22
3.3.1	Motion detection	22
3.3.2	Wagon separation	22
3.4	Processing	23
3.4.1	Wheel Detection	23
3.4.2	OCR Recognition	24
3.4.3	Hazard Sign Detection	25
3.4.4	Hazard Sign Classification	26
3.5	Integration	27
3.6	Testing	29
3.7	Try Out Yourself	29
3.7.1	Repository Structure	29
3.7.2	Prerequisites	30
3.7.3	Running the Pipeline	31
3.7.4	Output Format	31
4	Reflection	32
4.1	Project Choice	32
4.2	Technical Challenges	32
4.2.1	Limited and Non-Ideal Training Data	32
4.2.2	Dependency and Environment Conflicts	33
4.2.3	Hardware-Specific Reproducibility Issues	33
4.2.4	Parallelization and Concurrency Limitations	33
4.2.5	Wheel Counting	33
4.3	Future Suggestions	34
4.3.1	coupler detection model	34
4.3.2	Axle Count Improvement	35
4.3.3	OCR improvements	35
4.3.4	DINO + CLIP pipeline Improvement	36
5	Work Process	37
5.1	Scrum and Collaboration	37
5.2	Lessons Learned	37

6	Conclusion	38
A	Appendices	40
A.1	Model Performance Metrics	40
A.2	Pipelines	41
A.3	Data Examples	42

Chapter 1

Project Description

1.1 Assignment Overview

Strukton Systems provided a formal technical requirements specification (Version 2.0) for an intelligent camera system capable of automatically registering passing freight wagons and transmitting structured data to the customer's cloud environment. The full specification covers detection accuracy, message formatting, data transmission protocols, cybersecurity requirements, and physical installation constraints.

The student team was tasked with developing the core video analysis and information extraction component of this system — the software pipeline that processes camera footage and produces structured wagon data.

1.1.1 Requirements

The client specification defines the following key functional requirements relevant to the software pipeline:

ID	Requirement
S1	The system must automatically and accurately register all passing freight wagons at speeds up to 80 km/h, with no false positives or false negatives.
O1	The system must detect passing wagons and recognise the numbers, codes and labels specified in O1.4.
O1.4	Minimum recognised fields: UIC number, GEVI code, UN code, wagon length, number of axles, container number (if applicable).
O2.7	The number of axles must be provided for each wagon or locomotive.
O2.9	All fields in O1.4 must be detected and recognised with an accuracy of at least 95%.
O2.10	The correct number of passing wagons must be detected in at least 99% of cases.
O3.1	Output data must be sent in JSON format via MQTT to the customer cloud environment.

1.1.2 Project Scope and Objectives

The full client specification encompasses hardware installation, wireless communication, cybersecurity compliance, and a cloud integration layer. The scope of this student project

is limited to the software analysis pipeline — the component responsible for processing video footage and extracting wagon information.

The original objective as defined by the client was to process side-view video images of passing wagons to uniquely identify each wagon. Due to limitations in the availability of labelled training data and the complexity of reliable text extraction from real-world footage, it was agreed with that accuracy requirement may go lower than original value. The implemented scope was narrowed to the following objects:

- UIC code
- Length
- GEVI code and UN code (Kemler plate)
- Container code
- Number of axles (wheels)
- GHS Labels

1.1.3 Data Provided

For a computer vision system to perform reliably across a wide range of real-world conditions, a substantial and diverse dataset is typically required. In this project, however, the available training data provided by Strukton was limited, consisting of approximately ten video recordings captured using handheld devices from varying and non-standardised viewpoints.

Given these constraints, and in consultation with the project supervisor, it was agreed that strict adherence to the original accuracy requirements would not be feasible within the scope of the project. This decision reflects the limited availability of representative training data rather than a limitation in the chosen methodologies or implementation approach.

1.2 Timeline

Week	Activity
1–2	Project kick-off, requirements analysis, and research. Reviewed the client specification, discussed scope with the supervisor, and investigated available datasets and relevant prior work.
3	Model selection finalised. First real training data acquired from on-site recordings. Initial dataset preparation and annotation begun.
4–5	First versions of the wheel and coupler detection models trained. Pipeline architecture explored. Additional data collection attempted for OCR targets.
6–7	YOLO models retrained on improved datasets. Wheel counting and coupler separation approaches revised. First OCR experiments conducted. Initial pipeline structure defined.
8–9	Hazard sign detection and classification implemented using GroundingDINO and CLIP. Motion detection integrated. OCR pipeline critically reviewed and redesigned.
10	Full pipeline integration, end-to-end testing, and report writing. Pipeline design changed and documentation completed.

1.3 Task Division

After the research and pipeline design establishment, the tasks were divided the following way:

- Wiktor — majority selection and pipeline design/integration
- Alexandra — model training and wheel counting
- Tudor - hazard sign detection and pipeline design/integration
- Michael — motion detection, wagon separation and model training
- Alexandr — OCR and integration
- Danil — OCR extraction&filtering and pipeline design

Chapter 2

Project Design

This chapter describes the design decisions behind the system. The pipeline processes side-view video recordings of passing freight wagons in two sequential stages: a preprocessing stage that segments the video into per-wagon frame folders, and an analysis stage in which independent modules extract wheel count, OCR fields, and hazard sign classifications from each folder. The sections below cover the technology stack and model selection, dataset construction, pipeline structure, and the key design decisions made during development.

2.1 Technology Stack and Model Selection

This section outlines the core technologies and machine learning models used in the project, along with the rationale behind each choice.

2.1.1 Development Tools and Frameworks

Tool / Framework	Purpose
Python	Primary programming language
OpenCV	Video processing, frame extraction, visualisation
Ultralytics	YOLOv8 training, inference, and export
Roboflow	Dataset management, annotation, augmentation
Google Colab	Cloud GPU training environment (Tesla T4)
DeepSORT	Multi-object tracking for wheel counting
PaddleOCR	Text extraction from wagon surfaces
PyTorch	Deep learning backend for CLIP and GroundingDINO
scikit-learn	SVM training on top of CLIP embeddings

2.1.2 Machine Learning Model Selection

The project employs five machine learning components. Two YOLO-based detectors handle coupler detection for wagon segmentation and wheel localisation respectively. The wheel detector uses YOLOv8 [2] and the coupler detector uses YOLO26 [3]. GroundingDINO [4] localises hazard sign regions from wagon frames using free-form text prompts without requiring labelled training data. Its crops are then classified into GHS hazard categories by a CLIP-based [5] classifier, optionally upgraded with a trained SVM for improved accuracy on visually similar placards. Finally, PaddleOCR [1] extracts text from wagon surfaces including UIC codes, length markings, and Kemler plate codes. Of these five components, only the two YOLO detectors and the optional CLIP SVM required custom training data collection and annotation.

Wheel Detection and Wagon Segmentation: YOLOv8 and YOLO26

Two approaches were considered for wheel and coupler detection during the design phase: semantic segmentation and single-stage object detection. Semantic segmentation was discarded due to its computational cost — per-pixel classification is not well suited for real-time video processing. Single-stage object detection was selected instead, based on its real-time inference capability, suitability for detecting moving objects, and support for transfer learning from pretrained weights, which significantly reduces the amount of task-specific training data required. A potential concern was that bounding box outputs rather than pixel-level masks could cause miscounting when wheels are closely spaced, but in practice the fixed-camera setup and ROI cropping kept wheel spacing consistent enough that this did not occur.

The wheel detection model uses YOLOv8, trained via the Ultralytics framework on Google Colab. The coupler detection model uses YOLO26, which was the default architecture available on Roboflow at the time of training. Both belong to the Ultralytics YOLO family and share the same single-stage detection principles; the version difference reflects the training environment rather than a deliberate design choice

Hazard Sign Localisation: GroundingDINO

For sign localisation, two zero-shot approaches were considered: GroundingDINO and SAM (Segment Anything Model). Both require no task-specific training data, which was the primary constraint given the absence of annotated GHS placard crops from real wagon footage.

SAM was discarded because it is a segmentation model rather than a detector — it produces pixel-level masks but requires a prompt such as a point or bounding box to specify what to segment. Without a prior detection stage to supply those prompts, SAM alone cannot localise signs in a full wagon frame. GroundingDINO was selected instead because it accepts free-form text prompts and directly produces bounding box proposals, making it self-contained as a localisation stage without requiring any additional input.

Additionally, full wagon frames contain significant irrelevant information, making direct classification inefficient. GroundingDINO is therefore used as a filtering stage, leveraging text prompts to localise candidate sign regions which are then passed to the classification step. This enables flexible and efficient detection without requiring any annotated training data.

Hazard Sign Classification: CLIP

Zero-shot classification via CLIP was adopted as the primary mode because no sufficiently large labelled dataset of real-world GHS crop images was available at the start of the project. CLIP maps both images and natural-language descriptions into a shared embedding space, enabling classification by cosine similarity against hand-crafted text prompts without any labelled training data.

An optional SVM mode is also implemented, in which the CLIP image embedding is passed to a Support Vector Machine with an RBF kernel and balanced class weights, trained on a small set of labelled crop images collected during development. This mode

yields higher confidence on well-represented classes such as flammable but is more sensitive to the domain gap between rendered training images and real-world video crops. The architecture was deliberately designed so that the two modes are interchangeable at runtime: the SVM can be introduced or replaced without modifying any other part of the pipeline.

Text Extraction: PaddleOCR

At first, two OCR models were tested: EasyOCR and default PaddleOCR. EasyOCR was showing poor performance on complex scenes (bright sunlight with still visible objects, light motion blur, etc.). Default PaddleOCR model showed a better performance, but it struggled with multi-line objects. It was decided to switch to a specific PaddlePaddle model that combines detection and recognition, which allowed to circumvent the problem of data absence.

2.1.3 Data Management

This section focuses on the design decisions behind training, data selection, and tool choice. Model training is a critical part of the pipeline, requiring careful strategy to ensure generalisation across diverse real-world conditions, especially in scenarios where labelled data is limited, as in this project.

The system combines both custom-trained models and pretrained or zero-shot approaches. YOLO models are trained for coupler detection and wheel localisation, CLIP enhanced with an SVM to enhance hazard sign classification while GroundingDINO and PaddleOCR are used in a pretrained or zero-shot manner to maximise flexibility and reduce data requirements.

2.1.4 Coupler Detection Dataset

Because our system has specific operational needs, the dataset was carefully put together to include 98 high-quality images that were taken by cropping frames from the videos that were taken using mobile phones. Although a data set of this size is considered lean, it provided a high degree of angular consistency. By collecting the data ourselves, we ensured that the model was trained on the exact camera perspective used in the final deployment where the camera is placed at the height of the coupler. This purpose-built approach significantly reduced false positives, as the model did not have to filter out irrelevant perspectives such as top-down or front-facing views.

Even with only 98 base images, the dataset was designed to cover a range of lighting conditions. As shown in Figures 2.1 and 2.2, this included sun-hit frames where metal surfaces produce strong reflections, and dark shadowed frames where coupler details are hard to distinguish.



Figure 2.1: Bright image with visible reflections



Figure 2.2: Dark, shadowed image

2.1.5 Wheel Detection Dataset

The wheel detection model was trained on a dataset compiled from 21 video recordings of trains passing through a fixed camera inspection zone, each approximately 30 seconds in duration. Frames were extracted at 2 frames per second to avoid near-duplicate images from consecutive frames, yielding approximately 1,260 images in total. The examples of them can be found in Appendix A.3 in Figure A.3.

The dataset was split at the **video level** rather than the frame level to prevent data leakage. Splitting randomly at the frame level would place visually near-identical frames from the same video into both the training and validation sets, artificially inflating validation metrics. Videos were assigned exclusively to one partition:

Partition	Videos	Images	Proportion
Training	15	≈ 900	71%
Validation	3	≈ 180	14%
Test	3	≈ 180	14%

A region of interest (ROI) crop was applied to all images prior to annotation. Since wheels occupy a predictable vertical band in the lower portion of the frame, cropping removes irrelevant background and ensures the training distribution matches the inference distribution exactly. Approximately 10–15% of the dataset consists of negative samples — frames containing no wheels — to prevent false positive detections on track infrastructure and train chassis components.

Annotation was performed in Roboflow using foundation model-assisted auto-labelling, where initial bounding box proposals were reviewed and corrected manually to capture exactly the one wheel per box without other objects included. After labeling the following augmentations were applied to the training partition:

- Brightness variation ($\pm 25\%$) — simulates tunnel, night, and overcast conditions
- Exposure variation ($\pm 15\%$) — accounts for camera auto-exposure differences

- Gaussian blur (up to 2px) — approximates motion blur at train speed
- Image noise — improves robustness across camera hardware quality
- Horizontal flip — handles trains travelling in either direction

Heavy rotation augmentations were excluded as wheels on track are always horizontally oriented, and such transformations would introduce physically implausible training examples.

Figure 2.3 shows an early dataset example before ROI cropping and augmentation were applied. Figure 2.4 shows a sample from the final training set after both were introduced.

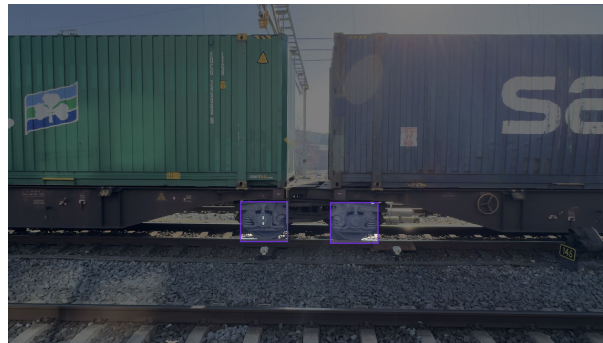


Figure 2.3: Data set during first attempts (no ROI, no augmentations)

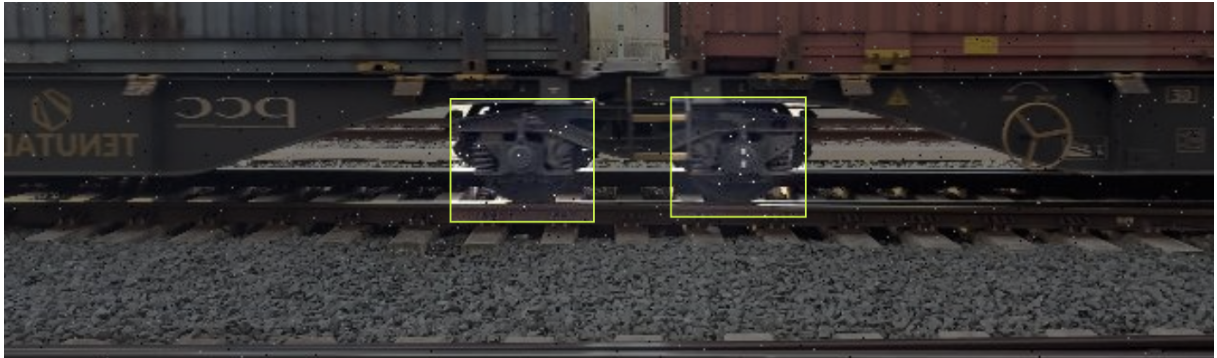


Figure 2.4: Example of data used in the final training set using ROI and image noise

2.1.6 Hazard Sign Detection

The choice of framework for hazard sign detection and classification was driven by the complete absence of GHS data on wagon videos. Training a custom detector or classifier from scratch was not viable without annotated examples of ADR placards captured from freight wagons under real operating conditions.

GroundingDINO was selected for localisation because it requires no task-specific training data. Given a text description of the target object, it produces bounding box proposals directly from the image, allowing the system to search for hazard diamonds using

natural-language prompts such as “*red diamond-shaped sign with flame*”. This approach generalises across placard types without any annotation effort.

CLIP was selected for classification of the localised crops for the same reason. Its pre-training on 400 million image-text pairs gives it strong representations of coloured geometric shapes and pictograms, making zero-shot classification both feasible and accurate without labelled training images.

2.2 Pipeline Design

The pipeline is structured in two stages. A preprocessing stage converts the raw video stream into a set of per-wagon frame folders, each representing a single physical wagon. Then an analysis stage processes these folders independently, with every module producing a structured record for its wagon. This shared intermediate representation, the wagon folder, is the central design choice that shapes the rest of the pipeline; the rationale behind it is discussed in Section 3.4.

A diagram of the full flow is given in Figure 2.5. Earlier iteration of the pipeline design which were rejected during integration phase are preserved in Appendix A.2 in Figure A.2.

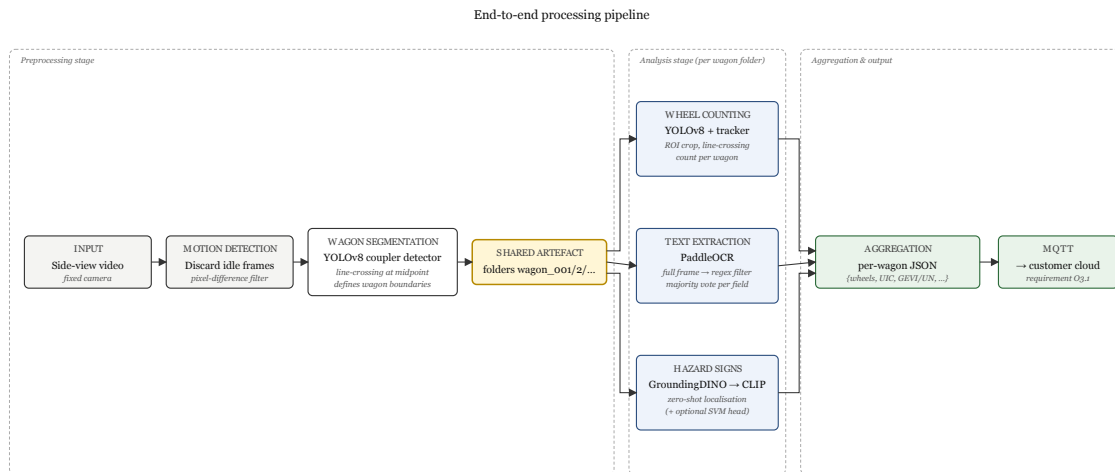


Figure 2.5: End-to-end pipeline, showing the preprocessing stage (motion detection, coupler-based segmentation) producing per-wagon frame folders that are consumed independently by the wheel, OCR, and hazard-sign modules.

The end-to-end flow comprises the following stages:

Input acquisition. The pipeline consumes a single side-view video recorded by a fixed camera at the inspection zone. No assumption is made about train direction, and the footage is processed offline rather than streamed in the current scope.

Motion detection. Before any model is invoked, a lightweight motion filter discards sequences in which the frame content is static. This removes idle footage at the start

and end of each recording and reduces downstream work, since wheel tracking, OCR, and hazard sign detection are only meaningful while a train is physically passing the camera.

Coupler detection and wagon segmentation. The first YOLOv8 detector locates couplers on the moving train. A coupler is, by construction, the mechanical boundary between two wagons, and the pipeline uses its track to define where one wagon ends and the next begins. A new wagon folder is created each time the tracked coupler centre crosses the horizontal midpoint of the frame (the same line-crossing rule used later for wheel counting), and subsequent frames are written into that folder until the next crossing event. The output of this stage is a deterministic sequence of folders `wagon_001/`, `wagon_002/`, ..., each containing the frames belonging to exactly one wagon.

Wheel counting. For each wagon folder, a second YOLOv8 detector localises wheels within a fixed region of interest in the lower portion of the frame, and a DeepSORT tracker maintains a consistent identity for every wheel as it moves through the detection zone. Counting is performed by line crossing: a wheel is counted exactly once when the centre of its track traverses a virtual vertical line at the horizontal midpoint of the ROI. A fresh tracker instance is created for each wagon folder so that tracking state cannot leak between wagons.

Text extraction. PaddleOCR is applied to full wagon frames rather than to cropped regions. The recognised strings are then filtered in postprocessing against regular expressions and spatial-layout rules to isolate UIC codes, length markings, Kemler plate fields (GEVI and UN), and container codes.

Hazard sign localisation and classification. Hazard sign handling is split into two sub-stages. GroundingDINO is queried in a zero-shot fashion with text prompts describing GHS placards; its bounding-box outputs are cropped and passed to a CLIP-based classifier, optionally augmented with a Support Vector Machine trained on top of CLIP image embeddings, which assigns each crop to a GHS pictogram class. A confidence threshold is applied to reject ambiguous crops.

Aggregation and output. Per-wagon results from the analysis modules are aggregated into a single JSON record per wagon and emitted in the format required by the client specification (O3.1), ready for transmission over MQTT to the customer cloud environment.

The defining property of this design is the contract between stages: every analysis module consumes a wagon folder and emits one structured record, with no direct dependency on any other analysis module. This makes each module testable in isolation and allows new modules — for example, wagon-length estimation — to be added without modifying the existing code, which directly supports the project supervisor’s extensibility requirement.

2.3 Major Design and Implementation Decisions

These design decisions were established in the first three weeks of the project, alongside initial data collection, and before implementation began in earnest. Several of them changed as the team encountered practical limitations during testing, and the decisions described here reflect the final rationale after those iterations.

2.3.1 Using a Frames-per-Wagon Approach

An early design had each module — wheel counting, sign detection, OCR — operating directly on the video stream, with wagon boundaries determined on the fly. This proved difficult to coordinate: each module operated at a different frame rate, needed different crops, and had different tolerance for noisy or transitional frames. The pipeline was restructured around a shared preprocessing step that segments the video into per-wagon frame folders before any analysis begins. Each downstream module then reads from its assigned folder independently, and can sample every n -th frame if it does not require full temporal resolution — sign detection and OCR, for example, do not benefit from processing every frame and run on a sparse sample.

An earlier version of this segmentation triggered a new wagon folder whenever a coupler was detected anywhere in the frame. This was inconsistent: detection could fire mid-frame before the wagon boundary had fully passed the camera, causing the first and last few frames of a wagon to be assigned to the wrong folder and introducing noise into downstream modules. The trigger was moved to the moment the coupler track crosses the horizontal midpoint of the frame, the same line-crossing logic used for wheel counting, which produces a clean and consistent split point regardless of train speed.

2.3.2 Wagon Separation Approach

The initial idea for separating wagons was a pixel-difference sliding window: a significant change in frame content would indicate that the gap between two wagons had passed the camera. This was tested and rejected for two reasons. First, many wagons carry no container and present a plain side surface with little texture, meaning the contrast between the wagon body and the inter-wagon gap was insufficient to trigger a reliable change signal. Second, when multiple containers are loaded on a single wagon they are visually separated but belong to the same unit, so the sliding window would split a single wagon into multiple segments.

Coupler detection was adopted instead. Couplers are mechanically present between every pair of wagons without exception, are geometrically distinctive, and occupy a consistent position in the frame. The approach did not seem promising initially given the limited training data available, but after iterative dataset collection and model refinement it produces acceptable separation results for the videos tested.

2.3.3 Region of Interest for Couplers and Wheels

Because the available training data was limited and the detection models are not highly generalised, a fixed region of interest crop is applied before passing frames to the coupler and wheel detectors. Wheels always occupy the lower portion of the frame and couplers appear at a predictable horizontal and vertical position, so restricting detection to those zones significantly reduces false positives from background clutter, track infrastructure, and irrelevant wagon surfaces. The tradeoff is that this approach assumes a stable, fixed camera position — the ROI coordinates would need to be recalibrated if the camera is repositioned or the installation geometry changes.

This strategy does not generalise to all pipeline components. Hazard signs can appear at varying heights and horizontal positions across different wagon types, so a fixed crop would miss many of them — GroundingDINO is used instead to localise sign regions dynamically. UIC codes present a similar problem as they can appear almost anywhere on the wagon surface and do not have a consistent location, which is why OCR is applied to the full wagon frame rather than a fixed crop.

2.3.4 Sequential vs. Combined Coupler and Wheel Detection

Since both wagon separation and wheel counting are YOLO-based and operate on the same video frames, two unification approaches were considered. The first was training a single two-class YOLO model to detect both couplers and wheels simultaneously, eliminating the need to run two separate models. This was rejected because the two detection tasks have conflicting requirements: the wheel detector needs a tightly cropped lower zone to avoid false positives on track infrastructure, while the coupler detector operates on a different region of the frame at a different vertical range. Training a single model on both classes would force a compromise on the input crop, likely degrading performance on both tasks. Each model also has a different confidence profile and failure mode — merging them would make it harder to retrain or replace one without affecting the other.

The second approach kept two separate models but ran them together on each frame. Because running models sequentially would cause a significant slowdown we tried to run it in parallel on each frame. using a `ThreadPoolExecutor`:

```
1 fut_w = executor.submit(detect_wheels, wheel_model, wheel_crop
2 )
3 fut_c = executor.submit(detect_coupler, coupler_model,
4     coupler_crop, cx1, cy1)
5 wheel_dets = fut_w.result()
6 coupler_dets = fut_c.result()
```

Listing 2.1: Parallel inference attempt

This did not yield the expected speedup. Python’s Global Interpreter Lock prevents true thread-level parallelism for CPU-bound inference, and on GPU both models competed for the same CUDA cores and VRAM. The combined loop also became difficult to debug — two trackers, two detection zones, frame saving, and wagon state resets all shared the same execution context, making it hard to isolate the source of a counting error. Threading introduced additional complexity, with race conditions appearing when a new

wagon’s tracker was initialised while the previous wagon’s frames were still being written to disk.

The pipeline was therefore restructured into two sequential stages: the coupler detector first segments the video into per-wagon frame folders, then the wheel counter processes each folder independently. This makes each stage testable in isolation and directly meets the project supervisor’s requirement for a maintainable, extensible system. Any future module — OCR, hazard sign classification, or length estimation — can be added as a new stage reading from the same folder structure without touching existing code.

2.3.5 Tracking Mechanisms

A core requirement of the wheel counter (see Appendix A.2 Figure A.1) is that each physical wheel is counted exactly once. This requires a tracking system that maintains consistent identities for detected objects across frames, so that the same wheel is not counted multiple times as it moves through the detection zone.

The first approach used a narrow region of interest window, sized so that only one wheel could be inside it at a time. Each wheel entering the window would receive a unique track ID, and counting IDs would directly give the wheel total. This worked for slow-moving trains but failed at higher speeds: if the window was narrow enough to isolate individual wheels, fast-moving trains would pass through it entirely between sampled frames and go undetected. Widening the window to capture every wheel meant multiple bogies were visible simultaneously, causing the matcher to assign IDs inconsistently and requiring parameter tuning for each video.

A custom Hungarian algorithm matcher was then implemented over a wider detection zone, matching detections to tracks by horizontal centroid distance. This was more stable but still caused ID swaps when two bogies were simultaneously close in horizontal position, and the `max_dist` threshold that worked for one video would fail on another due to differences in train speed or camera distance.

DeepSORT was adopted to address these limitations. It combines a Kalman filter, which predicts each track’s next position from its motion history, with a Hungarian assignment that matches new detections to existing tracks using both predicted position and appearance. Tracks that go unmatched for more than `max_age` frames are deleted; new tracks are held as tentative until confirmed by several consecutive detections, filtering out single-frame false positives. Since all wheels are visually identical the appearance component provides little benefit here, but the Kalman filter’s motion prediction maintains track continuity across frames where a detection is missed due to motion blur or a brief confidence drop.

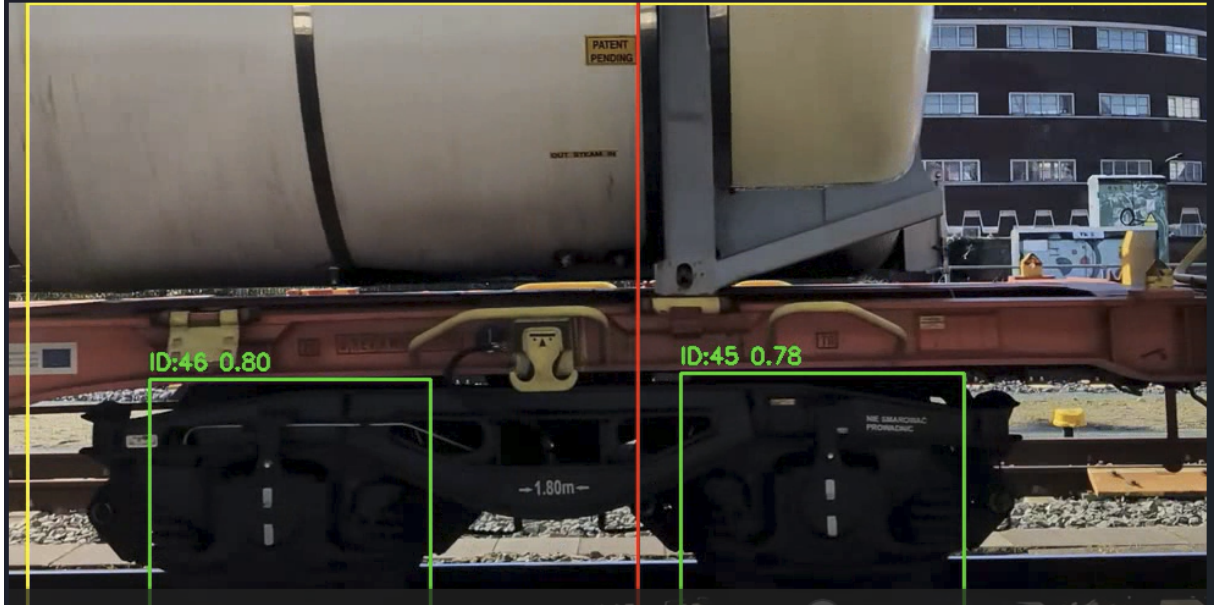


Figure 2.6: Yellow line represents the ROI border, the line in the middle is used for counting, and detected wheels are in green boxes being tracked by deepSort.

2.3.6 Full-frame OCR versus YOLO-cropped regions

Rather than a two-stage pipeline (detection crop to OCR), the system performs OCR on the full frame and applies structured filtering in post-processing. This choice was primarily driven by data availability. Training a sign detector requires a large number of annotated examples covering diverse real-world conditions - different lighting, camera angles, and partial obstructions. The collected dataset was not large enough to train such a detector reliably. A detector trained on too little data would miss objects too often, which would make the downstream recognition irrelevant.

The filtering-based approach avoids this problem entirely. Instead of relying on a learned detector, it uses the known structure of the target codes to decide which detected text is relevant. Text that does not match the expected format or spatial layout is discarded.

This makes the system more predictable: its behavior is governed by explicit rules rather than a model that could fail on unseen data. The trade-off is that more text is processed than strictly necessary, but given a relatively small number of text elements visible in a typical frame, this has no substantial impact on performance.

2.3.7 Motion Detection

To save resources such as energy and computing power we thought of implementing a motion detection. To implement this we found motion detection that works by comparing each frame to a learned background model (MOG2). For every frame, it looks at how many pixels inside the region of interest have changed. If that ratio is above 5%, the train is considered to be moving.

Chapter 3

Project Development

3.1 Data Acquisition

At the beginning of the project, we attempted to collect data from videos available online, as well as from labeled datasets provided by Roboflow. However, we soon realized that this data was too diverse and insufficient for our purposes. As a result, we made the decision to narrow the scope to specific train types and conditions. Although this reduced generality, it allowed us to focus on scenarios closer to real-world use and more aligned with the trains used by Structon.

Starting from the third week, several team members regularly visited Hengelo train station to record videos using smartphones. This process was repeated weekly, although we did not have access to precise train schedules until the final weeks. With multiple attempts and support from our supervisor, we collected approximately 25 high-quality videos for further development and testing.

One of the main challenges was maintaining consistency in recording conditions, such as camera angle, distance, and the visibility of train codes, which were not always clearly observable. The system is intended to operate with static cameras, whereas the collected footage varied in angle, clarity, and object positioning. This variability made it more difficult to design and adjust the processing pipeline.

Despite these challenges, and given the limited alternatives, the self-collected data proved valuable and enabled us to build the system in its current form.

3.2 Training Execution and Testing

The detailed overview of all the steps taken to train Machine Learning YOLO models used for coupler and wheel detection and the statistics obtained.

3.2.1 Wheel Detection Model

The YOLOv8m model was trained on Google Colaboratory using a Tesla T4 GPU. Training was configured with a maximum of 100 epochs and an early stopping patience of 20 epochs, meaning training terminates automatically if validation mAP50 does not improve for 20 consecutive epochs. This prevents overtraining without requiring manual intervention.

Parameter	Value	Justification
epochs	100 (max)	Early stopping governs actual duration
imgsz	640	YOLOv8 default; matches preprocessing
batch	16	Optimal for T4 VRAM
patience	20	Prevents overtraining
optimizer	AdamW	Auto-selected; lr=0.002, momentum=0.9

Training converged at **45 epochs** in 1 hour 3 minutes. The training curves (Figure 4.1) show all three loss components decreasing consistently across epochs, with validation precision and recall both stabilising above 0.95 from epoch 7 onwards. The characteristic spike in validation losses at epoch 2 is consistent with early-stage weight reorganisation during transfer learning fine-tuning and does not indicate instability.

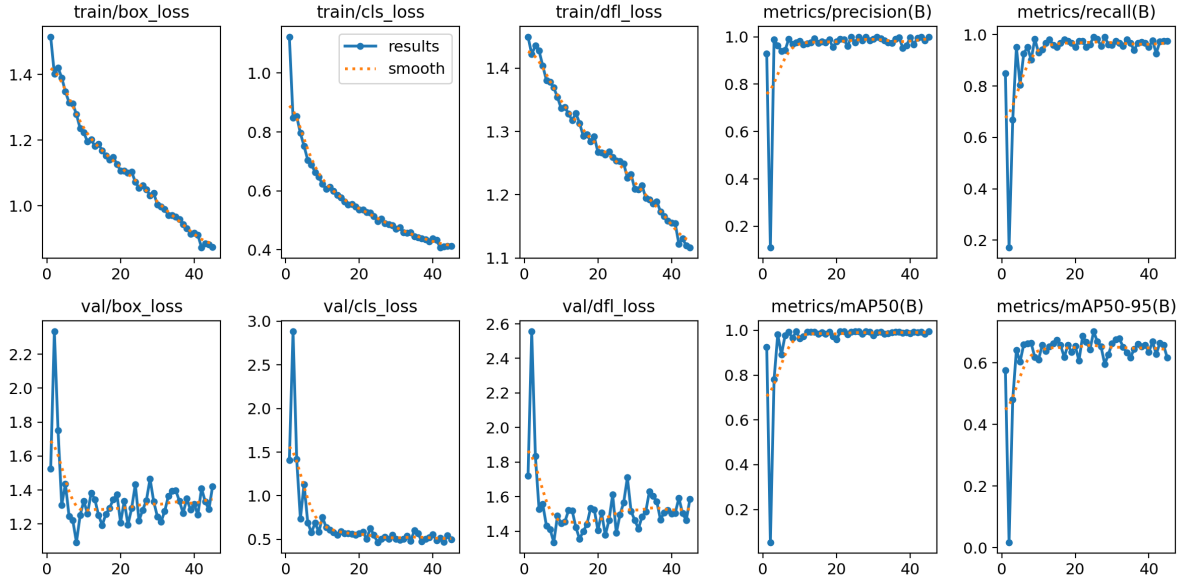


Figure 3.1: Training and validation loss curves, precision, recall, mAP50, and mAP50-95 across 45 training epochs.

Final performance on the validation set:

Metric	Value	Interpretation
mAP50	0.995	Near-perfect detection at 50% IoU threshold
mAP50-95	0.706	Strong box precision across stricter thresholds
Precision	0.989	98.9% of detections are genuine wheels
Recall	0.992	99.2% of all wheels in frame are detected

Evaluation on the held-out test set — three complete videos never seen during training or validation:

Metric	Validation	Test	Difference
mAP50	0.995	0.953	−0.042
mAP50-95	0.706	0.561	−0.145
Precision	0.989	0.974	−0.015
Recall	0.992	0.873	−0.119

The modest gap between validation and test performance confirms the model generalises to unseen footage rather than memorising the training distribution. The primary driver of the lower test recall is the higher proportion of background frames in the test set (64% vs 32% in validation), which places greater demand on the model’s specificity. Inference speed was approximately 14.1ms per frame end-to-end, corresponding to a theoretical throughput of 71 frames per second — well above the 25–30 fps of standard inspection cameras. Full validation and test set metrics for the YOLOv8m wheel detection model can be found in Appendix A.1.

3.2.2 Coupler Detection Model

The model was trained on Roboflow using the YOLO26 (Fast) architecture. Training was set for 200 epochs. It was trained on the dataset described in section 2.1.4 with such Training/validation/testing ratios: 0.8/0.1/0.1

Final performance on the validation set:

Metric	Value	Interpretation
mAP50	0.995	Near-perfect detection at 50% IoU threshold
Precision	1.0	100% of detections are genuine couplers
Recall	0.989	98.9% of all couplers in frame are detected

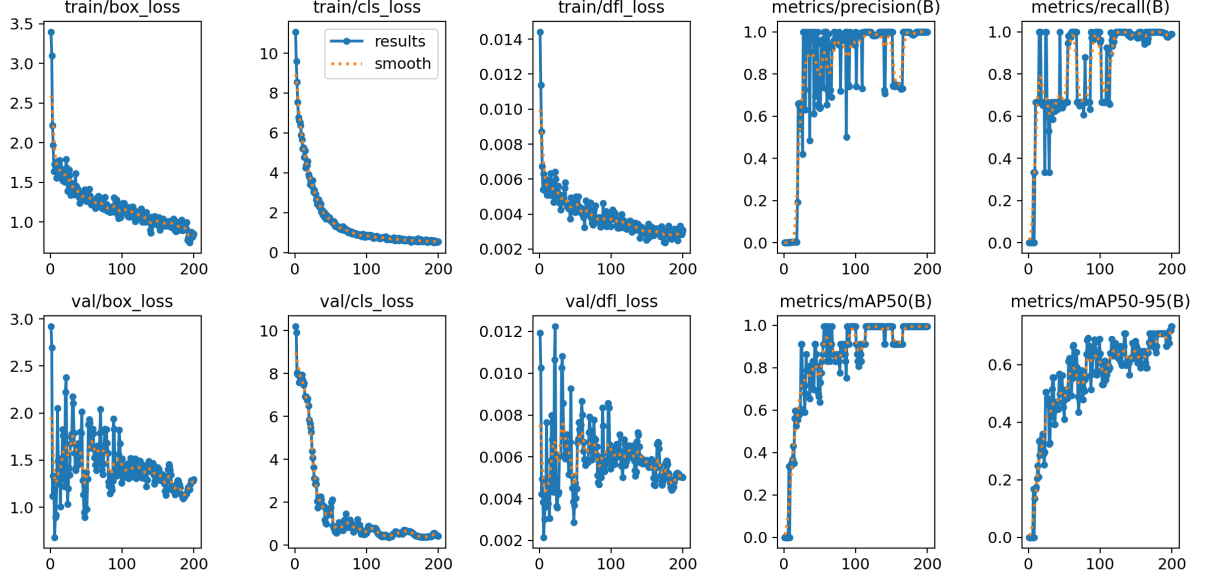


Figure 3.2: Training and validation loss curves, precision, recall, mAP50, and mAP50-95 across 200 training epochs.

3.3 Preprocessing

This section describes how the input video is processed and the preparatory steps taken before data acquisition begins.

3.3.1 Motion detection

Before our detection pipeline begins to run, we are continuously monitoring the live video footage for an incoming train. So, before any frame is ever buffered, the system checks whether there is a moving train. A background subtractor (MOG2) computes the ratio of changed pixels in the region of interest. Only when the motion ratio spikes above the motion threshold (default 5%), the system starts other processes.

3.3.2 Wagon separation

Separation of wagons is the key challenge for the system. The system must be able to correctly identify the end of one wagon and the start of the next using the video feed.

A train is not made up of a sequence of completely separated objects. The various wagons are mechanically coupled, meaning that when viewed from a camera, they are almost one continuous surface. The only reliable visual boundary between wagons is the coupler assembly a relatively small mechanical joint that's often partially obscured, in shadow, or moving fast enough to blur. The system has to detect this small, inconsistent feature and use it as the sole segmentation signal. Wagon separation works by using two things together. The first is a YOLO model that spots the physical coupler between wagons

in the video. While a coupler is visible, those frames are thrown out since they show the junction rather than clean wagon body. The second is a centre zone check when a detected coupler travels through the middle third of the frame, that is the moment the system decides that one wagon has ended and the next has begun, and the current batch of frames is saved as a wagon. Even with this, the frames right around a coupler are still a problem shadows, end-caps, and coupling hardware can linger in the frame even after the coupler itself is no longer detected. To deal with this, the system keeps a short cooldown after every coupler disappears, during which frames are still thrown out. How long this cooldown lasts depends on how fast the train is moving - slower trains get a longer cooldown, faster trains get a shorter one. This makes the system cautious by design: it would rather discard a few good frames than include a single bad one that could mess up the number reading or wheel counting later.

3.4 Processing

This section will describe in detail how system extracts needed data (wheels, codes, numbers, labels) after preprocessing wagon by wagon.

3.4.1 Wheel Detection

The wheel detector uses a YOLOv8m model and applies two post-processing filters to raw detections before passing them to the tracker:

1. **Confidence filter** — detections below 0.65 confidence are discarded.
2. **Size filter** — bounding boxes smaller than 50×50 pixels are removed as noise.

The NMS IoU threshold is set to 0.4, which suppresses redundant overlapping detections at the model level before tracking.

Wheel counting is performed per wagon using a DeepSORT tracker instantiated fresh for each wagon folder. Processing each wagon independently ensures that tracking state does not carry over between wagons, eliminating cross-wagon double-counting errors in case of frames per wagon are separated correctly.

For each frame sequence, detections from the wheel detector are passed to DeepSORT in the required format: a list of `([x1, y1, x2, y2], confidence, class)` tuples. The tracker returns confirmed tracks which are then checked for line crossing.

Counting is performed via a virtual line at the horizontal midpoint of the detection zone. A wheel is counted when its track centre crosses this line between frames:

```

1 if prev_cx < zone_mid_x <= cx or prev_cx > zone_mid_x >= cx:
2     wheel_count += 1
3     crossed_ids.add(tid)

```

Listing 3.1: Line crossing detection

Once a track ID is added to `crossed_ids` it is not counted again, even if the track remains active until `max_age` expires. The bidirectional condition handles trains travelling in either direction.

The final output is a JSON file of the form:

```

1 {
2   "wagon_001": 8,
3   "wagon_002": 4,
4   "wagon_003": 8
5 }
```

Listing 3.2: Wheel count output format

Key configurable parameters:

Parameter	Value	Effect
<code>conf</code>	0.65	Minimum detection confidence
<code>iou</code>	0.4	NMS threshold inside YOLO
<code>zone_width</code>	1000px	Horizontal width of detection zone
<code>max_age</code>	5	Frames a track survives without detection

The whole pipeline diagram can be found in Appendix A.2 in Figure A.1. The implementation performed stably on all footage collected by the team. Further testing on a wider range of train speeds and wheel types would be required before generalising to other operating conditions. Parameters should be adjusted for a future use.

3.4.2 OCR Recognition

To implement OCR detection, a dedicated `OCR` class was developed, built around two complementary models from the PaddleOCR library: the detection model *"PP-OCRv5_mobile_det"* and the recognition model *"PP-OCRv5_server_rec"*. The mobile variant was chosen for detection due to its lower computational overhead during the bounding box localisation stage, where speed is prioritised over precision. Conversely, the heavier server variant was selected for recognition, as accurately reading individual characters is the most quality-critical step in the pipeline.

In the main pipeline, an `OCR` instance is created per frame and exposes a dedicated extraction method for each target element. Each method returns a list of potential candidates (e.g. potential UIC numbers, potential container identifiers), which are subsequently passed to filtering and validation functions to determine the most likely correct value. All extraction methods share a common base extraction step, which runs the full detection-recognition pipeline on the frame and produces three outputs: a list of recognised text strings, their associated confidence scores, and the bounding polygon for each detected text region. These bounding polygons are four-point quads, providing the spatial coordinates necessary for proximity-based search. Based on this foundation, four extraction algorithms were implemented:

1. **UIC extraction** - The algorithm first locates the most structurally stable part of the UIC number: the last seven digits followed by the check digit, matched using a

regular expression. Given the bounding box of this anchor token, a spatial search is performed to find the nearest text boxes to its left and above. If at least one four-digit token or two two-digit tokens are found in this region, they are used to reconstruct all possible 12-digit UIC candidates, which are then validated using the Modulo 10 check digit algorithm.

2. **Kemler plate extraction** - The algorithm begins by identifying a four-digit UN code, then scans the region directly above that box for a two- or three-digit GEVI code. Candidate pairs are accepted only when the two boxes share sufficient horizontal overlap, reflecting the physical layout of a Kemler plate where the GEVI code is printed directly above the UN code.
3. **Length extraction** - This algorithm exploits the consistent formatting of wagon length markings: a floating-point number between 8 and 30, always suffixed with the unit **m** (e.g. **24.0m**). A regular expression is applied directly to the recognised text list without any spatial reasoning. The limitations of this approach due to small font size are discussed further in the Reflection section.
4. **Container ID extraction** - Container identifiers follow the ISO 6346 standard: three owner letters, one category letter, a six-digit serial number, and a check digit. Because the check digit is visually close to the serial number, OCR occasionally merges or omits it, reducing validation reliability. Candidates where the check digit could not be confidently extracted are therefore retained with a placeholder **?** in place of the final digit, and flagged as unvalidated in the output.

Overall, each extraction method is designed around the specific format and physical layout of its target element on the wagon and provides the most optimal potential candidates to the filtering in postprocessing.

3.4.3 Hazard Sign Detection

The first stage of hazard sign detection involves localising regions of interest using GroundingDINO. The model is queried using structured text prompts that describe the visual characteristics of the target hazard symbols. For GHS hazard diamonds, the prompts explicitly reference common pictogram variants, such as *“red diamond-shaped sign with flame”* or *“white diamond-shaped sign with skull and crossbones”*, allowing the model to perform zero-shot localisation without requiring task-specific training data.

After processing the frames of each wagon, the raw detections are filtered using a geometric and rule-based post-processing stage prior to classification:

- **Size filter** — bounding boxes smaller than 3% or larger than 15% of the frame dimensions are discarded.
- **Aspect ratio filter** — detections must have a width-to-height ratio between 0.55 and 1.4, corresponding to the expected geometry of hazard diamonds.
- **Label validation** — the free-form label returned by GroundingDINO is checked against expected keywords. A detection is accepted only if the label contains *“diamond”* and a recognised colour descriptor.

These filtering steps significantly reduce the number of false positives typically produced by zero-shot detection models. The remaining detections are cropped and stored per wagon, and subsequently passed to the CLIP-based classification stage.



Figure 3.3: Example of a sign crop sent from DINO for classification.

3.4.4 Hazard Sign Classification

Each cropped detection produced by the DINO stage is classified using CLIP (Contrastive Language–Image Pre-training, ViT-B/32), a vision-language model that maps images and natural-language descriptions into a shared embedding space and enables classification by cosine similarity without any labelled training data.

For each GHS label, a set of descriptive text prompts is defined that characterizes the visual appearance of the class. For example, the flammable class is described by prompts such as *“red diamond-shaped hazard placard with a bold black flame pictogram in the center”*, while the toxic class uses prompts referencing *“a black skull and two crossed bones”*. Multiple prompts per class are used to capture variation in viewing angle, lighting, and partial occlusion.

A confidence threshold of 0.85 is applied: crops whose top score falls below this value are rejected and do not contribute to the wagon result. This reduces false positives at the cost of leaving some ambiguous crops unclassified.

Additionally, for classes with a strongly distinctive color-flammable (red) and oxidizer (yellow)-a filtering step is applied after classification. The crop is converted to the HSV color space, and the fraction of pixels matching the expected hue range is computed. If fewer than 10% of pixels match, the classification is overridden, and the crop is rejected. This eliminates a class of false positives caused by dark metallic wagon components being misclassified as colored placards.

Duplicate detections within a single frame are suppressed by a centre-distance deduplication check. The final output per wagon is a deduplicated list of human-readable dangerous goods labels, for example ["Flammable Liquid", "Toxic"].



Figure 3.4: Example of a GHS sign classification.

3.5 Integration

Before the system was integrated together into a final product, the components of the pipeline had been developed in isolation. Each subteam worked on its own module, wrote its own small driver script, and tested on a handful of sample videos. Bringing them together into a single system turned out to be the most time-consuming part of the project.

The first problem was conceptual. So far the pipeline had been treated as a one-shot batch

process over a finished video file: read the file from disk, iterate over frames, produce an output at the end. The client specification, however, requires the system to run against a live camera feed, which is continuous and has no natural end point. The batch model could not be stretched to fit this requirement without a redesign of the control flow.

The solution was to split the runtime into two loosely coupled segments. A `TrainMonitor` continuously observes the camera feed and decides when a train is in view, and a `ProcessingWorker` is handed a bounded sequence of frames every time the monitor signals a passing train. This split turned the live-feed requirement into a local decision (“is a train present right now?”) and kept the rest of the pipeline working on bounded inputs, which is what it had originally been designed for.

Before any code was merged, both segments were specified as black boxes with fixed input and output contracts — the monitor consumes camera frames and emits a frame range; the worker consumes a frame range and emits the per-wagon JSON records described in Section 3.2. Once those contracts were written down, the remaining work could be parallelised: one subteam took the train detector, the other the processing pipeline.

The train detector was finished quickly. Most of the logic had already been validated as part of the standalone motion-detection experiments, so the subteam mainly had to wrap the existing behaviour in the monitor loop and add the signalling interface to the worker.

The processing pipeline was the harder of the two, and the difficulty was almost entirely in the dependency stack rather than in the pipeline logic itself. Combining PyTorch with GPU support and PaddleOCR with GPU support on NVIDIA hardware proved unexpectedly fragile: the two frameworks pull in different CUDA runtime expectations, and each has its own preferred combinations of Python, CUDA, and cuDNN versions. Several days were spent cycling between package versions, reinstalling environments, and reading compatibility matrices before a configuration was found in which both frameworks could be imported in the same process without one silently disabling GPU acceleration for the other.

The issue was also hardware-specific. The working configuration depended on the CUDA compute capability of the developer’s GPU, so the combination that resolved cleanly on one machine failed on another even with identical `requirements.txt`. On those machines the environment had to be patched manually — swapping individual wheels inside the virtual environment and pinning transitive dependencies — before the pipeline would run end to end.

Once the environment was stable, the integration code itself was relatively mechanical. The pipeline diagram in Section 3.2 was followed stage by stage: the worker invokes coupler-based segmentation, then iterates over the resulting wagon folders and runs the analysis modules in turn, and finally assembles the JSON records for the monitor to forward. No algorithmic changes were needed at this step; the effort was in making the separately developed modules agree on file layout, confidence thresholds, and error handling.

3.6 Testing

Testing was conducted modularly, with each component developed and validated in isolation before being integrated into the full pipeline. This approach made it significantly easier to identify the source of failures, as each module could be run independently on a single wagon folder or video clip without invoking the rest of the system.

Visual inspection was the primary testing method throughout. Rather than relying on automated test suites, each module was equipped with debug output that made its internal behaviour directly observable. For coupler and wheel modules was used the output video which depicted log of the system (coupler/wheels count), ROI and detected objects.

For hazard sign detection, intermediate crops were saved to disk after processing each wagon folder. Reviewing these crops made it possible to confirm whether GroundingDINO was localising the correct regions and whether the geometric filters were accepting or rejecting the expected detections.

For OCR, recognised strings and their bounding boxes were overlaid on the frame and saved as annotated images, allowing manual verification of which text regions were detected and whether the post-processing filters were selecting the correct candidates.

Each module was tested on majority of available video recordings before integration. Once combined into the full pipeline, end-to-end runs were performed on complete videos and the final JSON output was reviewed against ground truth counts and codes verified manually from the footage.

3.7 Try Out Yourself

This section describes the repository layout and explains how to run the system on a test video.

3.7.1 Repository Structure

The project root is organised as follows:

```
design-project/
src/
  main.py           # entry point (TrainMonitor + ProcessingWorker)
  pipeline.py       # TrainPipeline - orchestrates all stages
  debug.py          # logging and visual debug helpers
detection/
  wagon_segmenter.py # coupler-based wagon segmentation
  wheel_counter.py   # YOLOv8 wheel detection + DeepSORT tracking
  wheel_detection.py # low-level wheel detector wrapper
  frame_sampler.py   # evenly samples N frames from a wagon buffer
  model_roboflow.py  # Roboflow coupler model integration
  best.pt            # trained coupler-detector weights (YOLO26)
  wheel_detection_best.pt # trained wheel-detector weights (YOLOv8)
```

```

    prop/                                # standalone test scripts (not part of pipeline)
postprocessing/
    ocr_extractor.py                      # PaddleOCR wrapper, per-wagon field extraction
    ocr_filtering.py                     # regex and spatial-layout filters
    uic_voting.py                        # cross-frame majority vote for UIC fields
    json_forming.py                      # UIC / container-code validation helpers
    ocr.py                               # legacy standalone OCR script
preprocessing/
    image_stiching.py                    # frame stitching utility
ghs/
    ghs_detector.py                      # GroundingDINO + CLIP hazard sign detector
models/
    GroundingDINO_SwinT_OGC.py           # GroundingDINO model config
    groundingdino_swint_ogc.pth          # GroundingDINO pretrained weights
GroundingDINO/                          # cloned GroundingDINO library
data/
    testing_data/                       # sample videos used during development
outputs/                                 # pipeline output (wagon frames, JSON results)
requirements.txt                         # pinned Python dependencies
sample_wagon.json                       # example per-wagon output for reference

```

The two core files to understand are `src/main.py` and `src/pipeline.py`. The entry point in `main.py` launches two threads: a `TrainMonitor` that watches the video feed for motion and records each train pass as a segment file, and a `ProcessingWorker` that picks up each segment and runs the full `TrainPipeline` defined in `pipeline.py`. The pipeline itself calls the modules in order: wagon segmentation, wheel counting, hazard-sign detection, and OCR extraction, then assembles the results into a single JSON document per train.

3.7.2 Prerequisites

The system requires Python 3.12, an NVIDIA GPU with CUDA 12.x drivers, and the dependencies listed in `requirements.txt`. Note that both PyTorch and PaddlePaddle are pulled from their respective GPU indices; the exact index URLs are specified in `requirements.txt`. Due to the sensitivity of the CUDA dependency stack described in Section 4.5, it is recommended to create a fresh virtual environment:

```

python -m venv .venv
.venv\Scripts\activate          # Windows
pip install -r requirements.txt

```

If PaddleOCR or PyTorch fail to detect the GPU after installation, consult the troubleshooting notes in Section 4.5 — the fix typically involves matching the `nvidia-cuda-runtime-cu12` version to the host driver.

3.7.3 Running the Pipeline

The pipeline accepts a video file as input and treats it as a simulated live feed. The basic invocation is:

```
python -m src.main data/testing_data/cut_10s_good.mp4
```

This will write segment files and per-train JSON results into the `outputs/` directory. Two optional flags are available:

```
python -m src.main <video> --output-dir results/ --debug
```

The `--output-dir` flag changes where output is written. The `--debug` flag enables verbose logging and opens visual debug windows that overlay detections on each frame in real time, which is useful for verifying that wagon boundaries, wheel tracks, and OCR regions are behaving as expected.

Alternatively, the processing pipeline can be run in isolation (without the monitor thread) for quicker testing on a single known video:

```
python -m src.pipeline <video> --ocr-frames 20 --debug
```

3.7.4 Output Format

Each processed train produces a JSON file in the output directory. The structure of a single wagon entry is shown below (see also `sample_wagon.json` in the repository root):

```
{
  "wagon_id": 1,
  "wheels": 4,
  "dangerous_goods": ["Flammable Liquid"],
  "wagon_approximate_length": {"value": "14.20m", "confidence": 0.88},
  "uic": {
    "detected_text": "21 81 247 1217-3",
    "confidence": 0.95,
    "is_valid": true
  },
  "kemler_plate": {
    "gevi_code": "33", "un_code": "1203",
    "confidence": 0.91, "is_valid": true
  },
  "container_id": {
    "detected_text": "ABCU1234567",
    "confidence": 0.78, "is_valid": true
  }
}
```

Fields for which no confident detection was found are set to `null`. The `confidence` values reflect the majority vote agreement for OCR fields and the model score for hazard signs.

Chapter 4

Reflection

4.1 Project Choice

The project was primarily chosen due to a strong interest in the field of artificial intelligence and the opportunity to collaborate with an industrial partner on a real-world engineering problem. From the beginning, the project description and supporting documentation provided a clear goal and well-defined problem statement.

Communication with the company supervisor, Roel, was established quickly and remained smooth throughout the project. He was responsive to questions and provided valuable initial insights into the problem domain and expected outcomes. However, his limited experience with similar computer vision and machine learning projects meant that the team received relatively little technical guidance from the company side.

Another important limitation was the lack of provided training data, which significantly influenced both the project scope and planning. As a result, several design decisions had to be adapted to accommodate data scarcity and to rely more heavily on pretrained and zero-shot methods.

Despite these challenges, the overall experience was very positive. The team was invited to visit the company office, where we were introduced to ongoing projects and gained insight into how the organisation operates in practice. This visit provided valuable context and made the collaboration more concrete and insightful.

4.2 Technical Challenges

During development, the team encountered several limitations and challenges that influenced both the design and implementation of the system. Some of them are described in this section.

4.2.1 Limited and Non-Ideal Training Data

A major constraint throughout the project was the limited availability of high-quality, annotated training data. The dataset provided by the client was very small (approximately ten videos) and captured under non-standardised conditions, including varying camera angles, handheld recording, and inconsistent lighting.

This had several consequences for the system design:

- Fully supervised training of robust detectors for all tasks was not feasible.
- Some components (e.g. hazard sign localisation and OCR) had to rely on pretrained or zero-shot models rather than task-specific training.

- The system required stronger reliance on rule-based post-processing to compensate for model uncertainty.

To mitigate this limitation, the team focused on:

- collecting and annotating additional data specifically for coupler and wheel detection,
- using transfer learning wherever possible (e.g. YOLOv8 pretrained weights),
- and designing the pipeline in a modular way so that components could function even with weak or sparse training signals.

Despite these constraints, the system was designed to remain functional across all available test footage, but generalisation to broader real-world conditions remains limited by data availability.

4.2.2 Dependency and Environment Conflicts

- CUDA version mismatches - PyTorch vs PaddleOCR incompatibility - GPU acceleration silently disabled issues

4.2.3 Hardware-Specific Reproducibility Issues

- Different CUDA compute capability - Same requirements.txt behaving differently

4.2.4 Parallelization and Concurrency Limitations

- ThreadPoolExecutor attempt - GIL limitations - GPU resource contention - race conditions in tracker state

4.2.5 Wheel Counting

The most significant challenges in wheel counting arose during the development of the custom tracking algorithm. Three distinct failure modes were identified and resolved iteratively:

- **ID jumping between bogies** — caused by `max_dist` being too large, allowing tracks to match wheels on adjacent bogies. `max_dist` defines the maximum pixel distance within which a new detection can be assigned to an existing track — it is necessary because wheels move between frames and a track must be able to follow them. However, setting it too high causes the matcher to bridge across the gap between bogies. Reducing `max_dist` from 600 to 150 pixels resolved this for the available footage, but this value is inherently video-specific: it depends on train speed, frame rate, and camera distance. A threshold that works for a slow-moving train will cause missed assignments on a faster one, making it impossible to define a single stable value that generalises across all operating conditions.

- **Direction ambiguity** — tracks occasionally matched detections moving against the train direction. Resolved by implementing direction-aware matching that penalises counter-directional assignments with an infinite cost in the Hungarian assignment matrix.

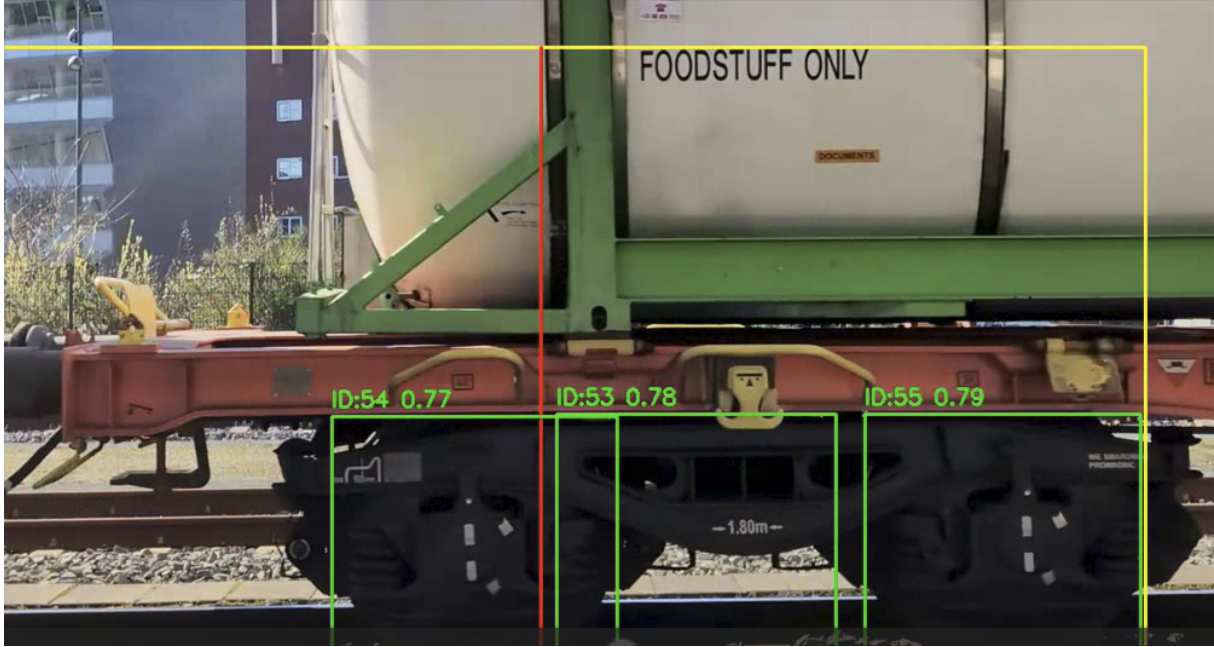


Figure 4.1: Failure of the first trained wheel detection model before dataset change

4.3 Future Suggestions

This section advises improvements/future work for this system from the team members.

4.3.1 coupler detection model

Implement lighting-invariant augmentations: Applying brightness and exposure adjustments will enable the model to generalize across varied environmental conditions, such as direct sunlight or deep shadows, ensuring the coupler remains identifiable regardless of contrast.

Integrate Gaussian noise simulations: Introducing pixel-level graininess forces the model to prioritize the structural "signature" of the coupler over fine textures, maintaining robustness when deployed on low-cost or legacy camera sensors.

Incorporate motion blur effects: Simulating camera vibrations or high-speed movement prevents the model from over-relying on sharp edges, which is essential for maintaining a high recall rate in dynamic real-world track environments.

4.3.2 Axle Count Improvement

Several improvements are recommended for future work:

- **Expand the training dataset** — the current model was trained exclusively on footage collected by the team, limiting its ability to generalise to different track environment.
- **Improve inference speed** — detection speed is the primary bottleneck. Exporting the model to TensorRT or ONNX format and running inference on a dedicated GPU or edge accelerator such as a Jetson device would give the most significant real-world speedup. Switching to a lighter YOLOv8 variant such as YOLOv8n could also reduce per-frame inference time at a modest cost to detection accuracy.
- **Replace DeepSORT with a centroid tracker** — since wheels follow predictable linear motion in a fixed camera setup, a lightweight centroid tracker would reduce per-frame tracking overhead considerably without sacrificing counting accuracy.
- **Broader parameter validation** — the counting logic and tracker parameters were tuned on a limited set of videos. Systematic evaluation across trains of different lengths and speeds is needed to confirm that the current thresholds generalise reliably.

4.3.3 OCR improvements

Even though OCR has high accuracy and practically makes no false positive predictions, there is still room for some improvements. These improvements can be divided into two parts: **external** and **internal**:

- **Internal upgrades:**
 - **Adaptive preprocessing per frame** - before running the base extraction, apply automatic contrast/brightness normalization and sharpening. This can help target missing detections caused by poor lighting or small motion blur; however, it will not be sufficient if the input is too noisy.
 - **Character correction table** - a pre-validation substitution step ($0 \leftrightarrow O; I \leftrightarrow 1, B \leftrightarrow 8$) applied to all recognised strings before regex matching, as the PaddleOCR model reads these characters in a similar way. This would reduce missed detections caused by character confusion rather than actual detection failure. On the other hand, this step can dramatically increase the time needed for OCR extraction .
 - **Fine-tuning on domain-specific data** - PaddleOCR is a general model, which was used due to the lack of data. Collecting a large annotated dataset of wagon markings and fine-tuning the recognition model on it would likely give the largest single improvement,
- **External upgrades:**

- **Camera quality and placement** - using a higher resolution camera with a fixed focal length and consistent mounting position would significantly reduce motion blur and perspective distortion, which are likely the major reason for missing detection in the current footage.
- **Controlled lighting** - installing fixed artificial lighting at the recording site would eliminate the variability caused by weather and time of day, making detection far more consistent without any changes to the pipeline.
- **Trigger-based capture** - instead of recording continuous video and processing every frame, a sensor (e.g. an infrared beam) placed at a fixed point on the track could trigger the camera at the exact moment a wagon passes in front of it. This leaves only the most optimal frames for OCR extraction, eliminating the need to filter through hundreds of frames where the wagon is too far, too close, or at a poor angle.

4.3.4 DINO + CLIP pipeline Improvement

Several directions could improve the DINO+CLIP sign detection pipeline. The most impactful would be replacing GroundingDINO’s open-vocabulary detection with a fine-tuned object detector trained on real wagon crops, which would eliminate the dependency on hand-crafted text prompts and reduce false positives without requiring a CLIP post-filter. The current CLIP zero-shot classifier generalises reasonably across sign types but produces low confidence scores on real-world crops due to the domain gap between training data (stylised Roboflow renders) and actual video frames - fine-tuning CLIP or training a dedicated classifier on a larger labelled dataset of ADR placards would significantly improve reliability. The SVM trained on CLIP embeddings shows strong performance on the dominant flammable class but struggles to generalise to sign types with few training examples, suggesting that data augmentation or few-shot learning techniques could help balance classification across all GHS classes. Additionally, the pipeline currently processes static image folders; integrating it directly into the video stream with temporal aggregation - confirming a detection only when the same sign is seen across multiple consecutive frames - would reduce spurious detections and improve overall precision.

Chapter 5

Work Process

5.1 Scrum and Collaboration

Meetings with the supervisor were held weekly, mostly online, with the final meeting taking place in person at the office. Meetings with the University of Twente supervisor were organized every two weeks during the second half of the project and were conducted in person. These sessions were particularly helpful, as they allowed us to demonstrate progress directly.

The team dynamics were flexible, with members working independently and managing their own schedules. However, this level of freedom required stronger communication and coordination. For major issues, the team held full-group meetings, while day-to-day work was often carried out in pairs or individually. Due to the modular structure of the project, especially in the early stages, it was possible to work with minimal dependency on others. While this improved productivity initially, it later led to challenges during integration, including inconsistencies such as differing model versions.

The team also organized a team-building activity, and overall, members were supportive and willing to share ideas. However, communication-particularly online-could have been more consistent and structured.

5.2 Lessons Learned

- Consider integration from the beginning and clearly define the technology stack.
- Use separate branches for development and avoid merging changes too early.
- Keep teammates informed about your plans and progress to prevent duplication of work or reliance on outdated code.
- Collect a substantial amount of data in order to train models, but also keep a significant part for testing
- Consider constantly updating the README file with dependency changes so there is no version mismatch

Chapter 6

Conclusion

This project set out to design and implement the software analysis component of an intelligent wagon identification system for Strukton, processing side-view video of passing freight wagons to extract UIC codes, wagon length, axle count, GEVI/UN hazard codes, container identifiers, and GHS labels. The resulting pipeline meets that goal across the footage available, combining two custom-trained YOLO detectors with zero-shot components (GroundingDINO, CLIP, PaddleOCR) in a modular two-stage architecture that first segments video into per-wagon frame folders and then dispatches each folder to independent analysis modules. The wheel and coupler detectors both achieved a validation mAP50 of 0.995, and the full system produces structured JSON records per wagon ready for MQTT transmission as specified in the client requirements.

The most significant limitation of the system is a direct consequence of the data available during development. With only around 25 self-collected videos and no pre-existing annotated datasets captured under inspection conditions, several components had to rely on pretrained and zero-shot models rather than task-specific training, and rule-based post-processing was used to compensate for model uncertainty. Parameter values such as the tracker association distance and the motion threshold were tuned against the available footage and would require recalibration for installations with different camera geometry or train speeds. The original 95% accuracy target was relaxed in consultation with the supervisor for this reason, and generalisation beyond the tested conditions remains an open question.

Future work should focus primarily on broadening the data foundation: expanding the training set to cover more wagon types, lighting conditions, and camera installations would allow the zero-shot stages - particularly hazard sign classification - to be replaced with fine-tuned models, and would permit systematic evaluation of tracker and counting parameters across operating regimes.

Bibliography

- [1] Cheng Cui, Fulong Sun, et al. PaddleOCR 3.0 technical report. 2025.
- [2] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics yolov8, 2023.
- [3] Glenn Jocher et al. Ultralytics yolo26, 2025.
- [4] Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, and Lei Zhang. Grounding DINO: Marrying DINO with grounded pre-training for open-set object detection. 2023.
- [5] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Aspell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

Chapter A

Appendices

A.1 Model Performance Metrics

Metric	Validation	Test
mAP50	0.995	0.953
mAP50-95	0.706	0.561
Precision	0.989	0.974
Recall	0.992	0.873
Inference speed	14.1ms/frame (≈ 71 FPS)	
Training time	45 epochs, 1h 3min, Tesla T4	

A.2 Pipelines

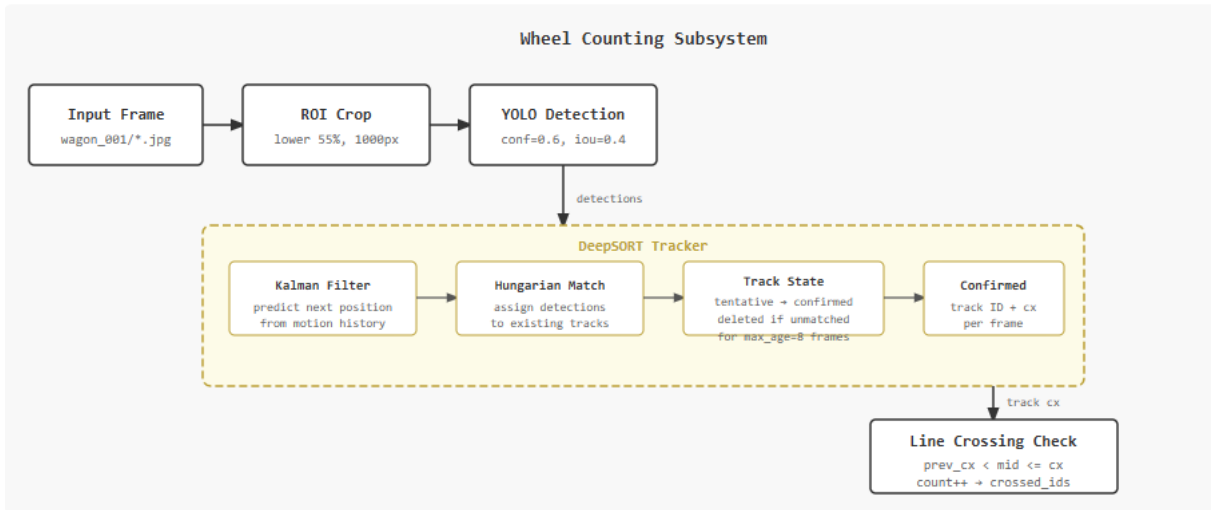


Figure A.1: Wheel counting subsystem flow.

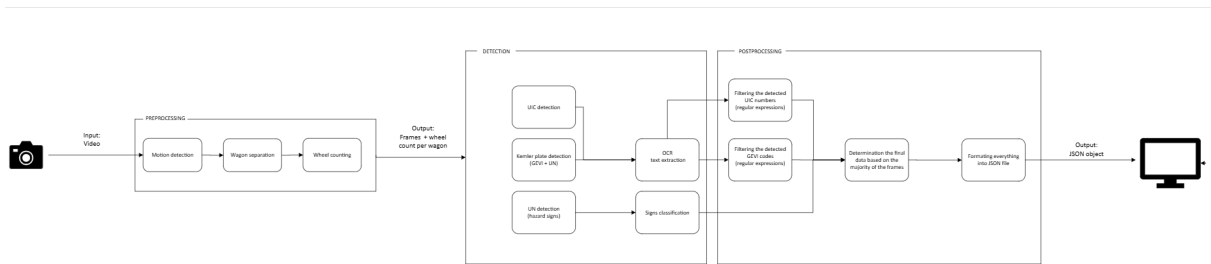


Figure A.2: Old pipeline design, showing the preprocessing stage (motion detection, coupler-based wagon separation and wheel counting), detection stage (UIC, Kemler plate, UN) and postprocessing stage with filtering.

A.3 Data Examples



Figure A.3: Examples of training data used for wheel detection.

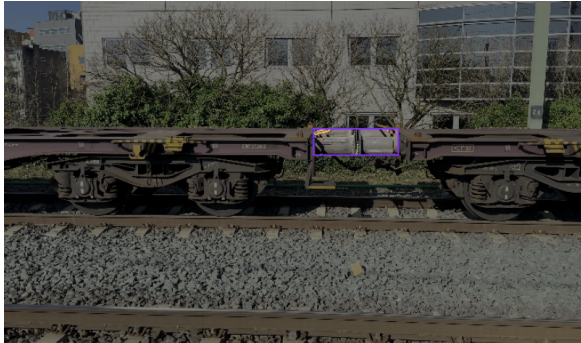


Figure A.4: Examples of training data used for coupler detection.



Figure A.5: Examples of visualization of PaddleOCR output