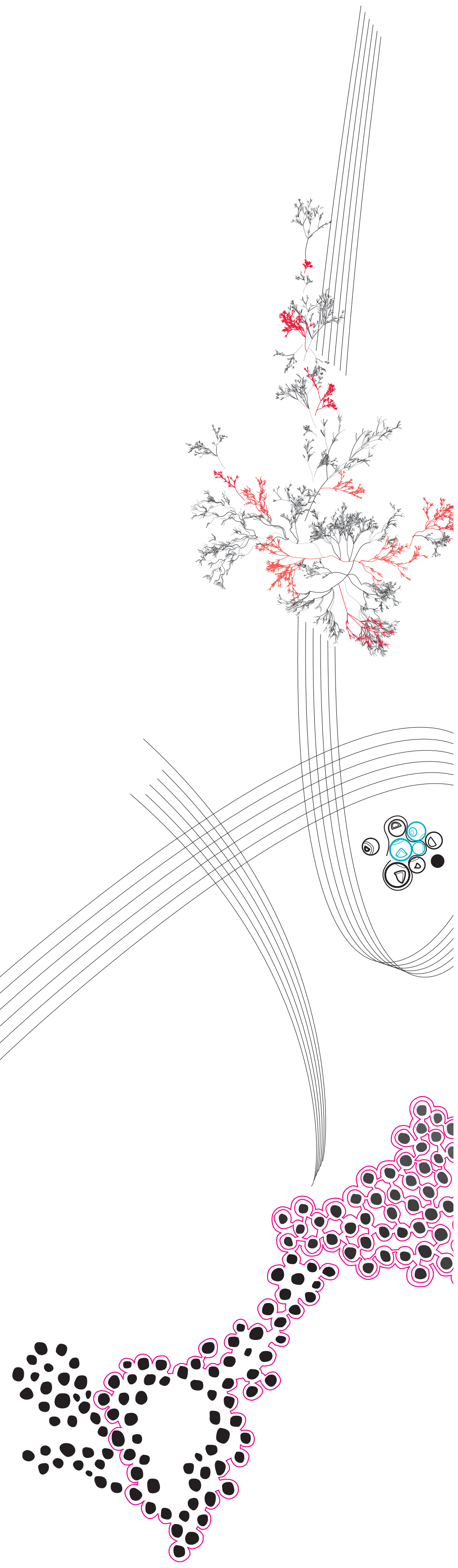
TCS Design Project



Design Report
Visual Level Crossing Monitoring

Amina Chairulina	s3008096
Arturs Visnausks	s2938685
Cristian-Andrei Begu	s2912953
Danil Aliforenko	s2792249
Moamen Elkayal	s2958333
Ksenija Kotliarova	s2943697

Supervisor: Shunxin Wang, Luuk Spreeuwers

April 22, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

This report presents our computer vision system for monitoring railway level crossings, developed as part of the Technical Computer Science Design Project at the University of Twente for the company Strukton Rail. Our system combines computer vision, an API, and a frontend in a scalable microservices architecture. The system detects crossing events, barrier operations, and traffic patterns in real-time while providing alerts and analytics through a web interface. The report discusses the entire design trajectory of the system, from the initial meetings with the client to the specific technical design of each component.

Keywords: computer vision, railway monitoring, level crossings, real-time systems

Contents

Chapter 1

Introduction

1.1 Client profile

The client for our project is Strukton Systems, a division of Strukton Power, operating under the broader Strukton Group. Strukton is a leading European service provider in sustainable infrastructure, with a strong presence in the Netherlands, Denmark, Italy, Sweden, and Belgium. The company focuses on green transport, electrification, and the integration of modern technologies. Strukton is committed to improving the safety, quality and sustainability of infrastructure systems across the rail, road, and energy sectors. Through its Systems division, the company develops advanced mobility and monitoring solutions for both rail and non-rail applications.
The System & Software Engineering department of Strukton Systems, located in Hengelo, is leading the current initiative. They seek to implement a camera-based monitoring solution for all level crossings in the Netherlands, with potential expansion to international markets.

1.2 Project background

The origins of this project date back to 2009. While the overall scope and requirements have undergone only minor adjustments, the primary objective remains consistent: to develop a system that provides comprehensive data related to level crossings. This includes operational information for traffic control, safety and management insights for maintenance contractors and infrastructure managers, as well as records of violations.
Although the original version [2] of the project was successfully developed, its performance was affected by the limited availability of advanced technological solutions at the time and was not yet feasible to develop on commercial scale. With the advancements in technology and the widespread use of visual monitoring systems in recent years, Strukton has decided to revive the concept of a visual level crossing monitoring system, this time integrating modern supporting technologies.
For the current implementation, we were provided with updated project specifications and a live demonstration of the original 2009 version. This served as a valuable reference point and offered a clear understanding of the intended functionality. However, it is important to emphasize that no components from the earlier project were reused. The previous version was employed solely for conceptual inspiration.

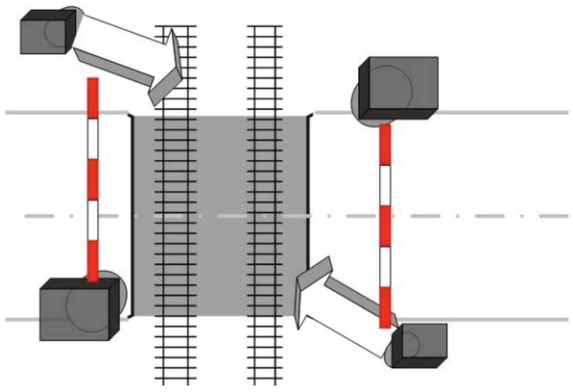


FIGURE 1.1: Crossing with bell

1.3 Scope of the project

The project aims to develop a visual monitoring system that provides real-time data on level crossing activity. Two cameras are installed diagonally on the bell poles at each crossing to ensure full visual coverage of the area (see Figure 1.1). The video stream from these cameras is analyzed using a trained image processing model capable of detecting specific objects and events during the level crossing cycle. These include events such as train passages or barrier movements, and objects such as barriers, signal lights, various types of vehicles, and pedestrians. All detected elements, along with the live video stream, are registered and visualized on a web-based platform, making them easily accessible for Strukton personnel.

The system consists of four main components: image processing, a database, a website, and an API. The image processing module detects events and objects at level crossings in real time. The database stores all relevant information, including object detections, timestamps, and crossing metadata. The website interface provides users with a dashboard to view live streams, event logs, and system statistics. Lastly, the API serves as a communication layer between system components and supports external data access when needed. Remote access through a secure online platform contributes to better decision making, faster maintenance, and better safety and performance at monitored level crossings. It is important to note that the set up of cameras at the crossings and ensuring the functionality of live video stream of installed cameras falls outside the scope of this project.

Chapter 2

Requirements & Timeline

2.1 Requirement specification

In this section, we present the requirements we defined through discussions with the client (Appendix A), categorizing them into user and system requirements. We prioritize these requirements using the MoSCoW method [1] and list them accordingly.

We group the system requirements based on their relevance to the computer vision and website development segments of the implementation. Additionally, we associate them with different components of the level-crossing barrier cycle. Some of these requirements cover level-crossing barriers, trains, vehicles, moving and stationary objects, and average statistics.

To ensure the system meets the needs of its users, we first identify the key stakeholders who will interact with or be affected by the system. Their roles and expectations directly influence both functional and quality requirements.

2.1.1 Stakeholders

- Railway maintenance organizations: Strukton Rail, Volker Stevin, Bam
- Railway managers: ProRail, Other European managers (DB) Railway operators: NS, other European operators (Eurostar, DB, Arriva)
- Government: Municipalities, Police, Traffic police, Provinces, European Union
- Traffic participants: car drivers, cyclists, pedestrians
- Train passengers

2.1.2 User Requirements

1. As a system user, I want to be able to log in and out of the system.
2. As a railway maintenance technician, I want to track the condition of level crossing hardware.
3. As a railway manager, I want to track statistics regarding train traffic at level crossings.
4. As a railway operator, I want to receive live (potential) accident alerts at level crossings.

5. As a government representative, I want to track statistics regarding the traffic around level crossings.

6. As a government representative, I want to track statistics regarding traffic participant misbehavior.

2.1.3 System Requirements

Computer vision

Must Have

1. The system must track the movement and status of barriers, including:
   (a) The incline angle of the barrier.
   (b) The time it takes for a barrier to open and close.
   (c) The alignment of the barrier movement to a given profile.
   (d) The simultaneous movement of all crossing barriers.
   (e) Anomalies in the idle position of the barriers.

2. The system must measure and track the timing of train crossing events, including:
   (a) The time between the barriers closing and the train passing.
   (b) The time between the train passing and the barriers opening.
   (c) The duration while the train is passing.
   (d) The compound time of the entire level-crossing process cycle [1].

3. The system must detect the presence of moving objects and measure:
   (a) Type (car, truck, cyclist, pedestrian).
   (b) Speed, direction, and position of moving objects.

4. The system must detect trains and measure their speed, direction, and length.

5. The system must detect and distinguish stationary objects, including:
   (a) The presence of stationary objects (e.g., lamps) and their types.
   (b) The detection of vandalized or destroyed stationary objects.

6. The system must track traffic light status and detect red riders, including:
   (a) The status of the traffic light (flashing, on, or off).
   (b) The type of red rider vehicles.
   (c) The status of activities[2] during a red light crossing.

[1] The total duration covering all activities when a train passes through the crossing

[2] stage of barrier cycle

Should Have

1. The system must detect track runners and measure their speed and direction.
2. The system should measure the average time (per arbitrary unit time) between:
   (a) The first light signal and the barrier closing.
   (b) The barrier closing and the train passing.
   (c) The train passing and the barrier opening.
   (d) The barrier opening and the last light signal.
3. The system should count the number of:
   (a) Vehicles (per arbitrary unit time, per direction, per vehicle type).
   (b) Traffic participants waiting to cross at red light (per arbitrary unit time, per direction, per vehicle type).
   (c) Trains passing (per arbitrary unit time, per direction).
   (d) Red riders on average (per arbitrary unit time).
   (e) Unexpected stationary objects between the barriers appear on average (per arbitrary unit time)
   (f) Track runners on average (per arbitrary unit time, per direction).
4. The system should track and measure:
   (a) The average speed (per arbitrary unit time, per direction, per vehicle type).
   (b) The average speed of trains (per arbitrary unit time, per direction).
5. The system must measure the average length of trains (per arbitrary unit time, per direction).
6. The system should detect unexpected stationary objects between the barriers.
7. The system should measure how long unexpected stationary objects have been between the barriers.

Could Have

1. The system could measure noise levels, including:
   (a) Traffic loudness.
   (b) Train loudness.
2. The system could measure the average loudness of trains (per arbitrary unit time, per direction).
3. The system could check the functionality of bells, including:
   (a) Checking if a bell rings during each cycle.
   (b) Issuing an alert if not all four bells ring as barriers close.
   (c) Issuing an alert if not all two bells ring as barriers stay closed.
4. The system could issue alerts in the following situations:
   (a) If train noise levels exceed a threshold.
   (b) If traffic noise levels exceed a threshold.
   (c) In case of poor visibility.

Won't Have

1. The system won't process information from other rail infrastructure.

Website

Must Have

1. The system must include authentication functionality, allowing users to:
    (a) Log in.
    (b) Log out.
2. The system must issue alerts for critical events, including:
    (a) If a barrier is open during a train passing.
    (b) If a barrier is closed while no train is expected.
    (c) If barriers move unevenly.
    (d) If a train passes while the barrier is still closing.
    (e) If barriers stay closed longer than expected after a train has passed.
    (f) If there is an unexpected stationary object on the track.
    (g) If an expected object has disappeared, been vandalized or broken.
3. The system must facilitate the management of level crossings, including the ability to:
    (a) List level crossings.
    (b) Add new level crossings.
    (c) Remove level crossings.
4. The system must provide live video monitoring of level crossings.
5. The system must enable video playback of level crossings.

Should Have

1. The system should display alerts from all level crossings in one feed

2.1.4 Quality Requirements

To ensure reliability, usability, and performance, the system must adhere to the following non-functional requirements:

Performance

1. The system must process and analyze video feeds in real-time, with a maximum latency of 3 seconds.
2. The website must load dashboard data within 3 seconds under normal network conditions.

Reliability & Availability

1. The system must achieve 99.9% uptime, excluding scheduled maintenance.
2. Alerts for critical failures (e.g., barrier malfunctions) must be delivered within 5 seconds of detection.

Usability

1. Alerts must be visually distinguishable (e.g., color-coded).

Scalability

1. The computer vision subsystem must support adding up to 50 additional level crossings without requiring architectural changes.
2. The database must handle at least 10,000 events per hour during peak traffic.

Maintainability

1. The codebase must include modular documentation for all major components.
2. Configuration changes (e.g., alert thresholds) must be adjustable without system downtime.

2.2 Timeline

Our project followed a structured 7-week timeline from initial requirements to final delivery. We organized the work into sequential phases with overlapping tasks where appropriate. At the beginning of each week, we organized a SCRUM meeting with the client in order to receive feedback and gather new requirements where necessary. The project timeline was divided as follows:

- **Week 9**: Requirements specification and project proposal finalization
- **Week 10**: Computer vision pipeline setup, API design, and database architecture
- **Week 11**: Labeled dataset creation, post-processing algorithm development, and initial frontend implementation
- **Week 12**: Model training and fine-tuning, frontend-backend integration
- **Week 13**: API-model integration, documentation completion, and post-processing algorithm refinement
- **Week 14**: System-wide testing, performance optimization, and bug fixes
- **Week 15**: Final delivery of system components and project report

Figure 2.1 illustrates our project schedule using a Gantt chart, showing task dependencies and parallel work streams:

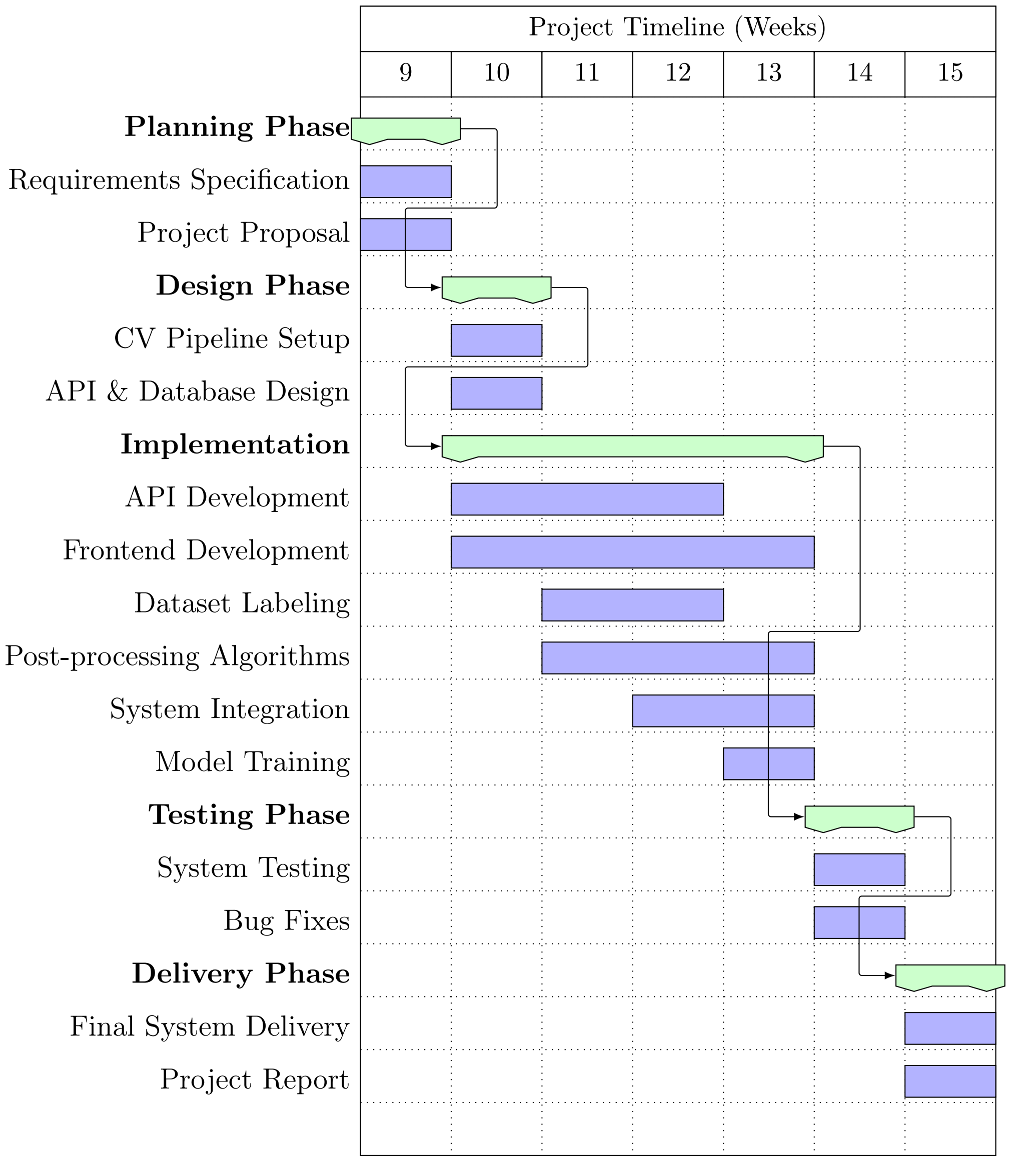


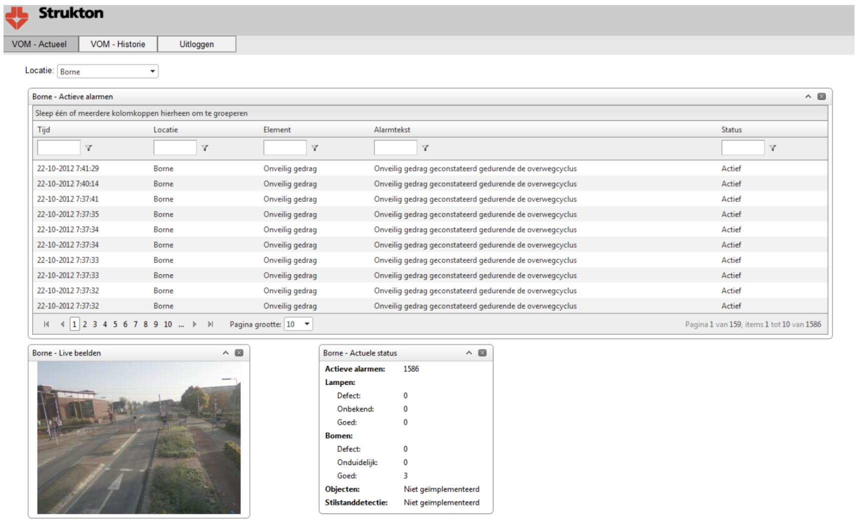
FIGURE 2.1: Project Gantt Chart showing task dependencies and timeline

Chapter 3

Design

3.1 User-facing design

In this section, we present, on the basis of the requirements, the user facing side of the system. Since an attempt at developing the system was carried out before (see Figure 3.1), some of the clients expectations regarding the user interface were already established.



Locatie: Borne

Borne - Actieve alarmen

Sleep één of meerdere kolomkoppen hierheen om te groeperen

Tijd	Locatie	Element	Alarmtekst	Status
22-10-2012 7:41:29	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief
22-10-2012 7:40:14	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief
22-10-2012 7:37:41	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief
22-10-2012 7:37:35	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief
22-10-2012 7:37:34	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief
22-10-2012 7:37:34	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief
22-10-2012 7:37:33	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief
22-10-2012 7:37:33	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief
22-10-2012 7:37:32	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief
22-10-2012 7:37:32	Borne	Onveilig gedrag	Onveilig gedrag geconstateerd gedurende de overwegcyclus	Actief

1 2 3 4 5 6 7 8 9 10 ... Pagina grootte: 10 Pagina 1 van 159, items 1 tot 10 van 1586

Borne - Live beelden

Borne - Actuele status

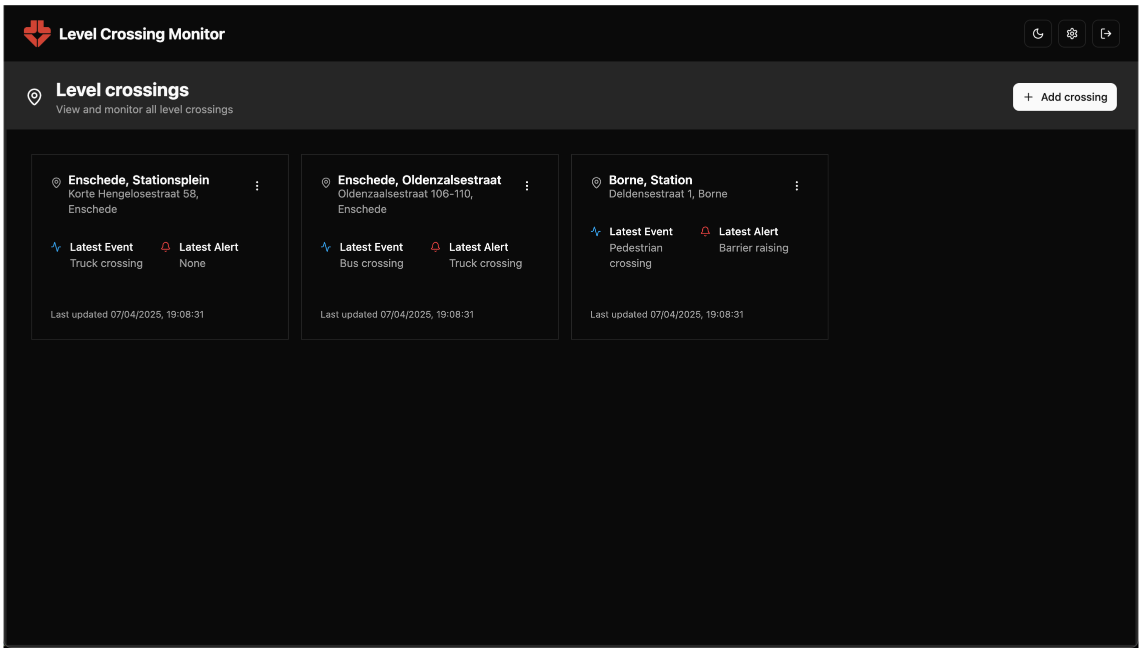
Actieve alarmen:	1586
Lampen:	
Defect:	0
Onbekend:	0
Goed:	0
Bomen:	
Defect:	0
Onduidelijk:	0
Goed:	3
Objecten:	Niet geïmplementeerd
Stilstanddetectie:	Niet geïmplementeerd

FIGURE 3.1: Old user interface

Based on the previous iteration, we have created a modern interpretation with the same functionality. The user interface is comprised of 3 main parts: the home page, the crossing page and the user administration page.

3.1.1 Home page

Since the old design did not have an established home page, we had complete liberty in designing it. The main goal with the design was to let users see information about many level crossings at once, without needing to dig deeper into the crossing page. After a discussion with our client (Appendix A), we settled on a grid of crossings, with each displaying the latest event and alert. This design can be seen in Figure 3.2



Enschede, Stationsplein
Korte Hengelosestraat 58, Enschede
Latest Event: Truck crossing
Latest Alert: None
Last updated 07/04/2025, 19:08:31

Enschede, Oldenzalsestraat
Oldenzaalsestraat 106-110, Enschede
Latest Event: Bus crossing
Latest Alert: Truck crossing
Last updated 07/04/2025, 19:08:31

Borne, Station
Deldensestraat 1, Borne
Latest Event: Pedestrian crossing
Latest Alert: Barrier raising
Last updated 07/04/2025, 19:08:31

FIGURE 3.2: Homepage design

Besides the home page, the old design also did not support a way for users to add new crossings, so an entirely new crossing setup flow had to be designed. After some discussion with the client on what can (not) be manually done (Appendix A), we settled on allowing the user to add any WebRTC stream and place 4 points to designate the crossing area (as seen in Figure 3.3). Following this setup, all recognition is done automatically with no user input.

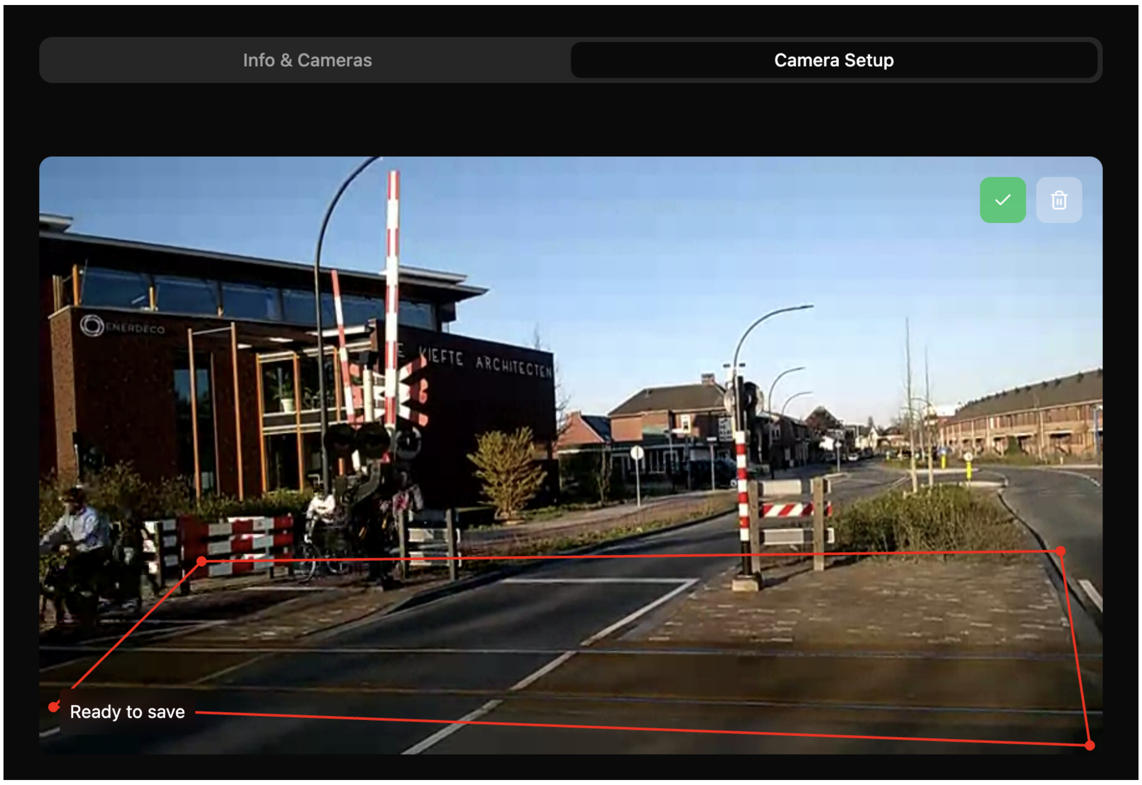
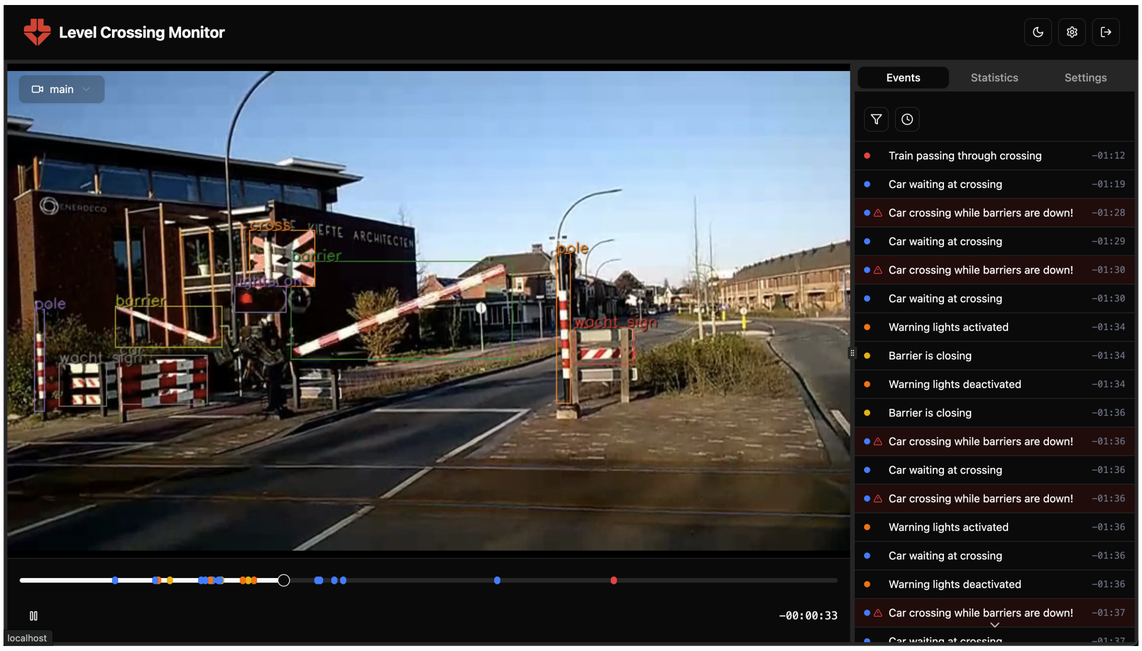


Figure 3.3: Camera setup screen

3.1.2 Crossing page

The crossing page was already designed in the previous of the system, but for lower resolution cameras. Taking into account modern hardware, and our machine learning based system, we decided on a new design incorporating a large camera view. The event list present in the previous iteration has now been shifted to the right of the page, and uses color coding to make events easy to spot. Besides that, filtering options have been added to allow the user to easily browse events. An overview of the page can be seen in Figure 3.4.



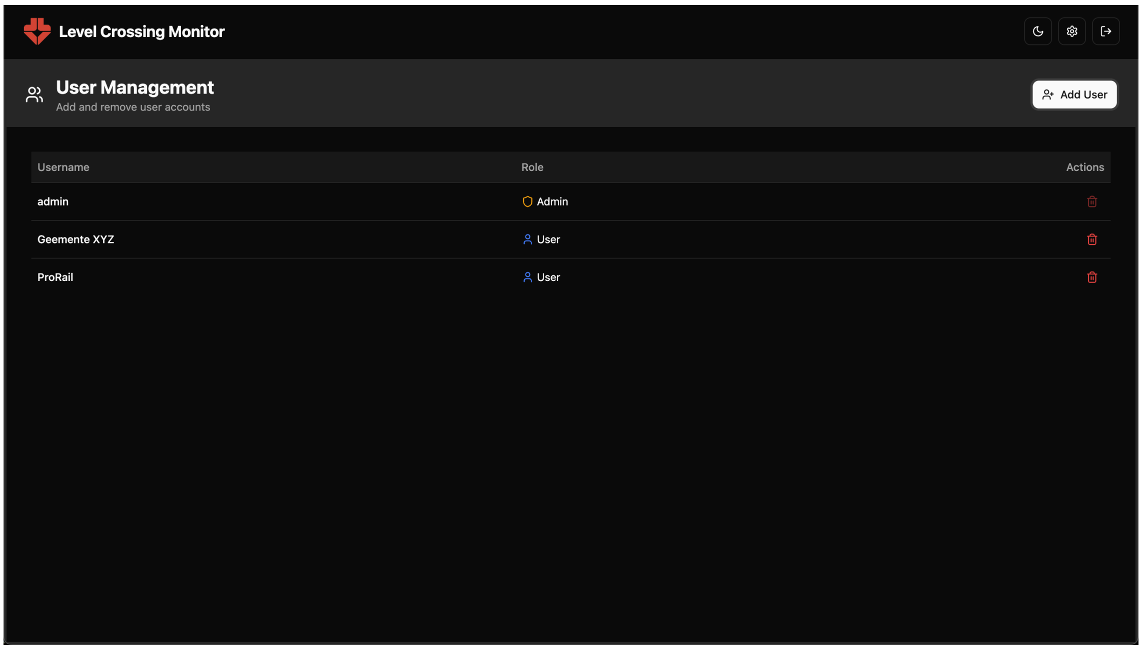
```
● Train passing through crossing                    -01:12
● Car waiting at crossing                           -01:19
● △ Car crossing while barriers are down!           -01:28
● Car waiting at crossing                           -01:29
● △ Car crossing while barriers are down!           -01:30
● Car waiting at crossing                           -01:30
● Warning lights activated                          -01:34
● Barrier is closing                                -01:34
● Warning lights deactivated                        -01:34
● Barrier is closing                                -01:36
● △ Car crossing while barriers are down!           -01:36
● Car waiting at crossing                           -01:36
● △ Car crossing while barriers are down!           -01:36
● Warning lights activated                          -01:36
● Car waiting at crossing                           -01:36
● Warning lights deactivated                        -01:36
● △ Car crossing while barriers are down!           -01:37
● Car waiting at crossing                           -01:37
```

FIGURE 3.4: Crossing page design

We have also brought the previously separate event details, statistics, and settings page into the sidebar, allowing the user to see the camera context while browsing them. As improvements over the old design, we have added a timeline with marked events as well as classification boxes over the stream. We designed these features in order to give a tighter integration between the events and the video stream.

3.1.3 User administration page

At around the half-way mark in development, the client requested that we implement an interface for managing different users (Appendix A). Since this page is only available to "admin" users, and no equivalent existed before, we simply designed it as a table with user information. The page is accessible at all times through a button in the top-right controls. The design of the page can be seen in Figure 3.5



Username	Role	Actions
admin	Admin	
Geemente XYZ	User	
ProRail	User	

FIGURE 3.5: User management page design

3.2 Technical architecture

In this section, we present the high-level technical architecture of our system and explain our technology choices and service interactions. We designed the project following a microservices architecture where each component functions as a self-contained, loosely coupled service that communicates over a shared message bus. We based this architectural decision on three key principles:

- **Scalability** — Our system serves as a real-time computer vision solution that should, eventually, be able to scale to over 2,500 crossings [4]. This requirement led us to choose a solution that allows for horizontal scaling, where we can distribute load across multiple instances of the same service (like database or computer vision instances). Unlike vertical scaling, this approach would let us scale each service independently, optimizing resource allocation.
- **Separation of Concerns** — We assigned each service a single responsibility, enabling our team members to work on different services simultaneously. This approach significantly accelerated our development process, which proved crucial for meeting our tight development timeline.
- **Technology Agnosticism** — Since we designed the services to operate independently, we could select the most appropriate technology for each one. For example, we chose Python for machine learning tasks due to its robust ecosystem, while we implemented the API in .NET to meet our client's preference. Our microservices architecture allowed us to integrate these different technologies seamlessly.

For our infrastructure, we chose Docker to containerize each service and Docker Compose to orchestrate them. We also implemented these key infrastructure components:

- **Kafka Message Queue** — We use Kafka as our message broker for inter-service communication. It enables our event-driven architecture where services react to events like new camera frames or crossing events in real time. Kafka also helps us achieve scalability by facilitating load balancing and message distribution.
- **TimescaleDB** — We selected this time-series database (built on PostgreSQL) because it specializes in storing and querying timestamped data. Since we timestamp all detection events, TimescaleDB became the natural choice for our application.

Our system consists of these core services:

- **Backend API** — We built this central API to manage level crossing data and provide endpoints for the frontend and other services. Following our client's preference, we implemented it in .NET.
- **Frontend** — We developed this user interface to monitor and manage level crossings while displaying real-time and historical data. We chose React because of our team's familiarity with it and its extensive ecosystem.
- **Computer Vision Service** — This service processes video frames to perform image analysis and detect events at level crossings. We implemented it in Python to leverage its strong machine learning ecosystem.
- **Frame Producer** — We designed this component to ingest and distribute video frames from connected webcams to other components. For development simplicity, we also wrote this service in Python.

Figure 3.6 illustrates our final architecture.

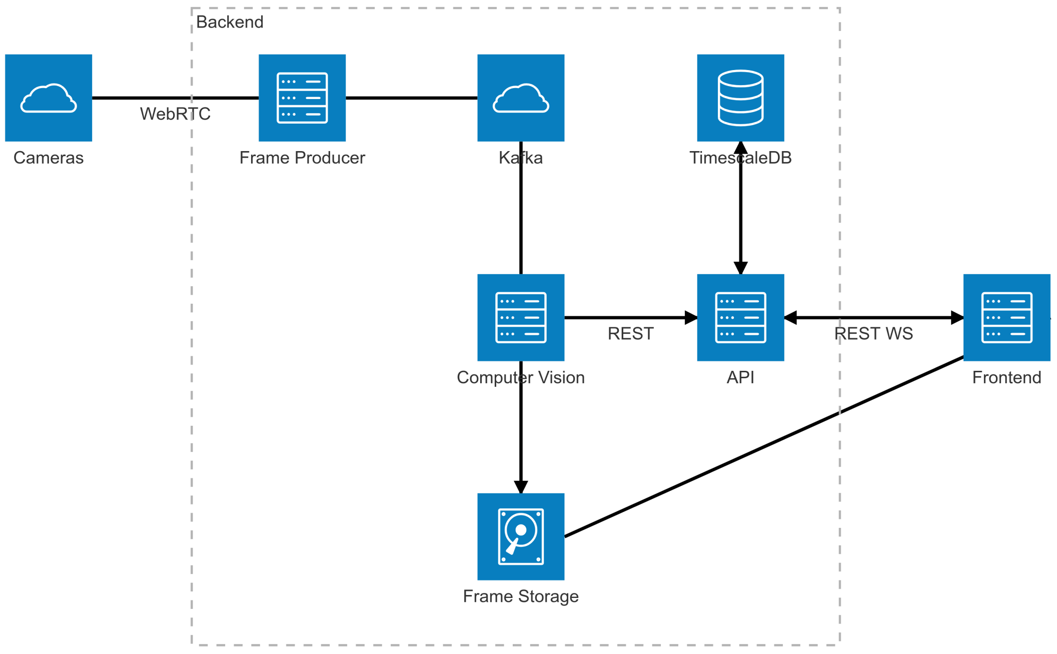


FIGURE 3.6: System architecture diagram

In Figure 3.7, we show how our components interact. Our frame producer serves as the primary data ingestion point, collecting frames from connected cameras (using a list provided by our API). We then send these frames to our computer vision (CV) service, which classifies and processes them into events. Our CV service forwards event data to the API while also providing a video stream for the frontend. Finally, our frontend communicates with the API to display statistics and event history.

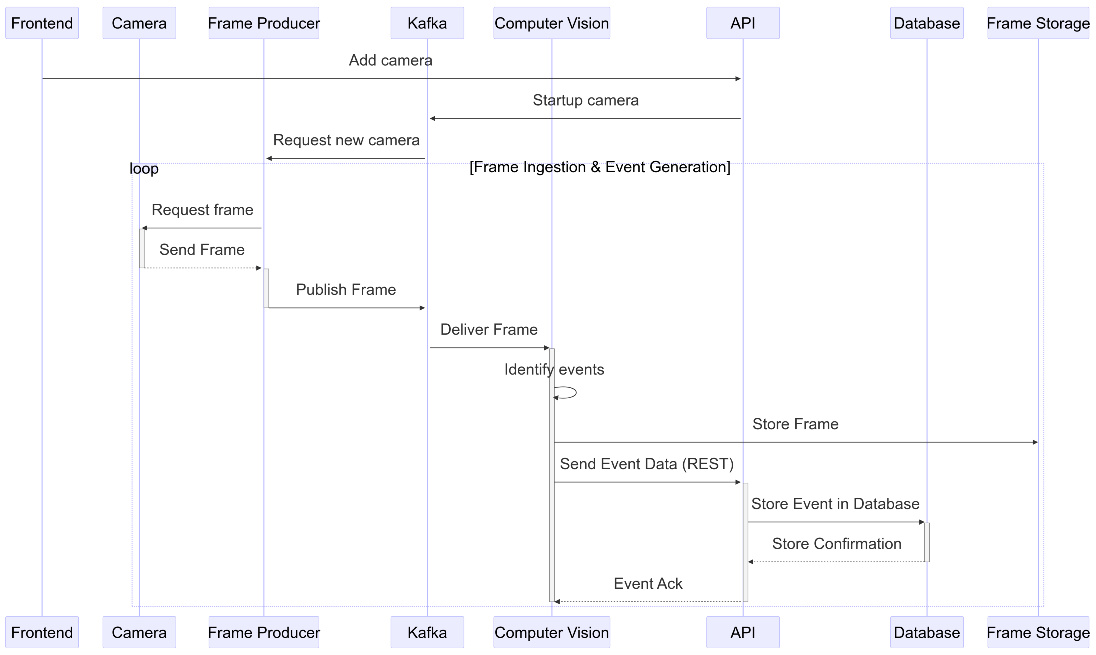


FIGURE 3.7: Sequence diagram of camera data flow

3.3 Detailed technical design

3.3.1 Database

The system's database is designed to efficiently store, query and manage data related to level crossings, users, cameras, and different event types that may occur at those crossings. The design was driven by functional requirements such as event classification, timestamp-based querying, user management with role differentiation and efficient aggregation for statistics.

Schema Overview

The database schema, shown in Figure 3.8, was designed to cover both the functional requirements of the system and the stakeholder expectations mentioned in Section 2.1. Through the design we tried as much as possible to make it as modular as we can and to have scalable for more event detection and more level crossings.

User management is handled through the `User` and `UserRole` tables. Each user is assigned a specific role (e.g., `User` or `Admin`), which enforces access control and satisfies Requirement 1 for the website.

The core table is `LevelCrossing`, which represents all level crossings that are being monitored. Each level crossing is associated with one or more `Camera` instances responsible for capturing real-time footage and providing bounding boxes for analysis. This setup supports requirements 2 and 3 for the website as well as Computer Vision requirements that mention track runners. The `VideoStream` field in the `Camera` class allows live streaming and playback on the web interface (Website Must Haves 4 and 5).

In the event processing system lies in the `Event` table, which records detected incidents and associates them with a specific level crossing and timestamp. Each event also includes metadata such as duration, label, confidence score, and alert status. The `CycleGroup` field enables grouping of events that occur within the same train crossing sequence (lights on, barrier lowering, train passing, etc.). The `EventType` enumeration determines which kind of alert this is: `Barrier`, `Train`, `Vehicle`, `Light`, and `Obstacle`.

Each event subtype inherits from the base `Event` table and has its own attributes. For instance, `BarrierEvent` records movement curves and barrier states (`Open`/`Closed`), directly correlating to requirement 1 in Computer Vision. `TrainEvent` stores train-specific attributes such as length, speed, and direction (Must Have 2), while `VehicleEvent` captures vehicle type, position, speed, and direction (Must Haves 3-5). The `VehicleType` enumeration ensures a consistent classification of object types.

`LightEvent` supports traffic light state logging to detect red light violations or timing anomalies (Must Have 6). `ObstacleEvent` enables detection of static obstructions on or near the tracks. Enumerated types such as `BarrierEventType`, `LightEventType`, and `EventType` enforce data integrity and clarity throughout the schema.

To avoid unnecessary complexity and ensure flexibility, all statistics required by the system (average event durations, alert frequencies or counts of vehicles, etc.) are calculated on demand rather than being stored in a table. Implementing a table just for statistics would require constantly updating it through a watcher or a background job which would introduce overhead and increase the systems' complexity. By computing statistics dynamically, the system avoids such problems.

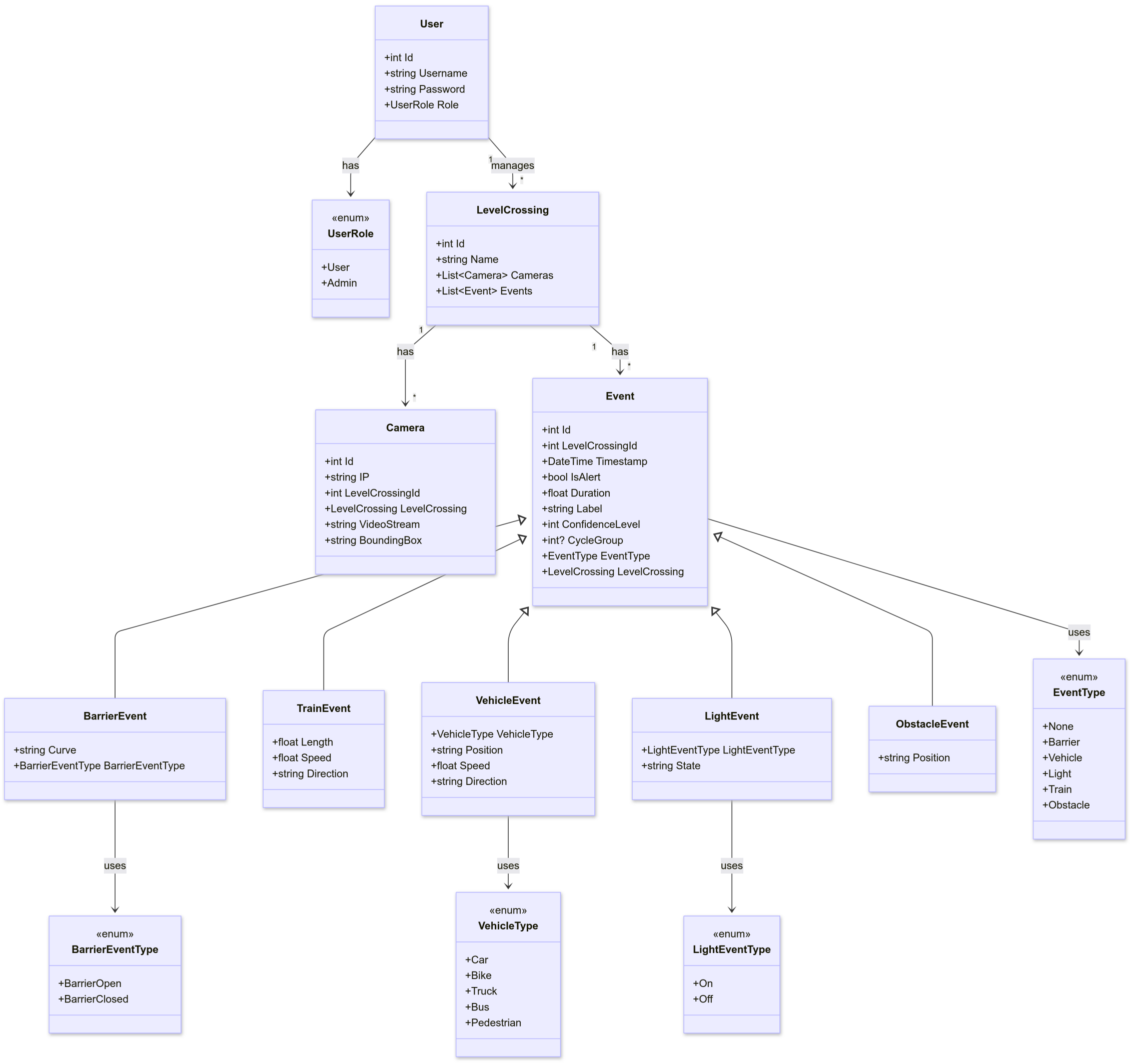


Figure 3.8: Entity-Relationship Diagram of the Level Crossing Database

Use of TimescaleDB and EF Core

To efficiently handle time-series data such as events, we use **TimescaleDB**, a PostgreSQL extension optimized for high-performance time-based querying. Specifically, the `Event` table is defined as a **hypertable**, enabling the system to scale efficiently while supporting real-time statistics and fast access to large volumes of timestamped records. The hypertable allows partitioning and indexing based on the timestamp, improving the performance of queries that filter or aggregate over time ranges.

Entity Framework Core (EF Core) is used as the Object-Relational Mapper (ORM). EF Core abstracts away most of the database interaction logic, allowing us to interact with the database using C# objects and LINQ queries. EF Core automatically manages the creation of primary keys, foreign keys and indexes on relevant columns, ensuring optimized query performance without manual index definitions.

Inheritance Strategy

The design uses the **Table-per-Hierarchy (TPH)** inheritance strategy for the `Event` class and its subtypes. In the TPH approach, all event types share a single table in the database with an added **discriminator column** to distinguish between different subtypes (e.g., BarrierEvent, VehicleEvent, etc.). This discriminator is automatically handled by EF Core.

We chose TPH for the as TimescaleDB requires hypertables to be single physical tables and does not support table inheritance across hypertables. TPH is compatible with such a constraint, while other inheritance models are not. Moreover, alternative inheritance strategies like Table-per-Type (TPT) or Table-per-Concrete-Class (TPC) require separate tables for each subclass, which breaks the concept of a unified hypertable and introduces join overhead during queries. Also, storing all events in a single table simplifies event aggregation, statistics computation and querying across different types. For example, calculating average duration or counting all alerts regardless of subtype is significantly faster when data is in a single table.

Lastly but not least, EF Core takes care of creating the necessary indexes on the discriminator and timestamp columns which improves the speed of querying events over time or by type.

3.3.2 API

Designed to align with the database schema described earlier, the API ensures seamless interaction between structured data storage and application functionality. Its architecture directly reflects the relational model of level crossings, cameras, events, and users, enabling efficient querying and data manipulation. Additionally, for some models we introduce DTOs (Data Transfer Objects) that exclude irrelevant or private information.

The API integrates database, front-end, and computer vision services, establishing error-free and secure data exchange. Built using ASP.NET Core framework and C# programming language, this RESTful API implements MVC (Model-View-Controller) architecture pattern. Each controller represents a set of endpoints grouped by the aspect of the system they act on. Our API consists of the following controllers:

- **User controller** - creation and authentication of users.
- **Level crossing controller** - interactions with level crossings and the cameras attached to them.
- **Event controller** - interactions with events of a level crossing.
- **Statistics controller** - collection of statistical data for a level crossing.
- **WebSocket controller** - establishment of WebSocket connections for event streaming.

All defined endpoints in the above-mentioned controllers return data in JSON format. Error handling functionality is implemented to ensure that appropriate HTTP response codes with concise error messages are returned to the client in case of an error, without exposing the system. Once an endpoint is reached, the controller then uses one of the underlying service classes that interact with the EF Core.

User controller

This controller provides endpoints that enable the retrieval, creation, deletion, and authentication of users in the system. To ensure only authorized access to other parts of the system, it utilizes JWT-based authentication and role-based access control. A special role, Admin, is defined in the system. Apart from accessing all other endpoints, Admin users are additionally allowed to create and delete other users in the system. Table 3.1 provides an overview of the endpoints defined in this controller.

Table 3.1: User controller API Endpoints

Endpoint	**Method**	**Description**
`/users`	GET	Retrieves all registered users in the system. Returns a list of user DTOs.
`/users/{id}`	DELETE	Deletes a specific user by their ID. Returns 204 No Content on success.
`/users/register`	POST	Registers a new user. Requires username and password in the request body.
`/users/login`	POST	Authenticates a user and returns a JWT token with their role. Public endpoint.

Level crossing controller

This controller manages the level crossings and the cameras associated with them. It allows to create, retrieve, update and delete level crossings and cameras. Once a camera is added or updated, a message is sent to the Kafka broker to initiate video stream processing. Table 3.2 provides an overview of the endpoints in this controller.

Table 3.2: Level crossing controller API Endpoints

Endpoint	**Method**	**Description**
`/crossings`	GET	Retrieves all level crossings in the system
`/crossings`	POST	Creates a new level crossing with associated data
`/crossings/{id}`	GET	Retrieves a specific level crossing by ID
`/crossings/{id}`	PUT	Updates an existing level crossing's data
`/crossings/{id}`	DELETE	Deletes a specific level crossing
`/crossings/{id}/cameras`	GET	Retrieves all cameras for a specific level crossing
`/crossings/{id}/cameras`	POST	Adds cameras to an existing level crossing
`/crossings/{id}/cameras/{cameraId}`	PUT	Updates a specific camera at a level crossing
`/crossings/{id}/cameras/{cameraId}`	DELETE	Deletes a specific camera from a level crossing

Event controller

This controller manages events from a level crossing. It allows retrieving and creating a single event, as well as getting all events of a crossing. Due to the potentially large and continuously updated amount of events, event list retrieval endpoint supports pagination together with various filters such as event type, date range and an alert filter. Furthermore, upon event creation the controller sends the event data to all established WebSocket clients for the crossing. An overview of this controller can be seen in Table 3.3.

TABLE 3.3: Event controller API Endpoints

Endpoint	**Method**	**Description**
`/crossings/{crossingId}/events`	GET	Retrieves paginated events with optional filters (type, date range, alerts). Returns pagination metadata.
`/crossings/{crossingId}/events`	POST	Creates a new event (supports multiple event types). Triggers WebSocket notifications.
`/crossings/{crossingId}/events/{id}`	GET	Retrieves a specific event by ID.

Statistics controller

This controller allows retrieving the following statistics calculated for a specific level crossing:

- Timing statistics:
  - Average time between the lights turning on and barriers closing
  - Average time between barriers closing and train passing
  - Average time between train passing and barriers opening
  - Average time between barriers opening and lights turning off
  - Average time the crossing is closed
  - Amount of crossing closures
- Vehicle statistics:
  - Total amount of traffic participants going through the crossing
  - Total amount of traffic participants waiting at the crossing
  - Maximum amount of traffic participants waiting at the crossing
  - Total amount of traffic participants crossing while the lights are on
  - Total amount of unexpected objects detected on the track
  - Total amount of cars detected at the crossing
  - Total amount of trucks detected at the crossing
  - Total amount of bikes detected at the crossing

  - Total amount of pedestrians detected at the crossing
  - Total amount of buses detected at the crossing

- Vehicle speed statistics:
  - Average car speed
  - Average bike speed
  - Average truck speed
  - Amount of speed limit violations
  - Average speed over the limit
  - Maximum recorded speed

- Train statistics:
  - Total amount of trains
  - Average train speed
  - Average train length

All of the above-mentioned statistics can be fetched for the last 24 hours, week or month. An overview of the endpoints defined in this controller can be found in the Table 3.4.

Table 3.4: Statistics controller API Endpoints

Endpoint	**Method**	**Description**
`/crossings/{crossingId}/statistics/timings`	GET	Retrieves timing statistics
`/crossings/{crossingId}/statistics/vehicles`	GET	Retrieves vehicle statistics
`/crossings/{crossingId}/statistics/speed`	GET	Retrieves vehicle speed statistics
`/crossings/{crossingId}/statistics/trains`	GET	Retrieves train statistics

WebSocket controller

This controller processes incoming requests for WebSocket connections that are later used for real-time event streaming. It upgrades HTTP requests to persistent WebSocket connections, requiring JWT authentication. Table 3.5 contains an overview of the endpoint.

Table 3.5: WebSocket controller API Endpoints

Endpoint	**Method**	**Description**
`/crossings/{crossingId}/events/subscribe`	GET	Establishes WebSocket connection for real-time event streaming. Requires JWT in `Sec-WebSocket-Protocol` header.

3.3.3 Computer Vision

The computer vision service is designed to process video streams from Kafka, perform object detection and tracking, analyze object behavior to generate events, and interact with an API. The system is modular, handling multiple video streams concurrently and generating standardized event data. Overall, it operates through a sequential pipeline:

1. **Initialization and Managemen:** Discovers Kafka video topics for each connected camera and starts dedicated stream processor for each topic. When a new camera added, it will dynamically discover new topic and will start processing.
2. **Stream Consumption:** Each processor instance consumes raw video frames from its assigned Kafka topic and then passes them to object detection.
3. **Object Detection:** Utilizes a YOLO model (`ultralytics`) to detect various objects within each decoded frame, returning bounding boxes, class labels, and confidence scores.
4. **Object Tracking:** Employs the `Norfair` library to track detected objects across consecutive frames. It assigns unique IDs and uses class-specific tracker configurations, including appearance-based Re-Identification (ReID) where needed.
5. **HLS Output:** Draws bounding boxes, classes and IDs onto the frame and pipes the processed frame data to an FFmpeg subprocess, which encodes it into an HLS (HTTP Live Streaming) stream playlist stored locally. The volume is shared with Frontend container.
6. **Event Handler Delegation:** It passes the list of currently tracked objects (from Norfair) to its dedicated Event Handler instance.
7. **Event Detection:** The Event Handler delegates the analysis of tracked objects to specialized handler modules based on the object's class label. These handlers maintain object state, analyze movement/position, and detect state transitions based on predefined rules.
8. **Event Contextualization:** When handlers report detected state transitions, they are passed to this module, which maintains the overall crossing cycle state (e.g., if barriers are down, lights are active) and contextualizes the incoming transition information.
9. **Event Production:** This module formats the contextualized information into standardized event objects. It then publishes it as a message to a dedicated Kafka topic.
10. **Event Consumption and API interaction:** A separate process/service running that subscribes to the event topic on Kafka and consumes the generated event messages. It then sends the consumed event data to a backend API via an HTTP POST request to the appropriate endpoint.

Object recognition

The primary role of the object detection component is to identify the presence and location of various objects of interest within each individual video frame processed from the Kafka stream. This forms the foundational input for the subsequent tracking and event analysis stages. To train the model, we needed to create our own dataset:

- **Primary Dataset:** We meticulously gathered a core dataset of 872 images from Dutch level crossing. Primary sources for that were videos, provided by the client, recordings of level crossings from YouTube and the ones that we made ourselves.
- **Annotations:** This dataset contained 8372 annotations distributed across 10 distinct classes (averaging approximately 10 annotations per image).
- **Data Enrichment:** We attempted to incorporate background images during training to help the model generalize better and distinguish foreground objects. We have also sourced additional images, particularly for classes that did not have high representation (such as cars, pedestrians), from the standard COCO dataset.
- **Dataset Split:** The combined dataset was divided into standard subsets for training (70%), validation (20%), and testing (10%), according to suggestions from `ultralytics` wiki.
- **Augmentation:** The effective size of the training dataset was then tripled (3x) through various pre-processing and data augmentation techniques (e.g., rotations, brightness changes, scaling) via `RoboFlow` platform.

Then, we trained a YOLOv12 custom model using this dataset. Metrics and its performance will be discussed further in the report in the evaluation section. Object detection flow happens as follows:

1. The function receives a single video frame.
2. It executes the loaded YOLO model on the frame. Parameter `conf=0.7` sets a confidence threshold; we found that our model produces consistent results at that threshold.
3. The results are iterated, extracting bounding box information, class ID, and confidence score for each detected object.
4. We decided to filter out detections identified as `"lights off"` at this step. This is a deliberate choice to ignore this specific class, because its absence was easier to handle in the Event Framework than its presence.
5. The coordinates, confidence score, and class name for each valid detection are compiled into a list, and then passed to the Object Tracking module.

Object Tracking

While YOLO detects objects in individual frames, the tracking component, implemented using the `Norfair` library, is responsible for associating these detections across consecutive frames. Its goals are to assign a consistent, unique ID to each distinct object instance, estimate its state (position) smoothly over time, and maintain the object's identity even through temporary occlusions. It also handles many cases of false positives or false negatives produced by YOLO. We implemented specific trackes with different ReID (re-identification) and tracking parameters for different classes to fine tune the detection process.

1. The tracker module takes the list of YOLO detections and the current video frame as input.

2. **Class-Specific Tracker Configuration:** A key feature of this implementation is the use of multiple `norfair.Tracker` instances, each configured differently for specific groups of object classes. T:
   - We implemented separate trackers for different object types. Rationale here is that static objects barely move, barriers have constrained movement, trains move quickly, and vehicles/pedestrians have more varied motion. Tailoring tracker parameters optimizes performance for each type.
   - Parameters adjusted per tracker type include:
     - `Distance Function`/`Distance Threshold`: How similarity between detections and tracks is measured (e.g., IoU, centroid distance) and the maximum allowed distance for a match. Static objects have a stricter (lower) threshold, as they are not supposed to move. We expect barriers to move slowly in small increments too, while, for example, cars move quicker.
     - `Initialization Delay`: The number of consecutive frames an object must be detected before its track is considered "live" (initialized). Barriers or static objects have longer delays to avoid spurious tracks, in case when YOLO hallucinated them where it should not have. Similarly, trains have minimal initialization delay, because sometimes they pass through the crossing in just a couple of frames.
     - `Hit Counter`: How many frames a track can persist without being matched to a new detection before being considered "dead". Longer values help bridge occlusions and camera artifacts.
     - `ReID parameters`/: Parameters for appearance-based Re-Identification (discussed below).

3. **Re-Identification (ReID)**:
   - ReID helps re-associate a track with its corresponding object based on visual appearance, especially useful when objects reappear after occlusion or cross paths.
   - When a detection is assigned to a ReID-enabled tracker, it then extracts the image patch corresponding to the bounding box from the current frame. Tracker then calculates a color histogram rom this patch. This histogram serves as the visual embedding for the detection.
   - The `IoU` (Intersection over Union) function is used during the tracking update. It compares the histogram embedding of a potential detection match against the historical embeddings stored within existing, currently unmatched tracks. A match below threshold suggests that the detection belongs to that track.

4. **Tracking Logic:**
   - Detections are grouped based on the mapping.
   - If applicable, the ReID embedding is calculated and added to the Detection.
   - The function collects the list of currently tracked objects returned by all tracker instances.

5. Output of this step is a unified list of Tracked Objects. Each object contains valuable information like its unique assigned `id`, its estimated bounding box, its `label`, its age and hit counts, and its lifecycle status.

Rule-based framework

The Event-Based Framework translates the low-level information about tracked objects into higher-level, meaningful events in the context of level crossing. It analyzes the *behavior*, *state changes*, and *interactions* of objects over time, according to rules and states, to understand what is happening. The ultimate goal is to generate standardized event messages (including potential alerts) that describe significant occurrences.

This framework is composed of several modules:

- `EventHandler`:
  - Receives the list of currently live Tracked Objects for each frame.
  - Sorts or groups these objects based on their label.
  - It delegates the detailed analysis of specific object types to specialized handler modules.
  - Acts as an intermediary, collecting results (state transitions, positions, messages) from the specialized handlers.
  - Passes this structured information (e.g., transition type, object ID, timestamp) to the `EventProcessor` for final processing and formatting.
- **Specialized Handlers**:
  - Each handler focuses on a single object category (or closely related categories like vehicles/pedestrians).
  - They maintain the state history for each individual object instance they are responsible for.
  - They implement the core rules for determining an object's current state based on its properties and history:
    * `BarrierHandler`: Calculates the barrier's angle from its bounding box dimensions, compares it to thresholds to determine its states. It also buffers angle history during movement and generates an SVG path of a time-angle curve representation upon completion.
    * `TrafficHandler`: Determines vehicle state based on position relative to a crossing area polygon and movement detection (comparing recent positions to detect if the vehicle is standing still).
    * `TrainHandler`: Infers the train's direction by analyzing the direction of movement based on the history of its horizontal position.
    * `LightHandler`: Detects the appearance and disappearance of individual traffic light signals based on the presence/absence of tracked `'lights on'` objects.
  - When a significant transition or event is triggered, the handler generates a dictionary containing details about the event (object ID, transition type, relevant data like position or angle graph).
- `EventProcessor`:
  - Crucially, it maintains the overall state of the level crossing cycle. This cycle is typically initiated by warning lights and involves barrier closure. The cycle state ends when lights deactivate after barriers are open.

- It uses the current cycle state to interpret the significance of events. For instance, a car event indicating crossing is flagged as an alert only if cycle is currently active (meaning lights/barriers should be preventing crossing).
- It formats the contextualized information into standardized event objects, assigns common metadata, and publishes event to API.

Chapter 4

Testing

Testing was a crucial part of the development process. We emphasized the importance of extensive testing of all components of the system, since ensuring that the dashboard offers a secure, consistent and reliable way to monitor level crossings was among our main priorities.

4.1 Test plan

Since we agreed to follow Test Driven Development approach during the planning phase of the project, we established the testing strategy before starting the development process. Our plan included unit testing each component, followed by integration tests of multiple components, namely integration of frontend with API and computer vision service with API. After successful integration, we planned to conduct system tests, to further verify that all the requirements are met and performance of the system is adequate.

4.1.1 Unit testing

- **Computer vision and post-processing**: Ensure model correctness, accuracy, detection and classification of objects, tracking of objects, accurately assessing their parameters (such as vehicle speed or stage of a barrier cycle), verify the alert triggers.
- **API**: Verify that login, level management, video stream endpoints, data retrieval endpoints work correctly. We plan to use Postman for endpoint testing and xUnit for ASP.NET testing.
- **Database**: Make sure that data is stored and retrieved correctly in PostgreSQL database, test performance for TimescaleDB with a live video stream, integrity of data relationships. We plan to use pgTAP and manual SQL querries to achieve that.

4.1.2 Integration testing

We plan to stick to the idea of continuous integration and write integration tests as soon as we start to integrate the components, with basic data pipelines tests in **week 10** and more advanced tests starting **week 12**.

- **Frontend and API**: Make sure that frontend displays data received from API correctly, test that frontend makes correct requests to the API, ensure functionality of live-stream and video playback.

- **Computer vision and API**: Verify that data is sent correctly to the API and that data is stored and processed correctly. We will use postman for API testing and mock data.

4.1.3 System testing

We planned to carry out system-wide tests during **weeks 13-14**.

- **Functional testing**: Verify that all the user requirements are met and that the system functions correctly under different scenarios.
- **Performance testing**: Check system response time, stress test the system under a high load, ensure that resource use is reasonable.

4.2 Test results

In this section we will discuss testing frameworks used for every type of test and component, as well as testing results and insights.

4.2.1 Unit tests

API and Database

As described in Section 3.3, we used ASP.NET Core and EF Core to build the API, as well as create and interact with the database. This created a possibility to test both API and the database using NUnit, a C# unit testing library. In order to achieve modularity and separation of concerns, we detached the controllers from the underlying services that directly interact with EF Core by using mock objects that only mimic the behavior of said services. We conducted unit tests for every controller, thus only validating error handling, response codes and objects returned by every endpoint. Our final test coverage for all controllers was 94% with 100% pass rate. Then, separately, we tested the services that contain the definitions of LINQ queries and procedures with 87% coverage and 100% pass rate, which ensured correct functionality of the database.

4.2.2 Integration tests

API and frontend

To simulate real-world user flows, catch UI specific issues and ensure that implemented security measures are functional, we decided to test API-frontend integration manually. We monitored and evaluated the logs to test whether the frontend is able to access the implemented endpoints and handle errors. Furthermore, we tested if the frontend UI acts on the API responses accordingly - whether it gets updated in case of a success response and displays comprehensible error messages. Additionally, we validated that neither API nor frontend are accessible by unauthenticated users and if admin-only UI elements can be accessed exclusively by admin users.

API and computer vision service

We used Postman to test API and computer vision integration. In our system, there are two scenarios in which API interacts with computer vision service and vice versa.

Event creation Once computer vision service detects an event, it must call a POST endpoint in API, providing valid event data in the request body. To test this scenario, we used a short video stream of a level crossing with all the events already manually extracted. The video stream was then processed in computer vision service. After the entire video stream was processed, we then used logging and Postman to verify whether all events were extracted and sent to API by computer vision service, if the event data was correct and if any errors occurred during execution. This process was then repeated for different test video streams.

Video stream processing After the user publishes a new camera with the WebRTC link to the video stream, the API must send a message to the Kafka broker, containing camera and crossing IDs, as well as WebRTC link. To test this case, using Postman we accessed API endpoints to create mock crossing containing mock camera, after which we analyzed computer vision service debug logs to validate if the video stream processing for the correct crossing and camera was initiated. We tested various scenarios, including adding multiple cameras at once and restarting the system to ensure that already added cameras continue being processed.

4.2.3 System tests

As outlined in our test plan, system-wide testing was conducted during weeks 13 and 14. The primary objective was to validate the end-to-end functionality of the integrated system, ensuring all components worked together seamlessly and met the specified functional and performance requirements.

We performed manual tests to verify user authentication and role-based access control (System Test 1). These tests confirmed that users could log in and out securely via the frontend interface.

We also tested the crossing set-up end-to-end (System Test 2). Following the steps from selecting "Add Crossing" to saving the final configuration, we verified that the system successfully displayed the camera preview upon entering a valid WebRTC endpoint, and correctly accepted exactly four points during the polygon calibration phase.

After that, we tested the event log functionality (System Test 3). After allowing the system to monitor a crossing and generate events, we tested the filtering mechanisms on the dashboard. Filtering events by a specific time window (e.g., 1 day) and by event type worked as expected, displaying only the relevant events.

Finally, to rigorously assess the core event detection performance of the computer vision service within the integrated system, we used a standardized test procedure System Test 4. This involved configuring a crossing to use a specific, predefined test video streamed locally via WebRTC. This reference video contained known events at precise timestamps. After processing the entire 5-minute footage, we inspected the generated event log and timeline. The system successfully detected all five key reference events (lights on, barrier closure, train passing, barrier opening, lights off).

Overall, the system tests demonstrated that the integrated application meets the core functional requirements.

4.3 ML model evaluation

We trained the model for 300 epochs. The training process was generally successful, showing significant improvement from the initial epochs. Key performance metrics like `mAP` increased substantially, and losses decreased accordingly. However, we noticed signs of potential overfitting in the later stages.

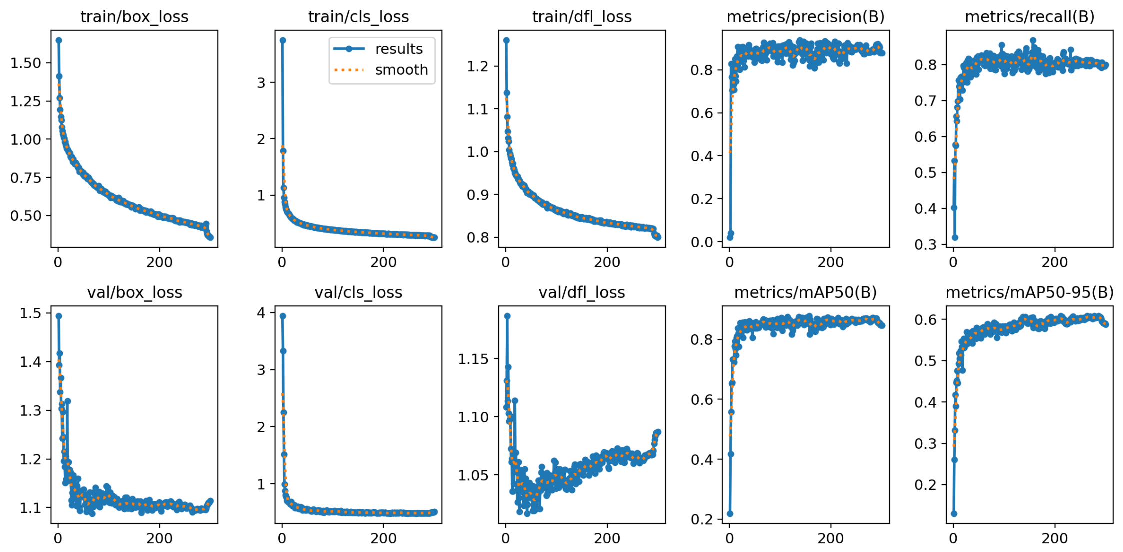


Figure 4.1: YOLOv12 training metrics

Figure 4.1 graphs the following training metrics:

- **Training Loss**: All training loss components show a clear downward trend throughout the 300 epochs. They decrease rapidly in the early epochs and continue to decrease, albeit more slowly, towards the end. This indicates the model was continuously learning from the training data.
- **Validation Loss**: Validation losses also decrease significantly from the start, mirroring the training loss initially. However, they seem to plateau or even slightly increase in the last $\approx 30-40$ epochs (roughly from epoch 260-270 onwards), without showing further significant improvement.
- **Precision & Recall**: Both precision and metrics show substantial improvement from very low starting values. Precision reaches high values (often >0.88), while Recall seems to plateau around 0.80-0.82 in the later epochs.
- **mAP50**: Average Precision at IoU threshold 50 also increases, peaking around 0.87.
- **mAP50-95**: `mAP50-95` shows strong growth, peaking around 0.608 also around epoch 260-270. It seems to stagnate after reaching this peak.

The divergence in training loss and validation loss, along with `mAP50` and `mAP50-95` starting to stagnate, is likely a sign of overfitting in the last $\approx$ 30 epochs, so there was no need to train the model for longer.

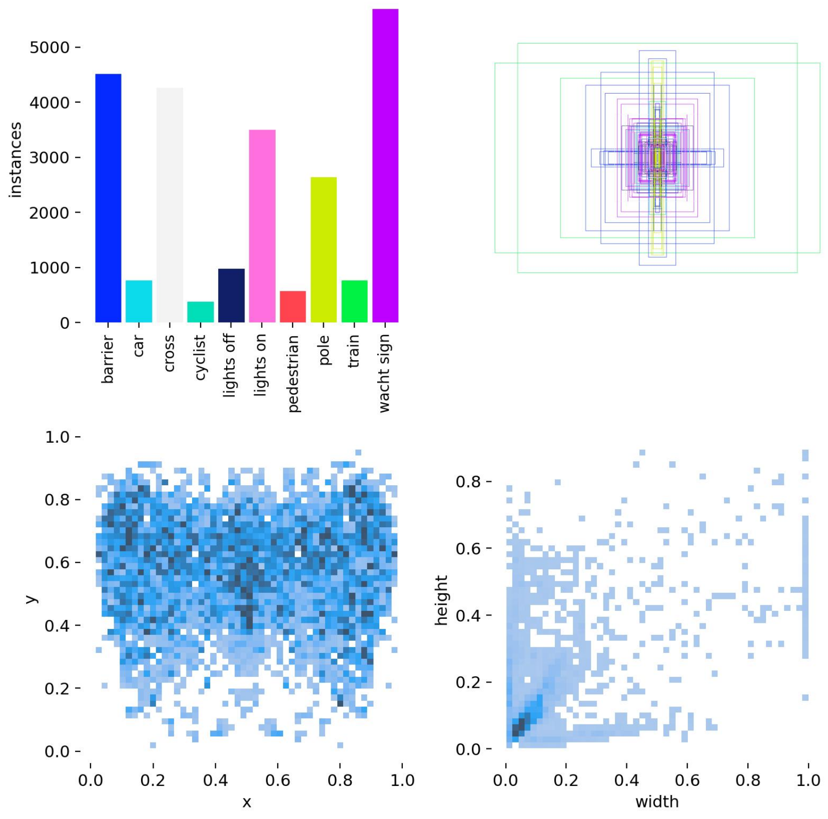


FIGURE 4.2: Label metrics

In Figure 4.2, we can notice that the dataset is highly imbalanced, with some classes such as pedestrian and cyclist being underrepresented. We attempted to augment our dataset by injecting more samples of these classes from the COCO dataset [3], but the results were not adequate. Object-size distribution also might give another insight to why our model is under performing on detecting pedestrians: on the heatmap there is a strong concentration of small objects, and detecting them properly, given camera quality constraints, is definitely a challenge.

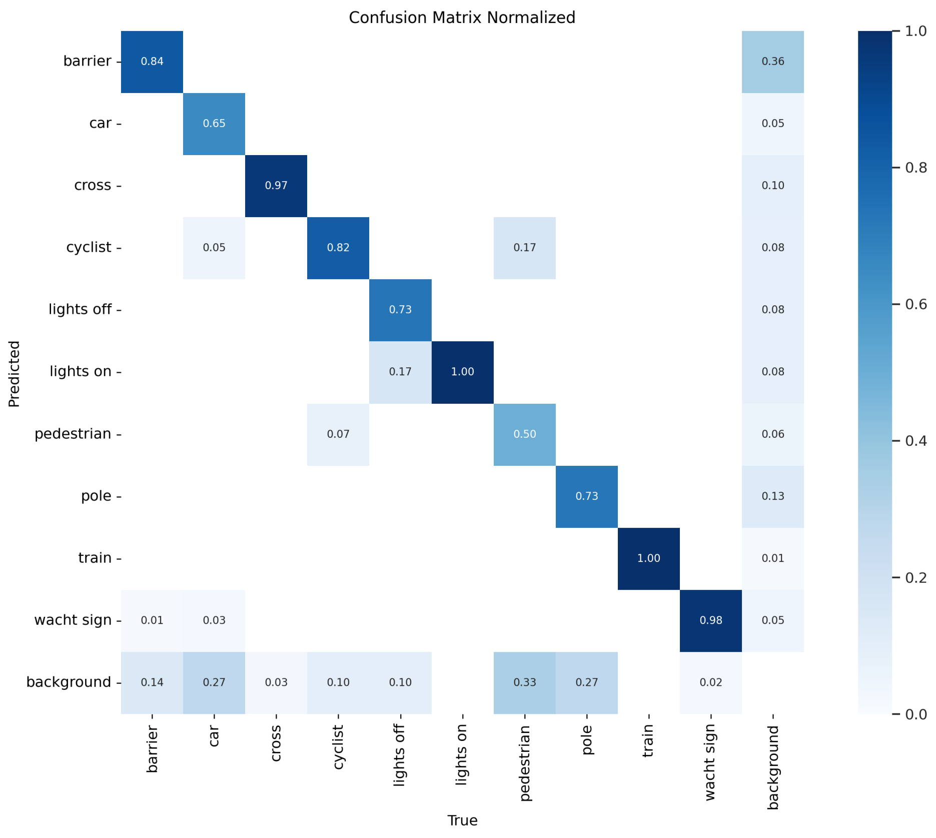
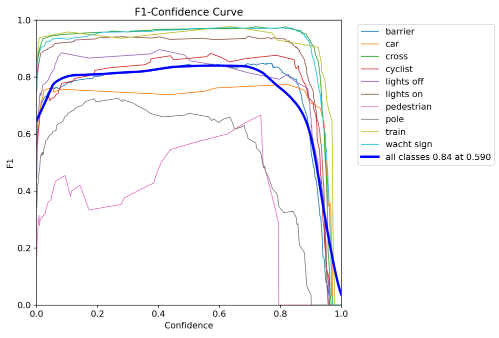
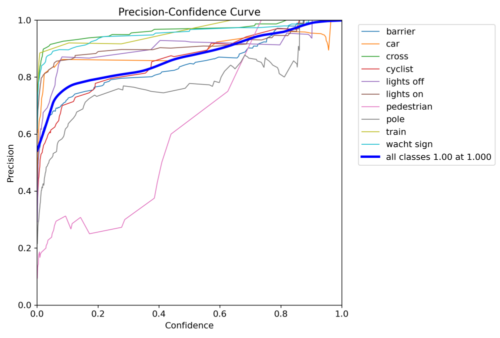


FIGURE 4.3: Normalized confusion matrix for object detection

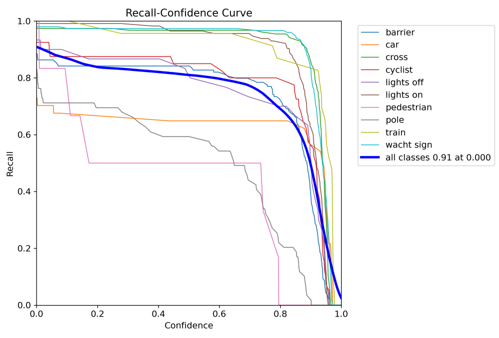
As we can see in Figure 4.3, the model is mostly correct. Barriers, lights, crosses, trains and signs are the strongest classes with high recall, which we consider a success, as those are the most important for our use case. There is some confusion of 'lights off' with 'lights on' and background, so we decided to base our logic on presence of 'lights on' and absence of 'lights off' instead. There is a noticeable overlap between 'cyclist' and 'pedestrian', which can be explained by the way we labeled these classes: we considered a person on a bike to be one object 'cyclist'. We made an attempt to relabel it into two objects ('person' and 'bycicle') and augment the dataset with more samples of these classes, but it was not fruitful. We also attempted to add more background images (with no classes present), but there was a serious flaw in this approach: background images were sourced from google maps, so there were elements of the Google Maps UI present, which probably introduced a shortcut that led to decrease in model performance.



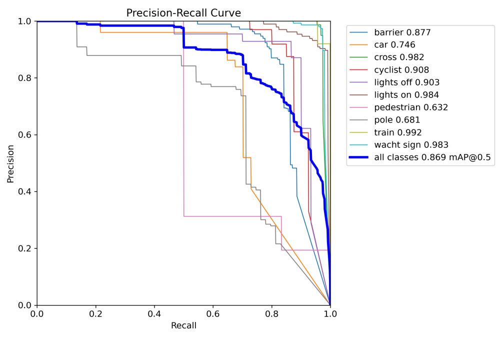
(A) F1 Curve



(B) Precision Curve



(C) Recall Curve



(D) PR Curve

FIGURE 4.4: Performance curves showing (a) F1 score, (b) Precision, (c) Recall, and (d) Precision-Recall relationships.

As in Figure 4.4, `mAP` is very good, even despite the fact that it is significatly pulled down by 'car', 'pedestrian' and 'pole'. Precision increases with confidence, recall drops as confidence threshold is increased. The F1 curve suggests confidence threshold for deployment of 0.59. We went with 0.7, which led to more objects such as pedestrians and cyclist not being detected, but gave us more stability in the realm of level crossing hardware.

Chapter 5

Future improvements

The current implementation of the level crossing monitoring system meets the key functionalities requested from our stakeholder, such as live stream and object detection from the level crossings together with statistics calculation based on the observed events. However, as the project development on our side is coming to an end, the Strukton company will be responsible for future production of the system. In this section, we want to draw attention to what changes and improvements are possible in the future for the developed level-crossing monitoring system.

5.1 Deployment

During the development of the project our team set up a Docker Compose environment that allowed to run and test all services: frontend, API, computer vision, TimescaleDB, Kafka, and the frame producer. Strukton will take over the management and hosting of the system after the project is finished, and will have to integrate the system in its own environment.

One of the key functionality of the system is based on live video streams from actual cameras placed at level crossings. At the moment, the system offers functionality for entering a generic WebRTC stream URL when creating a new crossing via the UI. This address is then used to establish a connection and receive frames for analysis. Currently, a pre-recorded test video is used as input to run and test the application. Strukton will be responsible for setting up the cameras at the crossings and making sure that it is installed and positioned correctly, and that it can stream video in order for the system to operate properly.

5.2 Maintenance

Once the Level Crossing Monitoring system is set up, in order to keep the system reliable and useful overtime, we believe that the following aspects will need attention in the future.

One of the things to consider is the computer vision model. We are using a object detection system based on a YOLOv12 model, which we adjusted with labeled images. The dataset that we used in our project was gathered from different sources, such as publicly available videos, and already existing datasets. We also made our own video from the same perspective that will be used by Strukton. This was important because the videos that we used for training have different views or camera angles compared to what is planned to use. Still, we encountered some problems during testing, as some classes were underrepresented, such as pedestrians and bikes, so we needed to integrate existing datasets for better object recognition. In total we acquired 872 labeled images. To maintain good accuracy, it is important to keep expanding the dataset, especially of underrepresented objects. Also, we tried to include the unusual weather conditions or behavior on level crossings, but we could not find much videos with those unexpected scenarios or the quality of the video was too poor for object detection (e.g. night time, fog). Future improvements could include collecting more videos from actual crossings to adapt to new places and situations.

In addition, the TimescaleDB now stores lots of time-series data about crossings. The events happen often, so the database grows quickly, especially in busy areas. So, it might require rules for keeping or deleting the data.

5.3 Extendability

One of the areas for potential improvement is the addition of different sensor types. Currently the system relies only on the visual input. As was mentioned by the stakeholder, the system could have the sensors for noise level detection, together with the statistics related to it, e.g. measuring the loudness of trains and traffic, check the functionality of bells.

The web application offers smooth functionality for level crossing monitoring, however in the future the number of registered crossings in the system will grow. The user interface could also be improved by introducing various filters for finding specific level crossings.

Chapter 6

Evaluation

6.1 Organization & Planning

From the beginning, we had clear task distribution and consistent team coordination. We used Trello as our primary tool for task management. For each week, we created a dedicated board where we add cards for features, bug fixes, and research tasks, assigning them to the appropriate team members. This setup allowed everyone to track overall progress and clearly understand who was responsible for what.

We held daily internal meetings to discuss what we had worked on the previous day, collaborate when needed and plan upcoming tasks. In addition, we met with our supervisor and client every Monday to present our progress and collect feedback or new requirements. These meetings were critical in ensuring our priorities are set straight and we meet the clients' requirements. To keep a record of our discussions, we took minutes during each meeting. These minutes served as a valuable reference point throughout the project (see Section A). For instance, during Meeting 3, the client suggested to add the option to select custom time ranges for filtering in the interface instead of the previously developed solution which was set ranges (i.e. Last Day, Last Week and Last Month). This led to a UI update the following week and improved the system's usability based on the clients' suggestion. This has been evident in all meetings we had and they showed great importance.

For development, we followed a structured workflow for contributing to the codebase. No one was allowed to commit directly to the `main` branch. Each team member developed on their own feature branch and created a pull request (PR) when their work was ready for review. Every PR had to be reviewed by at least one other member before being merged. This collaborative development approach helped reduce bugs, enforced code quality and maintained a clean and stable codebase.

To facilitate smooth communication, we set up a dedicated Discord server with multiple channels for different aspects of the project, such as frontend, backend, computer vision and general coordination. This made communication more efficient.

6.2 Contributions

During the project, we tried to maintain an equal work load between team members to ensure fairness, this can be seen in Table 6.1:

Task	Amina	Arturs	Andrei	Danil	Moamen	Ksenija
Report writing	×	×	×	×	×	×
Front-end Implementation			×			
API Design	×	×	×		×	×
Database design		×	×	×	×	
Backend Implementation	×	×			×	×
Kafka Implementation		×	×	×	×	
Docker Deployments			×	×		
Labeling Dataset	×				×	×
Training Model	×			×		×
Computer vision microservice		×	×	×		
Testing	×					×

Table 6.1: Team Member Contributions Table

6.3 Results

The project aimed to create a comprehensive system to enhance the safety and operational efficiency of railway level crossings, specifically for our client, Strukton Systems, who is responsible for maintaining rail infrastructure. The core objective was to provide a centralized application capable of processing video feeds from webcams installed at crossings, offering real-time monitoring, event detection, and historical analysis capabilities. The system was designed with railway personnel in mind, ensuring a user-friendly interface for observing crossing events, tracking hardware health (barriers, lights), verifying the correctness of operational cycles, gathering traffic statistics, and detecting anomalies such as barrier malfunctions or unauthorized vehicle presence. Throughout the development process, we maintained a strong focus on addressing the specific needs articulated by the client, incorporating feedback to shape the system's features and ensure usability.

The final system successfully integrates several key components as outlined in our architecture: a computer vision service for analyzing video streams, a robust API connecting the different parts, a database for storing event data and configurations, and an interactive web-based user interface. The computer vision pipeline, effectively identifies and tracks relevant objects like barriers, vehicles, trains, and lights. A rule-based event handling framework processes this tracking data to determine barrier states, traffic movements, train passages, and light activation states, ultimately managing the overall crossing cycle status and triggering alerts for anomalies. The system architecture, featuring a Kafka pipeline for managing video frames and events, ensures scalability and decouples the components effectively. The API provides a standardized interface for the frontend and potentially other external systems. The web UI allows users to configure new crossings, including camera setup and calibration, monitor live feeds, review historical events on an interactive timeline, and access recorded video playback associated with specific events.

Despite achieving the core objectives, the project encountered several challenges. Initial hurdles included delays in client communication and a significant bottleneck in acquiring and annotating sufficient image data for training the object detection model, which required considerable effort in data gathering, augmentation, and incorporating external datasets like COCO. Within the computer vision domain, the on-standardized configurations of different level crossings presented difficulties. Furthermore, rigorously testing the event detection logic was problematic due to a lack of diverse real-world footage covering all possible scenarios and edge cases. Defining a reliable confidence metric for the detected events also proved challenging. Due to time constraints, certain features, such as vehicle speed detection and monitoring the loudness of warning bells, were not implemented, though all features did not rank high on the MoSCoW scale. Reflecting on the process, while team collaboration, overall planning, system design, and communication were strong points, aspects like testing and integration practices could have been more rigorous from the start, and earlier field observation or better planning for the data labeling workload might have mitigated some hurdles.

Nevertheless, the project successfully delivered a functional and robust level crossing monitoring system. It meets all of the main requirements proposed by the client, providing reliable detection of key events, verification of crossing cycles, and an intuitive interface for monitoring and analysis.

Appendix A

Client meetings

Meeting 1 – 14.02.2025

Focus: Meeting client, initial requirements, previous documentation

- First time introductions with the client.
- Client showed a version of the project from 2006, discussed its limitations.
- Discussed a list of requirements and their urgency.
- Client offered some test footage and examples from the 2006 rendition of the project.

Meeting 2 – 10.03.2025

Focus: Validation of project proposal

- Presented the project proposal to the client.
- Proposal is fully green-lit, with no modifications.
- Established an overall timeline with the client for the entirety of the project.
- Established a weekly meeting schedule with the client.

Meeting 3 – 17.03.2025

Focus: Frontend feedback, computer vision progress, data handling

- Presented the nearly complete frontend and requested client feedback.
- Client asked for the ability to select a custom time range (from one date to another).
- Confirmed that the login system and event grouping (from the previous meeting) were implemented.
- Demonstrated initial computer vision results on a small annotated dataset.
- Discussed car detection behind a train; client accepted camera calibration as an optional feature, though not a current priority.
- Clarified that object detection is fully automatic and users do not need to manually select each object.

- Discussed difficulty detecting objects in low light conditions and requested a camera attachment mentioned by the client for capturing more data.
- Confirmed that partial obstruction (e.g., trees) will not negatively impact detection performance.

Meeting 4 – 24.03.2025

Focus: Integration update, frontend improvements

- Focused on component integration; demonstrated frontend updates:
  - Setting bounding boxes
  - Creating new level crossings
  - Displaying real-time event and statistical data
- Planned to finalize web interface integration within two days.
- Discussed the need for a camera mounting frame; the client could not provide it yet but would deliver it later.

Meeting 5 – 31.03.2025

Focus: Event display, stream synchronization, equipment delivery

- Demonstrated that events are displayed correctly, although the stream has a slight delay.
- Identified remaining tasks:
  - Fix stream delay
  - Improve event recognition and refresh rate
- Confirmed that statistics are already being calculated.
- Received the camera holder from the client, enabling further data collection.

Meeting 6 – 07.04.2025

Focus: New data recordings, feature refinement

- Recorded new video data with the client's equipment, gaining valuable insights.
- Identified visibility issues with light signals due to camera placement and made necessary adjustments.
- Observed different types of crossing, including very small ones; confirmed camera positioning on the bell.
- The discussion involved potential improvements involving reflections (details unclear).
- The client proposed the system to detect stationary objects or people on the tracks even when no train is approaching.

Meeting 7 – 14.04.2025

Focus: Final approval and deployment planning

- Presented the finalized version of the project to the client.
- The client reviewed the system and officially approved its current state.
- We discussed future integration of the product within the Strukton Systems development team.
- We agreed to provide complete documentation and a deployment manual to support further implementation.

Appendix B

System tests

Test 1: User Authentication and Access Control

Component	**Details**
Objective	Verify secure login/logout functionality and role-based access control
Steps	1. Navigate to login page 2. Enter valid admin credentials 3. Attempt to access user management page 4. Logout and login with standard user credentials 5. Attempt to access user management page via URL 6. Attempt login with invalid credentials
Expected Results	• Admin sees User Management option • Standard user gets redirected if accessing user management via URL • Invalid credentials show error message • Logout clears session cookies
Pass Criteria	All expected results match observed behavior

Test 2: Crossing Configuration Workflow

Component	Details
Objective	Validate complete crossing setup process
Steps	1. User selects "Add Crossing" 2. Enters valid name/location 3. Adds camera with valid WebRTC endpoint 4. Completes polygon calibration 5. Saves configuration 6. Verifies new crossing in dashboard
Expected Results	• System validates required fields • Camera preview appears after URL entry • Calibration accepts exactly 4 points • New crossing card appears in overview
Pass Criteria	Full setup complete <5 minutes with 100% success rate

Test 3: Historical Event Analysis

Component	**Details**
Objective	Verify temporal accuracy of recorded events
Steps	1. Allow system to identify events for 15 minutes on arbitrary crossing 2. Filter events by 1-day window 3. Filter events by arbitrary event type 4. Remove all filters 5. Select event from list 6. Initiate playback from timeline 7. Measure temporal offset
Expected Results	• Only filtered events are shown after filtering • All events are shown after removing filters • Position accuracy of events on timeline ±10 seconds • Event metadata matches visual evidence
Pass Criteria	100% temporal accuracy across 20 events

Test 4: Test Camera Validation

Component	Details
Objective	Verify event detection using standardized test footage
Steps	1. Set up crossing with predefined "test" camera* 2. Start 5-minute monitoring session 3. Let system process entire footage 4. Inspect event log for analysis
Expected Results	Timeline and event list show the following events: • **0:38** - Lights on • **0:42** - Barrier closure sequence • **1:05** - Train passing • **1:15** - Barrier opening sequence • **1:26** - Lights off
Pass Criteria	• 100% detection of reference events • Max 1 false positive allowed • Timestamp accuracy ±2 seconds

* The "test" camera refers to a program streaming via WebRTC locally, playing a reference video.

Bibliography

[1] DSDM Consortium. *DSDM: Dynamic Systems Development Method.* Tesseract Publishing, 1997. The original publication describing the MoSCoW prioritization technique as part of DSDM methodology.

[2] R. J. N. de Jong. Railroad crossing video monitoring: The development and comparison of algorithms. Masters thesis in applied mathematics, University of Twente, Enschede, The Netherlands, June 2009.

[3] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015. URL: `https://arxiv.org/abs/1405.0312`, `arXiv:1405.0312`.

[4] ProRail. Prorail in cijfers. `https://www.prorail.nl/over-prorail/wat-doet-prorail/prorail-in-cijfers`, 2017.