10

Edge-Optimized Modular Architecture for Real-Time Vehicle Re-Identification

Tomass Zutis[1], Dimitrios Georgiadis[2], Tassos Kanelos[3], Janis Judvaitis[1], Pēteris Račinskis[1], Konstantina Karathanasopoulou[2], George Dimitrakopoulos[2], Janis Arents[1], and Modris Greitans[1]

[1]Institute of Electronics and Computer Science (EDI), Latvia
[2]Harokopio University of Athens, Greece
[3]G.N.T. Information Systems S.A., Greece

Abstract

This work addresses the challenge of real-time vehicle re-identification (ReID) on resource-constrained edge devices. Our research aims to design and deploy a modular, edge-optimized system that integrates object detection and ReID for intelligent transportation applications. The proposed architecture combines a lightweight detection module with a ReID embedding generator, enabling accurate multi-camera vehicle tracking under strict computational limits. We evaluate the system on NVIDIA Jetson platforms, applying quantization, pruning, and TensorRT optimization to achieve low-latency inference while maintaining accuracy. Additionally, we explore scheduling strategies for day and night operations to optimize energy consumption and performance. The impact of 360-degree fisheye camera distortion on detection and ReID accuracy is analyzed. Experimental results demonstrate that the system achieves real-time performance on edge hardware without sacrificing reliability, making it suitable for large-scale deployment in smart city environments. This study highlights the feasibility of deploying deep learning computer vision pipelines on edge AI platforms for robust, privacy-preserving vehicle analytics.

Keywords: edge AI, privacy preservation, image recognition, object detection, object re-identification, 360-degree fisheye camera.

10.1 Introduction and Background

If the same vehicle drives past a person two times, during different parts of the day and in different locations, how often would a person be able to re-identify this vehicle? If such a task is not trivial for the human eye and memory, it is even less likely that it could somehow be a trivial, out-of-the-box computerized solution. Vision-based object re-identification is a problem of growing interest in security, traffic monitoring, and environment monitoring applications. Real-time footage analysis from network-connected cameras that contain imagery of vehicles in transit is a topic that the research community has taken the greatest interest in regarding vision-based re-identification problems [1, 2].

In [3] T.Zutis et al., who are also co-authors of this article, presented a multi-step object Re-identification pipeline - a prototype (proof-of-concept) with specifications comparable to those needed for deployment on edge devices. This Vehicle Re-Identification pipeline can be a valuable tool for identifying vehicles and tracking them in real-life scenarios without using license plate information and compromising personal information. A fully connected pipeline was presented that consisted of steps to detect objects in video streams, crop them from the corresponding frames, assign IDs, track their position through the field-of-view (FOV), and, once the vehicle had left the FOV, save it into a vector database. The vector database could later be queried for the vehicle from cameras in other locations. During development, this pipeline was split into two parts: vehicle detection and re-identification (or feature extraction, saving, and querying) itself. The first step in the larger pipeline was object detection. For this, the authors used an out-of-the-box YOLO v8 [4] model. For tracking, the *ByteTrack* [5] package was chosen to assign IDs and track them through consecutive frames of the same camera FOV. For the re-identification part, however, a precise and effective feature extraction model had to be trained and tailored. The development of this model and the training results are described in the cited article. The model was tested both with publicly and non-publicly available datasets and network camera video gathered at the Institute of Electronics and Computer Science of Latvia.

The most recent work, however, focused on delivering this pipeline for a true edge computing architecture. The concept and design of the system have undergone many iterations of improvements in speed, implementation, deployment, versatility, and compatibility. True to the original concept of splitting the pipeline into vehicle detection and re-identification during development, the pipeline has been modularised and consists of separate

systems for object detection, feature extraction, and vector database operations. The development of the object detection, feature extraction, and partly also database operator by separate development teams fulfils many other system requirements and makes each component reusable and functional in different contexts. The integration of the object detection system and the feature extraction module is also one of the key development steps that will be described in this article. The modular architecture has required research into integration and data flows. These data flows are enabled by a pub/sub messaging system based on the MQTT stack.

Even though the object detection system is designed also as a standalone obstacle and road conditions perception software, in this article, it fits into a role that seeks to replace the vanilla YOLO v8 solution that was used previously. The system focuses on embedded AI platforms through compression and quantization techniques (ONNX), while using custom inference engines (ONNX Runtime or TensorRT) for inference efficiency and resource minimization. Additionally, it integrates motion-gated frame admission (MOG2, hysteresis), ROIs (Regions of Interest), image tiling and tile scheduling for adaptive load management and low latency. Post-processing after object detection and tracking requires logical reasoning such as zone violation.

Another key aspect of improvement is the deployment of this pipeline on the Nvidia Jetson Orin [6] edge computing device. This proves the applicability of the system in real-world circumstances and enables measuring the performance and computing requirements of our systems. The models and accompanying software have been optimized for the Nvidia Jetson Orin.

An additional direction of experimentation in this work, though still in its earliest stages, has been the use of fisheye cameras for object detection and vehicle re-identification. Such cameras offer a drastically wider field of view compared to conventional surveillance optics, potentially allowing a single unit to cover larger intersections or parking areas. However, the severe geometric distortion introduced by fisheye lenses poses challenges for detection, tracking, and feature extraction, as the vehicle appearance may be warped differently depending on its position in the FOV. Early investigations have been exploratory rather than conclusive, and the research performed so far aims to offer practical engeneering solutions to analysis of distorted vehicle input.

In the following chapters, we aim to describe this proof-of-concept solution on edge devices, the evaluation that has been done to substantiate our claims of success and the results that the authors find consequential for the advancement of these technologies.

In the last chapter - Discussion and Future Work we present the envisioned architecture. Its main aim is to describe the departure from the proof-of-concept version that functions on a single Jetson device to one including a meta edge server and removal of the vector database from the deep-edge. This chapter can be read in paralel to the rest of the article as it gives more context for why the edge optimisation is ultimately necessary.

10.2 Approach

10.2.1 Overall modular architecture

10.2.1.1 Object detection system

The primary models for the object detection system are the YOLOv8's line models provided by Ultralytics which are exported to ONNX (FP16 is preferred) and executed in pure inference engines.

The system is initiated through frame acquisition determined by hardware limitations, influencing the quality and quantity of frames, that pass through the pipeline. It utilizes the FFmpeg video decoder that can feed raw BGR frames into the pipeline.

Next, a motion-gating (frame-level prefilter) technique is incorporated with a background subtractor (MOG2-downscaled) to generate the binary mask. This represents the relative area per frame and performs a morphological cleanup, followed by a simplistic hysteresis approach to avoid brief dropouts.

Initially, the frames are cropped via predefined regions, capturing the keypoints inside the image topology, maximizing the effective resolution for potential activity and simultaneously reducing the tiling load balancing, with fewer necessary tiles per image. Each tile is formatted to accommodate the inference model's inputs, with image tiling accounting for overlapping, thus correctly covering detection in a tile's edges. When ready, the tiles are fed one-by-one to generators and are then batched for concurrent inference.

Frames are also parsed through generators to be called only when necessary, reducing memory overload. This micro-batching reports the inference results from the GPU, which is where the ONNX Runtime with CUDA providers or a pure TensorRT engine are employed. During inference, inputs and outputs are bound to CUDA buffers and in both cases, the model is "warmed-up" for each batch shape, stabilizing latency and building of the engines. Prebinding outputs results in reusing memory that has already been allocated. Once the results have been obtained, an initial mutliclass NMS is

performed on each tile and while this may lower operational performance, it reduces the size of the batched tensor detections. Then, all detections are rescaled into the original input's format and are captured on top of their corresponding frame. The inference results (pre-NMS) are a single tensor (raw inference) and after NMS these become end-to-end four outputs including the bounding boxes, scores and class confidence scores for every detection.

Inference results are decomposed into meaningful information with post-processing mapping tile boxes onto the original frames. These are enhanced and validated using a global Non-Max Suppression (NMS), mixing batched-NMs and single-class NMS. Between engine inference and post-processing, minimal CPU traffic is generated with small tensors or metadata. Tracking is also enabled via the *ByteTracker* model to identify vehicles through frames, which is integrated in a custom way to enable seamless cooperation with the compressed detection model. Lastly, the system crops the finalized outputs, keeping only the detected objects, to maximize transmission efficiency between systems.

10.2.1.2 Re-identification system

The main component of the re-identification system is the re-identification model. The architecture of this model is applicable to re-identification of all types of objects; however, it has been specifically trained to recognize and extract features of vehicles. The model, its training, and the results have been described in the previously mentioned paper [3]. Here, we describe the model regarding its deployment on the edge, the model's and the surrounding software's optimization.

The feature extraction model (originally built in *Pytorch*) was first converted to run on ONNX runtime inference engine. This has also been the first step in optimizing the model, slightly decreasing its size. This way, we also standardized the model for deployment across different runtime environments (e.g., Jetson, CPU/GPU, cloud). Then, considering the goal of edge deployment on a Jetson device, a further conversion of the model to a TensorRT engine was performed. TensorRT [7] is an Nvidia GPU native engine format that further reduces latency and memory footprint. In fact, when converting the model to a TensoRT engine, some optimizations are performed automatically, like layer fusion and kernel auto-tuning.

Beyond model-level optimizations, consecutive load management strategies have been built into the algorithms surrounding the re-identification logic and feature extraction model. This reduces redundant computation across the detection, feature extraction, and database layers.

1. At the feature extraction stage, a zone-based filtering mechanism is applied: FOV's are divided into configurable sub-regions and embeddings are only generated when a tracked object enters a new zone. This prevents unnecessary feature extraction for vehicles that remain static or confined to a single region of interest. This also lets us focus on saving the features of vehicles in different poses/locations, improving the ReID precision.
2. On the detection side, a binary ROI mask is used to restrict the effective input area of the model, allowing the system to ignore irrelevant regions such as sidewalks or sky and thereby reduce the number of unnecessary detections.
3. Finally, database operations employ lightweight caching and periodic cleanup: repeated queries for the same vehicle ID within short timescales are resolved locally, while expired entries are automatically purged to maintain storage efficiency and improve ReID precision.

Together, these mechanisms balance resource usage across the pipeline and support real-time performance under limited computational resources.

10.2.1.3 System integration, data flows etc.

This proof-of-concept indicates that the system integration and data flow are, in most cases, beyond the state of the art (SoTA) and not readily found in the available literature, since most research concerning this matter is still very cloud-centric and relies on simple benchmark testing to signal ReID capability.

The first step in deploying the system onto an edge device was preparing the code for NVIDIA Jetson ORIN. The chosen strategy for this was the separation of the system into 3 different modules with 2 additional external services needed to run the pipeline. The 3 different modules of the pipeline: the object detection system, the feature extraction/Re-Identification module and the database operations module were developed by different entities; hence they had to be run in different environments and with different requirements.

The chosen path that has worked best for the implementation of these different modules on the NVIDIA Jetson has been containerization. Each module was transferred to a Docker container by crafting a Docker image that is suited both for the software requirements and the Jetson architecture, including the limited and specific software distributions available. In this prototype version both the MQTT broker and the Opensearch database

have also been deployed as docker containers in the same network as the 3 modules.

The Dockerization process itself was non-trivial, as the Jetson architecture (based on *JetPack* L4T) only supports specific CUDA, cuDNN, and TensorRT versions. This required building the containers on top of NVIDIA-provided base images (e.g., nvcr.io/nvidia/tensorrt:23.10-py3 for the feature extraction model) to ensure compatibility with the hardware. Furthermore, since the detection, feature extraction, and database operation modules were developed independently with heterogeneous dependencies, each had to be isolated in its own container to avoid version conflicts. GPU access also needed to be explicitly configured using the NVIDIA container runtime, as mismatches between *JetPack* libraries and container environments often led to runtime errors. Finally, resource limitations of the Jetson platform necessitated careful management, particularly because the database and inference engines competed for CPU and memory.

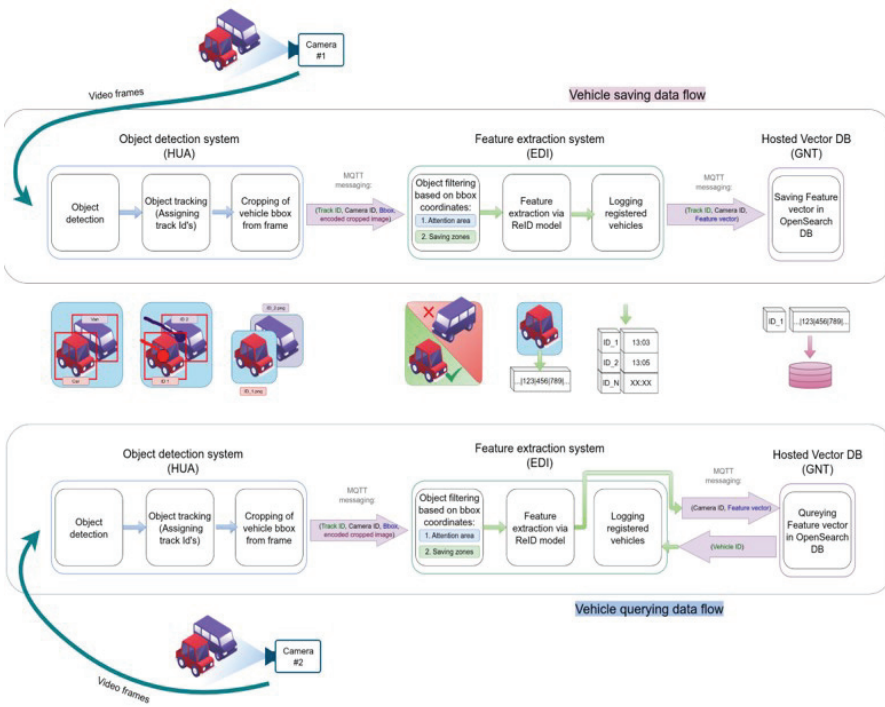

Figure 10.1 Depiction of the logical dataflow in the modularised re-identification pipeline.

After the separation of the pipeline into different modules, a data flow solution was devised (see Figure 10.1). The communication between containers is done via MQTT. The idea behind the data flow is as follows:

1. The object detection system detects different object classes on the roads and intersections. These object classes can be various, including People, Vehicles, and different obstacles. Some of these detections will be valuable for other systems unrelated to ours.
2. For this pipeline, the object detection system will then proceed to select the vehicle detections exclusively. These detections will be forwarded to the feature extraction module. Forwarding a detection means sending an MQTT message consisting of tracking ID, camera ID, bounding box (bbox) coordinates, and *Numpy* image array of the cropped vehicle. MQTT messages between the different modules are presented for comparison in Figure 10.2.
3. Upon reception into the feature extraction module, the vehicles will be filtered by the zone-based mechanism. The mechanism will take the bbox coordinates from the detection outputs into consideration to establish in which zone the current detection is located and if it has changed its zone from the previous time it was detected. If the zone has indeed been changed for the detected object with the same track ID, then it will proceed forward to the feature extraction model. If the vehicle with the same track ID is detected in the same zone as before, it will be discarded. More on this logic can be found in the previously published article.
4. In the next step, the cropped image of the detection is passed through the feature extraction model and converted into a feature vector.
5. Subsequently, the feature extraction model will construct an MQTT message that will be sent upstream to the database operations module. Even though this will be heavily changed upon departure from the proof-of-concept version. The message will consist of the track ID, camera ID and the feature vector.
6. Upon receiving the message, the database operations module first checks whether this track from the same camera has already been cached locally. If so, it avoids redundant queries and directly updates the database with the new vector.
7. If the track is new, the module queries the vector database (in this case, *OpenSearch*, configured with vector fields and k-NN search). The query uses cosine similarity to compare the incoming feature vector against stored vehicle embeddings.

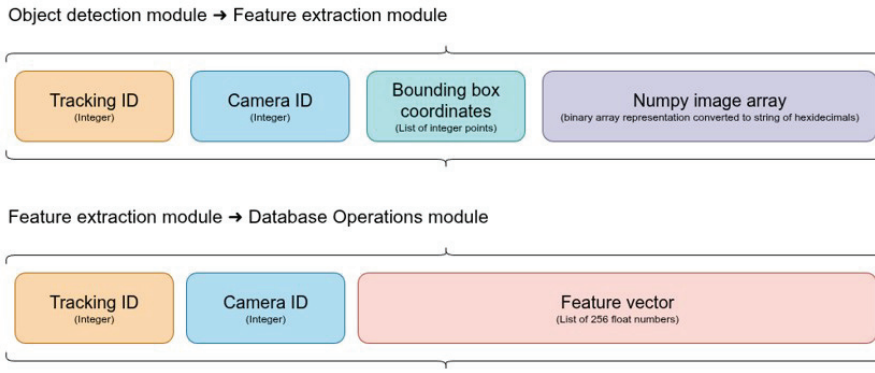

Figure 10.2 MQTT messages being sent from one module to another.

8. Depending on the result of the similarity search:
    a. If a sufficiently similar vehicle is found, the feature vector is added to that vehicle's existing entry (reinforcing the embedding).
    b. If no match is found, a new ID (composed of the camera ID + track ID) is generated and inserted into the database.
9. The database operations container also runs periodic cleanup. A time-based deletion process removes vectors older than a configured threshold (e.g., 60 seconds in one test setup)

The networking between modules has been another challenge: the prototype communication via MQTT required configuring a shared *Docker* network that also included the *Opensearch* database and broker.

10.2.1.4 Switching to Fisheye camera usage

To expand the real-world applicability of the pipeline, the design has shifted to considering the possible inclusion of fisheye cameras for vehicle detection. The motivation for this shift stems from the growing deployment of wide-angle surveillance cameras in intersections and parking environments, where a single fisheye lens can cover multiple lanes or regions of interest.

10.2.2 Fisheye camera usage in object detection

The initial step in preprocessing fisheye camera frames is to calibrate the camera lens. Calibration acquires camera intrinsics (K) and distortion coefficient (D) for the specific camera utilized. Calibration is essential for determining a metric mapping between image pixels and the real-world scene. For

preliminary experimentation, the object detection pipeline was fed the FishEye8K [8] dataset, which has been evaluated with YOLOv8. However, since it does not ship per-camera calibration, capturing the distortion intrinsics for true metric calibration is challenging and often ill-posed.

Existing literature proposes self-calibration or surrogate calibration [9] to fit parameters using scene constraints. Typically, these methods initialize a principal point near the image centre to perform grid-search on a global FOV to estimate distortion. Refinements are necessary to enforce straight edges to lane markings, poles, and building edges. Plumb-line optimization techniques exploit long line segments across small frame set to optimize K, D without requiring a calibration target. For each candidate FOV, the process involves de-warping a few tangent views and validating line straightness, with the FOV that minimizes curvature being selected. Although these approaches exist and can work without a "checkerboard", their accuracy depends on how much geometric signal is contained within the dataset, leading to noisier results than target-based calibration.

A computationally efficient and calibration-free alternative is distorting the initial frames to match the model's rectilinear profile. Conceptually, this constitutes a domain shift problem, from fisheye imagery to rectilinear. Precise K, D aren't required as we can approximate undistortion, leveraging barrel correction to straighten edges near borders while mild center cropping is used to remove the most warped peripheral regions. In fisheye imagery, the strongest distortion regions occur near the periphery, whereas the central region closely approximates a pinhole projection. Accordingly, a "de-fish" warp functionality compresses outer regions and expands inner ones, remapping pixels so that radial lines appear straighter. When the projection type is unknown, radial or polynomial distortion models can serve as estimation tools. Radial undistortion applies the inverse of such models, compensating for radial deviation by introducing a mapping that restores line straightness. This method assumes a generic k that flattens curvature enough to determine a pinhole image.

While de-fishing inherently stretches central pixels and compresses the edges which will result in resolution loss near the periphery and undefined regions at the corners it remains a real-time deployment mechanism with minimal computational overhead. It provides a practical replacement when neither calibration nor retraining is feasible, acting as a fast heuristic to recover baseline detection accuracy. For future extensions of the object detection pipeline, a geometry-based calibration procedure will be incorporated to obtain accurate K, D and achieve true metric undistortion.

10.2.3 Fisheye camera usage in feature extraction

For vehicle feature extraction, experiments were conducted to evaluate the performance of the feature extraction model under fisheye camera conditions. To simulate this scenario, the existing AICity [10] test dataset was modified by applying synthetic fisheye distortion to each frame, mimicking the radial deformation characteristics of such optics. The modified dataset also adjusted the ground truth annotations, enabling a direct comparison of model precision between standard and distorted inputs.

While no dedicated fisheye-specific training or correction layers were applied at this stage, running the pipeline on this experimental footage establishes a baseline for future work with geometric adaptation and fisheye lens effects.

10.3 Evaluation

10.3.1 Object detection system

The object detection system must not be evaluated only on the detection performance of the well-established YOLO models but should address the accuracy and effectiveness holistically. The KPIs recommended in this subsection align directly to the structure of the pipeline. On the model's side, the fundamental mAP/Precision/Recall and F1 are computed on the finalized, full-frame outputs after the post-processing mechanism is completed, hence right after the global-NMS. This provides a clear measurement of detection quality, when utilizing ONNX or TensorRT backends. Such metrics alongside Frames-per-Second are standard in literature as per [11], where a streaming evaluation protocol is introduced to assess latency-aware detection and report the FPS vs accuracy comparison. Similarily, approaches can be found in [12] and [13], which have a more focused end-to-end benchmarking scheme, integrating NMS performance and on edge devices such as Jetson-enabled GPUs. For this work's pipeline NMS is incorporated for evaluation, specifically duplicate rate across tiles which are removed by NMS (suppression ratio). Additionally, misses near tile boundaries directly validate per-tile NMS and final-frame global NMS strategy with tile overlap. Background motion gating is validated by accounting for frames saved (background-filter lift) and recall under motion gating as per [14], thus ensuring the pipeline skips frames without missing vehicles. Contextually aligned, tracking can be validated through the traditional MOTA (Multi-Object Tracking Accuracy) and the modern HOTA (Higher Order Tracking Accuracy) [15] are attained to reflect how well the *ByteTrack* stiches final detection into stable tracks.

Such a pipeline is required to be applicable for resource-constrained platforms. Measurements, mostly involve the throughput (FPS) and end-to-end latency with the p50/p95/p99 percentiles to yield the system's response times and to capture real-tile performance. Also, a pipeline stage breakdown shows time expenditure based on each component and allows for fair comparison between the inference engines. Batch efficiency, indicated as the ratio of tiles inside the buffer per the maximum buffer size and suppression ratio for the global NMS quantify processing concurrency and deduplication. Lastly, GPU utilization and peak RAM memory allocation can be combined with CPU utilization and outlier counts or dropped frames to connect performance back to the hardware limitations and data movements. These are tailored for tuning image tiling, batch sizes and gating parameters to hit the target FPS and latency budget without sacrificing detection and tracking quality.

10.3.2 Feature extraction

The following subsections describe the ways we have evaluated the optimisation of our solution. We looked at the inference speed per image for the feature extraction model optimization and the precision of re-identification on distorted fisheye projection imagery together with a comparison of the embedding similarity in the feature space for normal and distorted images.

10.3.2.1 Inference speed per image

The feature extraction model will be embedded into a Jetson + Docker + TensorRT setup. This setup is what allows us to deploy a real-time solution on the edge. Feature extraction models have been successfully converted to ONNX and further optimized to run with TensorRT.

The evaluation compares the original PyTorch implementation of the re-identification model, the ONNX and TensorRT conversions. First the model was exported to the ONNX format for hardware-agnostic inference combined with several automatic graph-level optimizations (constant folding, operator fusion, and elimination of redundant nodes). Following this step, the ONNX model was compiled using NVIDIA TensorRT. TensorRT applies additional layer and precision optimizations, such as kernel auto-tuning, dynamic tensor memory management, to achieve substantial runtime acceleration on the Jetson platform.

The evaluation of the three model representations: PyTorch, ONNX, and TensorRT, was performed using the same AICity test dataset and batch size, with two metrics.

The first metric is inference time measured as the average inference time per image (in milliseconds). Inference time is crucial to real-time deployments. When processing inputs from a live-stream video of detected vehicles, processing one image should take as little time as possible. If inference time is too long, the feature extraction risks becoming a bottleneck in the pipeline and disrupting real-time re-identification. It is also noteworthy that there is no set limit to how many vehicles the model would have to process in a second, as traffic is irregular, and some moments can see more cars on the road than at others. The second metric is memory consumption, recorded as the average GPU memory usage (in gigabytes) during the inference test. Having low memory consumption for feature extraction is crucial because the computation will be done on an edge device. The feature extraction module is also planned to be the most resource consuming process in the entire pipeline.

For the same feature extraction model, these metrics can vary when deployed on different devices; hence, it is important to test them on the Nvidia Jetson ORIN device. These metrics provide a crucial understanding of the model's performance on a resource-limited computational device.

10.3.2.2 Re-identification on Fisheye projection images

The evaluation of Fisheye distorted image impact on the extracted vehicle features consisted of two main analyses. First, the pipeline's re-identification Rank-1 accuracy [16] was measured on the distorted dataset, showing how the existing model, trained on (mostly but not exclusively) perspective projection camera images, generalizes to fisheye conditions. Second, feature vector cosine similarity [17] was examined in the embedding space, comparing cosine distances between features extracted from the same vehicles before and after distortion. This allowed for a quantitative estimation of how the distortion affects the learned feature representations.

Re-running the tests of the previous article, where the model was first presented, with distorted images is an obvious choice due to the direct comparability between the two types of images. If the Re-identification test results were to fall it could be directly attributed to the image distortion.

Measuring the vector distance in the embedding space is a slightly more in-depth analysis of the distortion effects. If images of the same vehicles were further apart in the embedding space after fisheye distortion was applied, we could directly indicate that the model's learned embedding function is sensitive to geometric deformation, causing vehicles that should be recognized as identical to occupy disjoint regions of the feature space. Conversely, a small

or negligible change in intra-class distances would suggest that the feature extractor preserves its discriminative capability despite the altered image geometry. This embedding-based comparison thus provides a quantitative complement to the precision and recall measurements, offering deeper insight into how fisheye distortion affects the internal feature representations used for vehicle re-identification.

10.3.2.3 System integration

The fully containerized system, consisting of the object detection, feature extraction, and database operations modules, is to be deployed and executed on the NVIDIA Jetson Orin edge device. Successful operation of these modules will confirm the feasibility of running the complete pipeline as each module depends on at least one other.

To evaluate the reliability of the data flow between modules, message transmission metrics were recorded. Specifically, the number of detection messages sent from the object detection module was compared against the number of messages received by the feature extraction module. This measurement served to confirm that the MQTT communication setup was stable and lossless under continuous operation.

A second comparison was made between the number of detections received by the feature extraction module and the number of feature vectors it produced and sent onward to the database operation module. This ratio reflects the effectiveness of the load-reduction mechanisms within the feature extraction module, such as motion gating and zone-based filtering, which intentionally suppress redundant detections to optimize processing throughput.

10.4 Results

10.4.1 Object detection performance

Experimentation of the object detection system was executed through real-time, publicly available camera sources, set on top of highways. For reference the current results have all been gathered for a 1920x1080 resolution video in rectilinear format extracted from these sources, capturing 2 minutes and 3 seconds. For this evaluation ONNX runtime and TensorRT engine were separately used. We consider that the GPU and RAM utilization start from 0 for ease of demonstration purposes.

Table 10.1 Object detection system speed and resource consumption overview of ONNX Runtime and TensorRT

Metric	ONNX Runtime	TensorRT
Total Execution Time (s)	**294.8**	344.1
Peak Memory Usage (MB)	211.07	**210.92**
GPU Utilization (%)	29.2	**7.96**
Avg. Inference time (ms)	**70.52**	122.70
Mean FPS	**5.49**	4.7

Both systems were executed with a batch size of 16 tiles. Preprocessing involved motion-gating, initial ROI cropping and transformation into tensors. Image tiling is performed with a grid of (2,4) tiles. NMS is performed per tile before depicting the finalized detection back to the original image and then a global NMS ensures no duplicates are remaining. Tracking is also enabled.

In Table 10.1 ONNX runtime which uses the CUDA execution provider, appears faster but more memory-intense due to the more brute-force and less-optimized execution pattern, which keeps the GPU busier and less resource efficient. It achieves higher FPS, due to lighter setup overhead and more aggressive kernel dispatching. Hence, ONNX runtime reflects higher kernel activity and shorter inter-kernel gaps despite less optimization. Conversely, TensorRT performs layer fusion and kernel optimization during engine building, leading to longer setup times, but saves execution time during warmup.

Analytically the p50/p95/p99 per operation and mean time are depicted in Table 10.2. Precision handling influences both systems in how they operate.

Table 10.2 Object detection performance comparison in seconds between ONNX Runtime and TensorRT

Metric	TensorRT	ONNX Runtime
Setup Process (mean)	**1.25**	1.49
Setup Model (mean)	5.46	**3.37**
Setup Logic (mean)	**4.7**	6.1
Setup MQTT (mean)	0.203	**0.202**
Warmup Session (mean)	**5.58**	9.51
ROI Cropping (mean)	**0.202**	0.301
Motion-Gating (mean)	**0.85**	0.86
Post Process (mean)	**1.025**	1.094
Frame Time p50	0.201	**0.168**
Frame Time p95	0.310	**0.279**
Frame Time p99	0.332	**0.291**

ONNX Runtime is set up for FP32 (full precision). TensorRT engine is also built for FP32, thus limiting acceleration benefits for smaller precision. while keeping all the overhead required. Additionally, total inference time includes synchronization points, forcing all asynchronous streams to be completed, making the measured time much longer. By default, ONNX runtime hides this by pipelining operations on multiple streams.

It is important to contextualize the overall performance of the pipeline in terms of end-to-end FPS, due to this metric showcasing the cumulative cost of all separate processing stages rather than the raw model inference, explicitly. The reported measurements, thus, reflect the complete system, including the motion-gating mechanism, model warm-up, optional image tiling, ROI handling, and also post-processing steps such as frame cropping, sub-mask generation and downstream communication or in-memory buffering. In its current state, the pipeline underperforms compared to state of the art deployments, which handle regular camera feeds. Isolated from the rest of the system, the detection model achieves an effective latency of approximately 60 ms per batch ($\approx$3.75 ms per frame for a batch size of 16), but inference on high-resolution fisheye imagery increases to roughly 100 ms per batch ($\approx$6.25 ms per frame). By comparison optimized TensorRT deployments of YOLOv8 on embedded AI platforms have reported mean inference latencies of approximately 3.2 ms per frame under comparable conditions. While this highlights a current performance gap, ongoing optimization efforts across preprocessing, memory transfers and post-processing are steadily reducing latency, with successive iterations returning measurable improvements towards state of the art performance.

10.4.2 ReID Feature extraction performance

10.4.2.1 Inference speed per image results

The results are summarized in Figure 10.3, showing that, specifically for the feature extraction perfromed by the ReID model, the TensorRT implementation achieves the lowest latency and the smallest memory footprint among the three frameworks, confirming its suitability for real-time edge deployment of our usecase.

The average inference speed on Jetson ORIN per image fell from around 40 milliseconds (ms) for the *PyTorch* model to approx. 35 ms for the optimized ONNX model. Implementation of the model in the TensorRT framework resulted in a decrease of a further 20 ms. The 15 ms per image is explained by the deep optimization capabilities of TensorRT, including

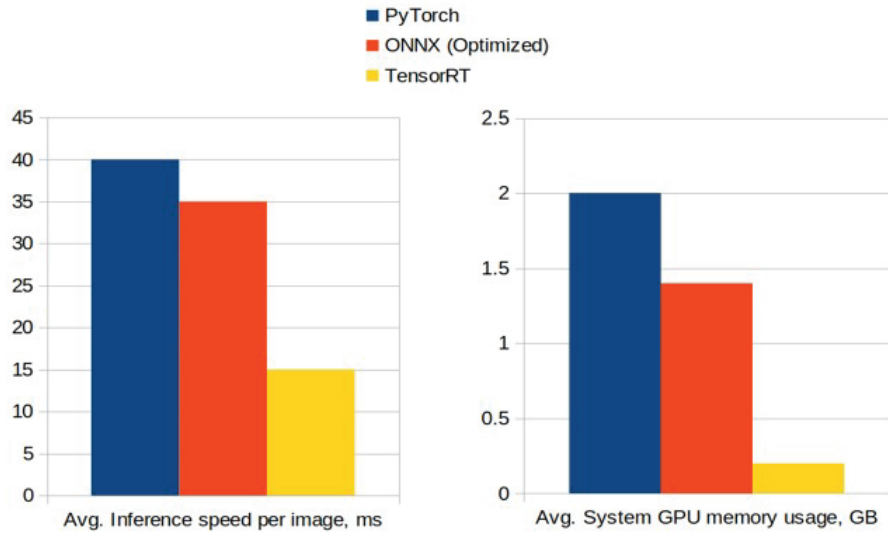

Figure 10.3 Comparison of average inference speed and average GPU memory usage between the three different iterations of the re-identification model implementation in different frameworks: Pytorch, ONNX and TensorRT.

kernel fusion, layer reordering, and dynamic tensor memory management. These optimizations allow the engine to exploit the Jetson's GPU and tensor cores more effectively than generic inference runtimes.

In addition to latency improvements, a notable reduction in average GPU memory usage was observed across the three versions. The *PyTorch* implementation consistently required the largest memory allocation (∼2GB) due to its dynamic computation graph, while ONNX reduced memory overhead (∼1,4GB) through static graph optimization. The NVIDIA native TensorRT engine achieved the lowest average GPU memory footprint (∼0.2GB), indicating efficient memory reuse and reduced buffer fragmentation during inference.

Overall, these results confirm that deploying the model through TensorRT significantly enhances both inference speed and resource efficiency, enabling the proposed re-identification system to operate within the real-time constraints on the edge.

It is also worthy of mentioning that the results gained in this chapter do not nescessarily correspond to the preference that the ONNX Runtime garnered in the chapter for Object detection results. It is evident that each

of the models (and their surrounding software systems) may better perform with a different engine and runtime than the other. These differences may be attributed to the different models used, the various software packages enabling the systems and various surrounding code implementations like batching, pre- and post- processing, filtering or motion gating, sizes of input data and others.

10.4.2.2 Re-identification on Fisheye projection images results

To evaluate the potential effect of fisheye distortion on vehicle re-identification performance, the previously used AICity test scenario was reprocessed with a synthetic fisheye lens transformation applied to all frames. The same model was then evaluated on both the original and the distorted datasets under identical conditions.

The accuracy results are summarized in Table 10.3. The overall accuracy difference between the normal and fisheye-transformed versions is minimal, suggesting that the current model maintains a largely stable performance despite geometric distortion.

The minor difference in re-identification accuracy, together with the nearly identical average cosine similarity in the embedding space, indicates that the model's feature representations are relatively robust to moderate fisheye distortion. Given the small scale of variation, these results are considered preliminary and within expected uncertainty.

10.4.2.3 System integration results

To verify the reliability of the deployed modular system and the effectiveness of its internal data flow, message transmission and reception statistics were collected during testing with the AICity dataset. The three key metrics observed were the number of detection messages published by the object detection module, the number of messages received by the feature extraction module, and the number of vehicle detections that were finally processed and forwarded to the database module.

The results are summarized in Table 10.4. The one-to-one correspondence between published and received messages confirms the stability and lossless

Table 10.3 Effect of fisheye distortion on vehicle feature extraction

Metric	Normal video	Fisheye video
Re-ID CMC Rank-1 Accuracy (%)	72.90	73.24
Average Cosine Similarity (across IDs)	0.6526	0.6317

Table 10.4 MQTT communication and filtering statistics during test on AICity dataset videos

Metric	Count
Detection module messages published	5771
Feature extraction module messages received	5771
Feature extraction module detections processed and forwarded	473

operation of the MQTT-based communication setup. Furthermore, the significantly lower number of forwarded detections demonstrates the intended behaviour of the zone-based and motion-gating filtering mechanisms within the feature extraction module to reduce processing load on the edge device.

These results validate that the system integration is operational and that the intra-pipeline communication is reliable. The filtering mechanisms effectively reduce the computational load, allowing the feature extraction model to focus on the most relevant vehicle observations for re-identification.

10.5 Discussion and Future Work

As described in [2], scaling a network camera perception system across a wider area is possible. The prototype implementation described in the previous chapters served as a proof of concept for validating the pipeline under real-world constraints. This setup has been deployed on a single Jetson device with a MQTT-based communication layer. This configuration successfully demonstrated the technical feasibility and performance of the ReID system.

Additionally, detecting and monitoring traffic via computer vision and deep learning methods require moving away from a single data collection and management site, as stated in [18]. Both the processing power of a single machine and the capacity of network paths are bottlenecks in smart city scenarios and a multi-tier edge computing based architecture is proposed. This architecture extends our proof-of-concept version to a distributed, multi-layered infrastructure with secure communication between multiple Road-Side Perception Units (RSPUs) and a meta-edge server.

In the envisioned architecture, the modular design established so far is preserved, but communication and data management responsibilities are expanded across two levels: a deep-edge layer, comprising the Jetson-based RSPUs, and a meta-edge layer, hosting data aggregation, indexing, and visualization services. This separation allows for efficient message handling, cross-device data fusion, and secure long-term storage, while maintaining low latency for inference tasks executed at the edge.

The following subsections describe in detail the communication mechanisms, data security provisions, and operational scheduling that enhance the envisioned system architecture.

10.5.1 Explanding the architecture

While the current proof-of-concept architecture featured a local MQTT broker and *OpenSearch* database deployment on the same Jetson device, this architecture will be improved in the deployments going forward with parts of the architecture having already been tested out.

Since the *OpenSearch* database can quickly grow during real-world deployment, it might not be possible to have it on the individual Jetson edge device. The Jetson edge devices will be joined by the meta-edge server in the architecture. On board the Jetson, we will have an MQTT broker and it will be coupled with an InfluxDB [19]. At the meta-edge server, we will also have a stack comprising Kafka and OpenSearch. Although there might be a small gain in using InfluxDB for the ReID case, it will be used by other regions of interest. The dataflow will be enabled by an MQTT broker on the Jetson that's connected to a Kafka Proxy (on the same Jetson), streaming data to the *OpenSearch* on the meta-edge and further to Grafana.

A dedicated module forwards MQTT messages published on the corresponding topic to a *Kafka* broker that is hosted on a server at higher edge layer (meta-edge) and aggregates messages from multiple Jetson devices. Another module at the meta-edge forwards messages published on a dedicated Kafka topic to a corresponding index of the database operations module (*OpenSearch* repository).

10.5.2 Dual Data Fusion mechanism for supporting communications

Communication within each module on board the Jetson and communication between multiple Jetson devices is made possible via a **dual data fusion mechanism**.

This mechanism is based on the use of the pub/sub messaging pattern that allows for loosely coupled and scalable interaction between modules, efficient and robust exchange of information [20, 21], thus facilitating integration and interoperability. In this context, a distributed messaging system is employed on a dual level: (1) at the level of each RSPU Jetson device (deep-edge layer) enabling communication between modules and microservices hosted on board the device and (2) at the level of a central server

(meta-edge layer), enabling communication between multiple RSPU Jetson devices through the server. On each level, the messaging system is responsible for aggregating and fusing data from modules that act as data producers and re-distributing it to all necessary modules that act as data consumers. At the deep-edge layer, the messaging system is implemented through the use of an MQTT message broker, which is more lightweight in terms of computational resource demands, while at the meta-edge layer, it is implemented through the use of an Apache Kafka broker that requires increased resources, but offers enhanced reliability and scalability features.

The data fusion mechanism is complemented with the following modules:

1. On board the Jetson devices, a **proxy service** is deployed that is responsible for subscribing to selected MQTT topics of the deep-edge message broker and relaying the received information to the meta-edge by publishing it to corresponding topics of the Kafka message broker.
2. At the meta-edge server, a **data repository** based on the *OpenSearch* stack is deployed, where information published to the *Kafka* message broker is persistently stored. This is achieved using a dedicated **ingestion service** that subscribes to selected *Kafka* topics and registers the incoming published data to corresponding indices of the OpenSearch data repository. The data remains on this repository for any subsequent access, including the use of vector search queries, natively supported by the API exposed by *OpenSearch*.

The data fusion mechanism includes a **security layer** implemented on both architecture levels (i.e., deep-edge and meta-edge). At the deep-edge level, the security layer refers to the protection of communications between the MQTT pub/sub messaging system and MQTT producer and consumer clients within each RSPU device. In such a configuration, even though both MQTT broker and clients are hosted on the same RSPU device, the clients could still be exposed to local threats, such as privilege escalation, malware or insider threats, while the communication could be susceptible to Man-in-the-Middle (MitM) attacks. In addition, future needs of the data fusion mechanism may demand the connection of the MQTT broker to clients that are deployed on other hosts. The provided solution addresses all these issues and is based on X.509 certificates that are used for authenticating MQTT clients and enabling TLS encryption in communications and is complemented by custom RBAC configurations that can provide fine-grained authorization of clients to access MQTT topics. At the meta-edge level, a similar solution is provided, i.e., X.509 certificates, mTLS authentication and encryption, a

centralised PKI and RBAC mechanisms are used to protect communications with the Kafka-based pub/sub messaging system. In this case, there are multiple producer clients at the deep-edge level that propagate data from the deep-edge level to the meta-edge level, where the Kafka broker is hosted. Each such producer client is embedded within the aforementioned **proxy service** that acts as a central gateway on each RSPU for communications with the meta-edge. This design of a single point-of-contact further reinforces cyber-security protection as it reduces the attack surface to a single service on board each RSPU. Finally, it should be noted that the entire deployment is limited to the edge layer and only extends to a meta-edge system. All the host computational infrastructure is self-managed and/or on-premises, thus contributing to a further increase of secure and reliable edge analytics by design. The absence of a cloud layer or infrastructure managed by a third party provides an additional level of security, as **(a)** data control is exclusively maintained and not delegated to a third party, **(b)** there is a reduced risk of unauthorized access and data breaches because the involved resources, networks and data can be physically or logically isolated from external sources and **(c)** the entire deployment is less exposed to external access, leading to a reduction of the potential attack surface.

10.5.3 Scheduling plan for deployment

The Jetson devices are deployed as part of a networked infrastructure of multiple Road-Side Perception Units (RSPUs) that are responsible for a series of functions apart from the ones that explicitly support the re-identification use case described herein. For example, RSPUs are also used to collect data from deployed environmental sensors and to process these data to perform Air Quality Index (AQI) Forecasting. In this use case, RSPUs are also used for local ML model training based on locally aggregated data and are connected through a server at the meta-edge layer, as part of a Federated Learning framework. Special provisions have been made to allow for complementarity between the vehicle re-identification use case and the AQI Forecasting use case. In particular, a challenge that was considered was that the local ML model training for the AQI Forecasting use case has a significant computational footprint, demanding increased resources for a period of a few hours per day in order to process the data collected within the day and produce an updated ML model. A concurrent operation of the ML training and the inference for vehicle re-identification would not be possible, as the latter is also considerably demanding in terms of consumed computational resources.

This conflict was solved by scheduling the ML training process to be activated during nighttime, when the vehicle re-identification use case is inactive due to lack of visibility from the deployed cameras. This results in an efficient and complementary operation for both use cases and demonstrates the wide spectrum of application fields that the RSPU can handle.

Acknowledgements

This work was supported by Chips Joint Undertaking EdgeAI project. The project EdgeAI "Edge AI Technologies for Optimised Performance Embedded Processing" is supported by the Chips Joint Undertaking and its members including top-up funding by Austria, Belgium, France, Greece, Italy, Latvia, Netherlands, and Norway under grant agreement No 101097300.

References

[1] X. Li and Z. Zhou, "Object Re-Identification Based on Deep Learning," *IntechOpen eBooks*, Jul. 2019, https://doi.org/10.5772/intechopen.86564.

[2] J. Barthélemy, N. Verstaevel, H. Forehead, and P. Perez, "Edge-Computing Video Analytics for Real-Time Traffic Monitoring in a Smart City," *Sensors*, vol. 19, no. 9, p. 2048, May 2019, https://doi.org/10.3390/s19092048.

[3] T. Zutis *et al.*, "Multi-Step Object Re-Identification on Edge Devices: A Pipeline for Vehicle Re-Identification", *Charting the Intelligence Frontiers – Edge AI Systems Nexus*, 2025, https://doi.org/10.13052/rp-9788743808831. https://cloud.riverpublishers.com/pdf/ebook/RP_E9788743808831.pdf

[4] Glenn Jocher and Ayush Chaurasia and Jing Qiu, 2023, Ultralytics YOLOv8, 8.0.0, https://github.com/ultralytics/ultralytics.

[5] Y. Zhang *et al.*, "ByteTrack: Multi-Object Tracking by Associating Every Detection Box," *arXiv:2110.06864 [cs]*, Apr. 2022, https://arxiv.org/abs/2110.06864

[6] L. Karumbunathan, "NVIDIA Jetson AGX Orin Series A Giant Leap Forward for Robotics and Edge AI Applications Technical Brief," *NVIDIA Jetson AGX Orin Series Technical Brief*, vol. 1, no. 2, 2022, https://www.nvidia.cn/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf

[7] Abbas Aqeel Kareem, Dalal Abdulmohsin Hammood, and Ruaa Ali Khamees, "Optimizing Siamese neural network with TensorRT on NVIDIA jetson nano," *AIP conference proceedings*, Jan. 2023, https://doi.org/10.1063/5.0154881.

[8] M. Gochoo *et al.*, "FishEye8K: A Benchmark and Dataset for Fisheye Camera Object Detection." https://openaccess.thecvf.com/content/CVPR2023W/AICity/papers/Gochoo_FishEye8K_A_Benchmark_and_Dataset_for_Fisheye_Camera_Object_Detection_CVPRW_2023_paper.pdf

[9] K. Liao *et al.*, "Deep Learning for Camera Calibration and Beyond: A Survey." https://arxiv.org/pdf/2303.10559

[10] Z. Tang *et al.*, "CityFlow: A City-Scale Benchmark for Multi-Target Multi-Camera Vehicle Tracking and Re-Identification." Accessed: Oct. 17, 2025. https://openaccess.thecvf.com/content_CVPR_2019/papers/Tang_CityFlow_A_City-Scale_Benchmark_for_Multi-Target_Multi-Camera_Vehicle_Tracking_and_CVPR_2019_paper.pdf

[11] J. Yang, S. Liu, Z. Li, X. Li, and J. Sun, "Real-time Object Detection for Streaming Perception." Accessed: Oct. 17, 2025. https://openaccess.thecvf.com/content/CVPR2022/papers/Yang_Real-Time_Object_Detection_for_Streaming_Perception_CVPR_2022_paper.pdf

[12] Y. Zhao *et al.*, "DETRs Beat YOLOs on Real-time Object Detection." https://openaccess.thecvf.com/content/CVPR2024/papers/Zhao_DETRs_Beat_YOLOs_on_Real-time_Object_Detection_CVPR_2024_paper.pdf

[13] P. Ganesh, Y. Chen, Y. Yang, D. Chen, and M. Winslett, "YOLO-ReT: Towards High Accuracy Real-time Object Detection on Edge GPUs." Accessed: Oct. 17, 2025. https://openaccess.thecvf.com/content/WACV2022/papers/Ganesh_YOLO-ReT_Towards_High_Accuracy_Real-Time_Object_Detection_on_Edge_GPUs_WACV_2022_paper.pdf

[14] Z. Fu, Y. Chen, H. Yong, R. Jiang, L. Zhang, and X.-S. Hua, "Foreground Gating and Background Refining Network for Surveillance Object Detection," *IEEE transactions on image processing*, vol. 28, no. 12, pp. 6077–6090, Jun. 2019, https://doi.org/10.1109/tip.2019.2922095.

[15] J. Luiten *et al.*, "HOTA: A Higher Order Metric for Evaluating Multi-object Tracking," *International Journal of Computer Vision*, vol. 129, no. 2, pp. 548–578, Oct. 2020, https://doi.org/10.1007/s11263-020-01375-2.

[16] T. Xiao, "Evaluation Metrics — Open-ReID documentation," *Github.io*, 2017. https://cysu.github.io/open-reid/notes/evaluation_metrics.html.

[17] "Measuring similarity from embeddings," *Google for Developers*, 2025. https://developers.google.com/machine-learning/clustering/dnn-clustering/supervised-similarity.

[18] G. Liu *et al.*, "Smart Traffic Monitoring System Using Computer Vision and Edge Computing," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 8, pp. 1–12, 2021, https://doi.org/10.1109/tits.2021.3109481.

[19] M. Schneider *et al.*, "Open Traffic Data for Mobility-as-a-Service Applications – Architecture and Challenges," *River Publishers eBooks*, pp. 375–385, Sep. 2022, https://doi.org/10.1201/9781003337232-31.

[20] Kansakar, Anila. "Integrating Message Queuing Telemetry Transport (Mqtt) With Kafka Connect For Processing Iot Data." Diss. Pulchowk Campus, 2019. https://elibrary.tucl.edu.np/JQ99OgQIizUxyjI9nB0on9OyLkqsGIf4/api/core/bitstreams/6c7b6130-0691-44bd-bc86-51156784d20c/content

[21] K. N. Haque, "Decentralized pub/sub architecture for real-time remote patient monitoring," *Urn.fi*, 2024, https://oulurepo.oulu.fi/handle/10024/51125.