computation

MDPI

Article

Performance Analysis of the YOLO Object Detection Algorithm in Embedded Systems: Generated Code vs. Native Implementation

Pablo Martínez Otero [1,*], Alberto Tellaeche [1,*] and Mar Hernández Melero [2]

[1] Computer Science, Electronics and Communication Technologies Department, Universidad de Deusto, Avenida de las Universidades 24, 48007 Bilbao, Bizkaia, Spain
[2] Embedded Systems, IKERLAN, Paseo José María Arizmendiarrieta 2, 20500 Arrasate, Gipuzkoa, Spain; mar.hernandez@ikerlan.es
* Correspondence: pablomartinez171299@gmail.com (P.M.O.); alberto.tellaeche@deusto.es (A.T.)

Abstract

This paper evaluates the current maturity of automatic code-generation workflows for deploying modern CNN-based object detectors on embedded GPU platforms. We compare a native pipeline against a code generation pipeline through a Model-Based Engineering (MBE) approach, using YOLOv8/YOLOv9 inference on NVIDIA Jetson Orin Nano and Jetson AGX Orin as representative edge-GPU workloads. We report detection-quality metrics (mAP, PR curves) and system-level metrics (latency distribution and initialization overhead) under a controlled single-class scenario based on a CARLA-generated sequence with frame-level annotations. Absolute accuracy and latency values are scenario-dependent and may vary under different camera optics, illumination, motion blur, sensor noise, occlusion patterns, and multi-class scene. Results quantify the performance gap between code generation and native pipelines and show that, for the evaluated workloads, the automated pipeline remains less competitive in both latency and accuracy. We discuss the implications of this gap for deployment workflows in safety-oriented domains, and we outline bottlenecks that should be addressed. The study is intended as a controlled traffic-light detection micro-benchmark and does not aim to validate full ADAS perception stacks.

Keywords: YOLO; deep learning; embedded systems; traffic light detection; micro-benchmark; code generation

1. Introduction

Nowadays, artificial intelligence (AI) plays a crucial role in technological development. One of the fields in which it has had the greatest impact over the past decade is autonomous driving. The emergence of one-stage Neural Networks (NNs) [1], such as You Only Look Once (YOLO) [2], has revolutionized this field by achieving a remarkable balance between detection speed and accuracy [3]. The implementation of such algorithms enables real-time object detection as a key building block of perception pipelines. This is focused on the traffic-light detection subtask, which is safety-relevant and latency-sensitive in embedded deployments. In this work, traffic-light detection is used as a representative case study for deployment-oriented benchmarking, rather than as a claim of full-stack perception validation.

One of the most relevant tasks of any autonomous vehicle is the reliable detection of traffic light signals [4], a process that can be challenging due to the variability of lighting or weather conditions, occlusions, and design variations. Failure to accurately detect or

Academic Editors: Taqwa AlHadidi, Shadi Jaradat and Ahmed Jaber

Received: 10 February 2026
Revised: 3 March 2026
Accepted: 9 March 2026
Published: 12 March 2026

interpret a traffic light can result in catastrophic consequences, including traffic collisions that may lead to serious injuries or even fatalities, highlighting the critical importance of robust and reliable detection systems in autonomous vehicles [5].

While real-world traffic-light detection is affected by camera optics, illumination changes, motion blur, sensor noise, multi-class clutter, and diverse occlusion patterns, systematically covering this operational diversity is outside the scope of this manuscript. Instead, we intentionally adopt a controlled single-class CARLA sequence to maximize internal validity and isolate the impact of the deployment pipeline itself. Consequently, the reported accuracy and latency results should be interpreted as pipeline-level comparative measurements under the stated configuration, rather than as general claims about traffic-light perception robustness in unconstrained driving conditions.

Despite advances in the development of AI-based object detection algorithms, their deployment in autonomous vehicles must consider the practical constraints of embedded systems. As these systems rely on edge computing, where a large amount of data is processed locally in the vehicle [6], they must operate with limitations on battery capacity, computational power, and storage [7]. Given that AI-based algorithms are among the most computationally demanding components, these limitations directly impact the feasibility of deploying complex models in real-time scenarios.

In light of these challenges, there is a growing need for object detection models that can maintain high accuracy while meeting the constraints imposed by embedded systems. This study is motivated by the safety relevance of traffic-light detection and the practical constraints of deploying object detectors on embedded platforms. Despite the growing adoption of Model-Based Engineering and automatic code generation in safety-oriented domains, there is limited public evidence on how close these toolchains are to state-of-practice native pipelines for modern deep learning workloads on embedded GPUs. In practice, engineers must trade off development productivity, certification arguments, and portability against latency and accuracy. This paper addresses the following question: how large is the current performance gap between automatic code-generation and native pipelines when deploying modern object detectors on embedded GPU platforms?

Accordingly, this work contributes a controlled deployment-oriented benchmark for comparing native inference and automatic code generation under identical weights, input resolution, and evaluation criteria on embedded GPUs. Using NVIDIA Jetson Orin Nano and Jetson AGX Orin (NVIDIA Corporation, Santa Clara, CA, USA), we quantify the resulting gap in detection quality and latency, and we summarize practical insights into the maturity and limitations of current code-generation workflows, including failure cases for larger models.

2. Related Work

In urban environments, autonomous vehicles rely on comprehensive environment perception and intelligent control systems to navigate safely and efficiently. In order to operate safely, an autonomous vehicle must be equipped with a reliable perception system capable of detecting, identifying, and tracking various types of objects in its environment, including traffic signals, other vehicles, pedestrians, and cyclists. This task is particularly complex, not only because of the high accuracy required for reliable object detection, but also because environmental changes often cause discrepancies in the data captured by the vehicle's sensors [8,9]. While full-stack perception systems typically address multiple road users and objects, controlled single-class benchmarks remain useful to isolate deployment effects and study latency trade-offs on embedded hardware; traffic-light detection is used here as a representative case study.

Modern self-driving cars integrate multiple sensor modalities, including LiDAR, radar, cameras, and ultrasonic sensors to capture a robust view of their surroundings [10–12].

Once the environment has been perceived through the vehicle's integrated sensors, the acquired data must be interpreted in real time to enable accurate and responsive vehicle control. Over the past decade, traffic light detection has undergone a significant evolution, transitioning from traditional image processing techniques to predominantly deep learning-based approaches [13–19], which have become the standard due to their enhanced accuracy, adaptability, and robustness, especially in real-time applications.

Among the various approaches based on deep learning, Convolutional Neural Networks (CNNs) have emerged as the most effective tools for object detection tasks, including traffic light recognition [20,21]. Architectures such as YOLO (You Only Look Once) [22,23], SSD (Single Shot MultiBox Detector) [24,25], and R-CNN (Region-Based Convolutional Neural Networks) [15,26,27] have demonstrated remarkable performance in detecting traffic lights with high precision and low latency, which makes them particularly well suited for real-time autonomous driving scenarios. These models differ fundamentally in their detection pipelines: single-stage detectors perform object localization and classification in a single forward pass through the network, enabling faster inference. In contrast, two-stage detectors first generate region proposals and then classify each proposed region in a second pass, typically achieving higher accuracy at the cost of increased computational complexity [28,29]. These differences are illustrated in Figure 1.

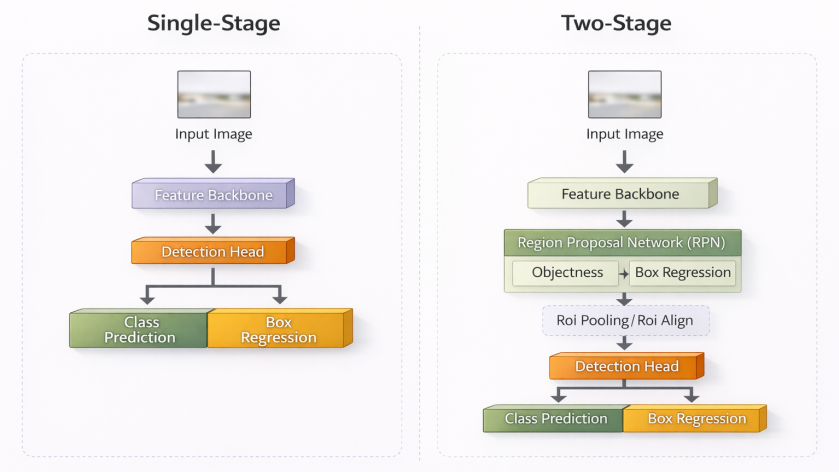

Figure 1. Single-stage and two-stage architectures.

To better illustrate the comparative strengths of the mentioned architectures in the context of real-time object detection, Table 1 compiles the general characteristics and performance aspects reported in the literature [30,31]. As shown, YOLO achieves the most balanced trade-off between detection precision and processing efficiency, making it a preferred choice for deployment in autonomous vehicles.

Table 1. Comparison of CNN-based object detection architecture performance.

Model	Architecture	mAP (%)	Inference Speed	Usability
Faster R-CNN	Two-stage	83	Low	Avoid
SSD	Single-stage	75	Medium	Poor choice
YOLOv8	Single-stage	89	High	Best compact model

YOLO is widely recognized as the practical standard for object detection in autonomous driving [32]. Unlike traditional approaches that rely on multistage object detection pipelines, such as R-CNN and its variants [33], YOLO adopts a unified architecture

that performs object detection in a single pass through the neural network. This results in significantly faster processing speeds without substantially compromising accuracy [34,35], making YOLO especially suitable for real-time scenarios [36].

Throughout its successive versions, the YOLO architecture has consistently improved its detection accuracy and computational efficiency [37–41]. Key advancements include the adoption of multiscale detection strategies, spatial attention mechanisms, and optimized loss functions, all contributing to more robust performance in real-time and complex traffic scenarios.

Given these continuous improvements in accuracy and efficiency, recent YOLO versions have become increasingly viable for deployment in embedded systems, which are commonly used in autonomous vehicles and traffic monitoring infrastructure. However, implementing YOLO in such resource-constrained environments presents several challenges. Embedded systems often have limited computational power, memory capacity, processing speed, and energy availability [42,43], making it essential to optimize the size of models and their inference time [7,44]. These studies [45,46] demonstrate that lightweight versions of YOLO, such as YOLOv8n, have been specifically developed to balance detection performance with inference speed, making them well suited for deployment on resource-constrained embedded hardware.

More recently, several works have proposed architecture-level optimizations to further improve YOLO-based detection in resource-constrained environments. For instance, Chen et al. propose a lightweight perception framework for railway foreign object detection that integrates re-parameterizable bottlenecks and lightweight self-attention mechanisms, achieving competitive accuracy at only 1.7 GFLOPs and demonstrating deployment feasibility on embedded platforms such as Jetson TX2 [47]. Similarly, recent embedded-oriented YOLO variants have explored structural redesign, feature refinement strategies, and task-specific enhancements to balance detection performance and computational cost in UAV and edge-computing scenarios [48,49]. In the context of autonomous driving, Wang et al. propose TCE-YOLOv5, a lightweight object detection architecture based on YOLOv5 that incorporates structural compression and efficiency-driven enhancements to achieve real-time performance while reducing parameter count [50].

These approaches primarily focus on modifying the network architecture, introducing pruning strategies, or incorporating attention mechanisms to reduce computational complexity while preserving accuracy. In contrast, the present work does not alter the internal architecture or training procedure of the evaluated YOLO models. Instead, it investigates how different deployment paradigms, the native pipeline versus the code-generation pipeline, affect inference performance under identical model configurations. Therefore, our contribution is complementary to architecture-level optimization studies, as it isolates and quantifies the impact of the implementation pipeline itself.

However, the deployment of CNNs in embedded systems has progressed through various strategies, with one of the most prominent being the integration of deep learning into Model-Based Engineering (MBE) [51] and automatic code generation. The literature [52] provides a comparative analysis of automatic code generation versus manual coding, highlighting the respective benefits and limitations of each approach.

While automatic code generation and Model-Based Engineering have been widely adopted in safety-critical software development, relatively few studies provide systematic performance evaluations of deep learning inference pipelines generated through these workflows on embedded GPU platforms. Most existing works focus either on architectural optimization of neural networks or on high-level productivity and maintainability aspects of model-based design, rather than on quantitative latency and accuracy comparisons between native deep learning frameworks and automatically generated code under identical

deployment conditions. Consequently, empirical evidence regarding the maturity and performance gap of code-generation toolchains for modern object detection models on embedded hardware remains limited.

Based on this context, the present study focuses on analyzing the performance of YOLOv8 [53] and YOLOv9 [54] by comparing their deployment through two distinct deployment pipelines: a native pipeline and a code-generation pipeline based on a Model-Based Engineering (MBE) workflow.

3. Materials and Methods

The proposed methodology follows the complete embedded vision pipeline, beginning with the systematic generation and validation of a traffic light Ground Truth. Then it proceeds with the deployment of each model using both the native pipeline and the code-generation pipeline, and the definition of evaluation metrics that consider both detection accuracy and computational efficiency. These methodological steps provide a structured and consistent basis for the comparative analysis developed in the subsequent sections.

3.1. Ground Truth

The reference data, or Ground Truth, used for evaluation was obtained through systematic annotation of a sequence generated in the CARLA (Car Learning to Act) simulator [55], depicting a realistic urban driving scenario. The sequence includes a clearly visible moving traffic light as the sole object of interest and was rendered at a resolution of 640 × 640. This setup allows for controlled testing of perception models under consistent and reproducible conditions. A systematic annotation approach was adopted to ensure high accuracy, which is essential for a reliable evaluation of the YOLO algorithm. Annotation was performed following a well-defined set of criteria, illustrated in Figure 2, which specify the conditions under which a label is considered valid. This protocol was designed to minimize subjectivity and to ensure consistency between annotations. Deviations from established criteria would compromise the integrity of Ground Truth data and, consequently, affect the validity of the results presented in this study. Since YOLO's performance is evaluated based on its ability to detect traffic lights with respect to this dataset, any inconsistencies in the labeling process would directly impact the reliability of the evaluation metrics and the conclusions drawn from them.

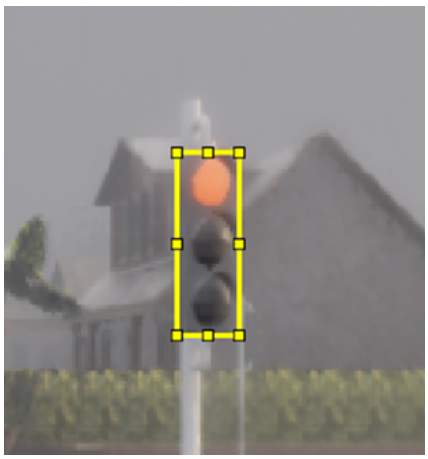

Figure 2. Ground Truth labeling acceptance criteria, representing the minimum valid annotation and defining the required object localization and bounding-box precision.

3.2. Dataset Composition and Annotation Protocol

The evaluation dataset consists of 400 consecutive frames extracted from a video sequence generated in the CARLA simulator (v0.9.14; CARLA Team, Computer Vision

Center, Bellaterra, Barcelona, Spain). The video was directly rendered at a fixed spatial resolution of 640 × 640 pixels, matching the input resolution of the pre-trained YOLO models used in this study (trained on the COCO dataset (Microsoft Research, Redmond, WA, USA)). No additional resizing was applied during evaluation.

Frames were processed independently; although they originate from a continuous driving sequence and therefore exhibit temporal correlation, no temporal filtering, tracking, or smoothing was applied. Each frame was treated as an independent image during inference and evaluation.

All annotations were performed manually on a frame-by-frame basis using the MATLAB Video Labeler tool (Computer Vision Toolbox, MathWorks, Version 24.2, Release R2024b). Bounding boxes were defined as the minimal axis-aligned rectangle tightly enclosing the visible traffic light, following the strict acceptance criteria illustrated in Figure 2. The annotation protocol intentionally avoided oversized or loosely fitted bounding boxes to minimize localization ambiguity.

The dataset includes frames both with and without visible traffic lights, reflecting realistic driving conditions. Frames without annotated objects were retained in the evaluation, and any detections in such frames were counted as false positives.

Since the dataset contains a single object class and was annotated by a single operator under a predefined labeling protocol, inter-annotator variability was not applicable. A second manual verification pass was performed to ensure labeling consistency across the sequence.

The objective of this dataset is not to provide statistical generalization across diverse domains, but to enable a controlled, deployment-oriented comparison under consistent visual and environmental conditions.

3.3. Pre-Processing and Post-Processing Pipeline

The pre-processing and post-processing stages were not modified from their default configurations in the respective deployment frameworks. Since the input sequence was already rendered at 640 × 640 pixels, no additional resizing or letterboxing was required.

For the native pipeline implementation, the standard YOLO inference pipeline provided by the Ultralytics framework was used without modification. This includes normalization of input pixel values to the range [0, 1], channel reordering as required by the model, and built-in non-maximum suppression (NMS).

Similarly, the code-generation pipeline (Model-Based Engineering, MBE) followed the default preprocessing and post-processing settings defined by the corresponding MATLAB/Simulink YOLO implementation.

In all experiments, pre-processing, neural inference, and post-processing were executed as an integrated pipeline. Inference time measurements therefore include the complete processing chain rather than isolated forward-pass latency.

Inference was executed using the default confidence and non-maximum suppression (NMS) parameters defined by each framework (Ultralytics for the native pipeline and the corresponding MATLAB/Simulink backend for code generation). No modifications were applied at inference time. For evaluation purposes, detections with confidence scores below 0.3 were discarded during post-processing in MATLAB prior to computing performance metrics. This additional filtering step was applied uniformly to both deployment pipelines to ensure consistent metric extraction.

The objective of this work is to compare complete deployment pipelines rather than isolated neural forward passes; therefore, all reported latency measurements and detection metrics reflect end-to-end behavior under default backend configurations.

3.4. Evaluation Metrics

To comprehensively assess the performance of YOLOv8 and YOLOv9 in the task of traffic light detection within embedded systems, a set of well-established evaluation metrics has been defined, following the definitions and formulations presented in the literature [56]. These metrics are selected to reflect both detection accuracy and computational efficiency, which are critical factors in real-time applications. The evaluation considers inference time, mean Average Precision (mAP), precision–recall (PR) curves, and confusion matrices computed across multiple Intersection over Union (IoU) thresholds. In this study, IoU thresholds range from 0.5 to 0.9 in increments of 0.1, allowing a detailed analysis of the detection robustness under varying localization strictness. All detection metrics (including mAP@0.5 and mAP@0.5:0.95) were computed deterministically over the complete evaluation dataset using fixed model weights and a single ground-truth set. Since mAP corresponds to a single dataset-level aggregated evaluation rather than repeated independent trials, confidence intervals were not estimated. Obtaining statistical uncertainty estimates for mAP would require additional resampling procedures (e.g., bootstrap over images), which falls outside the scope of this deployment-oriented benchmarking study.

On the other hand, the inference time refers to the time elapsed between receiving an input frame and generating the corresponding detection output. It is measured in milliseconds (ms) and represents the computational efficiency of the model on a given hardware platform. Lower inference times are critical for real-time deployment scenarios, particularly in embedded systems with constrained processing resources.

3.5. Experimental Setup

The experimental setup relies on the coordinated use of several tools and platforms, each selected for its suitability to the requirements of embedded perception systems and real-time evaluation. For this purpose, MATLAB/Simulink (Version 24.2, R2024b; MathWorks, Natick, MA, USA). was used in its R2024b release [57], which represents the latest advancements in the MBE workflow for control and perception software. This environment allows engineers and researchers to model algorithms graphically, validate them via simulation, and generate production-grade C/C++ or CUDA code for embedded ECUs and GPUs within a unified toolchain. MATLAB/Simulink provides an ISO 26262 qualification kit and tool certification support up to Automotive Safety Integrity Level (ASIL) D for specific toolchains and usage conditions [58]. This certification applies to the tool qualification process, not to the generated application software itself. The object detection system evaluated in this work is a research prototype and is not safety-certified.

Robot Operating System 2 (ROS 2) Humble Hawksbill (Open Robotics, San Jose, CA, USA) was selected as middleware for communication and synchronization between modules, specifically the Humble Hawksbill distribution. While the original ROS [59] introduced a master-based architecture for rapid message passing, it lacked real-time determinism and Quality of Service (QoS) guarantees [60]. ROS 2 addresses these limitations by leveraging the Data Distribution Service (DDS) standard, eliminating the central master, and introducing features such as configurable QoS policies, multicast discovery, and secure communication. These capabilities enable submillisecond latencies and bounded jitter in multicore embedded systems [61], making ROS 2 a suitable middleware for real-time applications in autonomous driving. This tool has enabled us to develop a synchronized system capable of ensuring that no data is duplicated or lost during the evaluation.

The object detection models selected for this research are recent high-performing architectures widely adopted in computer vision tasks. Since network training is not part of the scope of this evaluation, the selected models were verified to share an identical training dataset and input resolution to ensure a fair and objective comparison. Specifically, for

the native pipeline approach, the Ultralytics-developed YOLOv8 and YOLOv9 models were used. Ultralytics is the organization responsible for maintaining and advancing the most recent iterations of the YOLO architecture, providing high-performance, production-ready implementations widely adopted in the computer vision community. These models were trained on the COCO dataset [62] with 640 × 640 input images, ensuring consistent training conditions for a fair and objective comparison. Correspondingly, the MBE approach employed the MATLAB-adapted versions of YOLOv8 [63] and YOLOv9 [64], trained under the same conditions.

Regarding computing platforms, Ubuntu 22.04 [65] was used as the operating system for development and testing on PC, while JetPack 6.2 [66] was used for the deployment on NVIDIA embedded hardware. Two NVIDIA platforms were selected for evaluation under realistic embedded conditions: the Jetson Orin Nano and the Jetson AGX Orin, both of which support GPU acceleration and real-time processing capabilities required for autonomous perception tasks [67].

For the native pipeline, Python version 3.10 was used in combination with PyTorch 2.3.0 [68] as the primary deep learning framework. This configuration ensures compatibility with the latest YOLO implementations while leveraging the hardware acceleration features of the underlying GPU architectures. The experimental setup was designed to validate and compare the performance of YOLOv8 and YOLOv9 models under realistic deployment conditions using a combination of X-in-the-Loop (XiL) strategies [69], specifically Model-in-the-Loop (MiL) [70] and Hardware-in-the-Loop (HiL) [71]. This hybrid evaluation approach enables both early-stage functional testing and final-stage validation on target hardware, ensuring consistency and reliability throughout the development cycle. The MiL stage, illustrated in Figure 3, was used to simulate and verify model behavior in a controlled environment, while the HiL stage, illustrated in Figure 4, allowed real testing using embedded hardware.

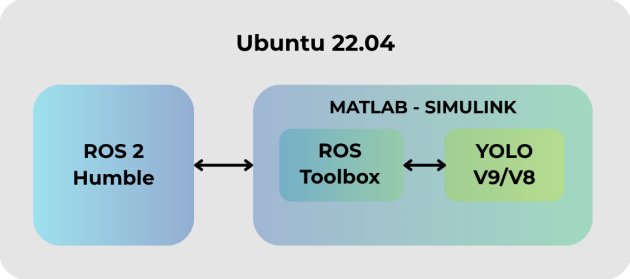

Figure 3. Model in the Loop scheme.

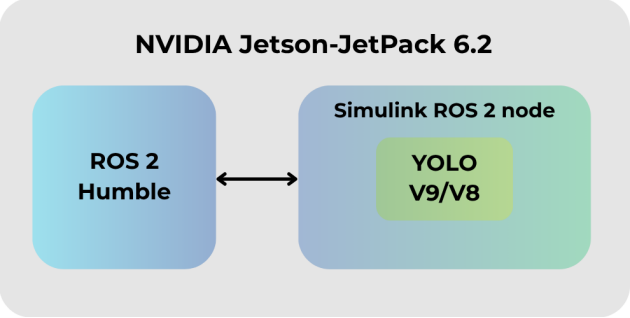

Figure 4. Hardware in the Loop scheme.

3.6. Hardware and Software Configuration

All experiments were conducted on NVIDIA Jetson Orin Nano and Jetson AGX Orin platforms running JetPack 6.2. Both devices were configured in MAXN/60 W power mode, respectively, to enable maximum performance.

No manual clock locking was applied, and default dynamic frequency scaling was preserved. The `nvpmodel` utility was not manually reconfigured beyond selecting the maximum performance mode.

All measurements were performed under standard laboratory thermal conditions after system warm-up, allowing the devices to reach steady-state operating temperature. No active thermal management modifications or forced cooling adjustments were applied. Therefore, reported results reflect realistic deployment behavior under sustained high-performance operation.

In addition to the sample mean and standard deviation, 95% confidence intervals (CIs) for the mean latency were computed as $\bar{t} \pm t_{0.975,\,n-1}\, s/\sqrt{n}$, with $n = 400$ frames and $t_{0.975,399} \approx 1.966$. Although frames originate from a continuous sequence, the CI is reported as an approximate uncertainty estimate for the sample mean over the evaluated sequence.

3.7. Reproducibility Checklist

To ensure transparency and facilitate independent replication of the experimental setup, a concise summary of all relevant configuration parameters is provided in Table 2. This checklist consolidates dataset characteristics, annotation protocol, inference settings, hardware configuration, and runtime conditions under which all measurements were obtained.

Table 2. Reproducibility checklist table.

Category	Configuration
Dataset size	400 consecutive frames
Input resolution	640 × 640 pixels
Temporal correlation	Yes (continuous video sequence)
Classes	1 (traffic light)
Annotation method	Manual frame-by-frame labeling
Bounding box policy	Tight axis-aligned bounding box
Confidence threshold (evaluation)	0.3
NMS IoU threshold	Default framework value
Evaluation IoU range	0.5–0.9 (step 0.1)
Batch size	1
Precision mode	FP32
Latency measurement	End-to-end (pre + inference + post)
Jetson Orin Nano power mode	MAXN
Jetson AGX Orin power mode	60W
Clock locking	Not applied
Dynamic frequency scaling	Enabled (default)
Thermal condition	Steady-state after warm-up
JetPack version	6.2
PyTorch version	2.3.0
MATLAB/Simulink version	R2024b

3.8. Latency Benchmarking Methodology

Latency was measured as end-to-end per-frame processing time, including pre-processing, neural inference, and post-processing. The neural forward pass was not isolated from the complete execution pipeline. All experiments were conducted with batch size equal to 1, consistent with real-time embedded perception scenarios.

For each model and deployment configuration, latency statistics were computed over the complete evaluation sequence of 400 frames (i.e., 400 iterations). The reported "Avg inference time" corresponds to the arithmetic mean across all frames, and the standard deviation reflects variability across the same 400 measurements. The maximum observed latency and its corresponding frame index are also reported to characterize worst-case behavior. No outlier removal or post hoc filtering was applied.

All measurements on Jetson platforms were performed after system warm-up under steady-state thermal conditions, as described in Table 2. Devices were configured in MAXN power mode, with default dynamic frequency scaling enabled and without manual clock locking.

3.9. Model Training Configuration and Domain Considerations

The YOLOv8 and YOLOv9 models used in this study were pre-trained on the COCO dataset and were not fine-tuned on the CARLA-generated sequence. The objective of this work is not to optimize detection performance for a specific domain, but to compare deployment pipelines under identical model conditions.

We acknowledge that evaluating COCO-pretrained models on a synthetic CARLA environment introduces domain shift, which may affect absolute detection performance. However, since all deployment approaches use the same model weights and are evaluated on the same dataset, relative performance differences between native and code-generated implementations remain comparable.

Therefore, reported accuracy metrics should be interpreted as deployment-level comparative indicators rather than as optimized domain-specific detection results.

3.10. Model Selection Rationale

YOLOv8 and YOLOv9 were selected because they represent relatively recent, high-performance single-stage detectors widely adopted in both academic and industrial contexts. Additionally, MATLAB-compatible implementations were available for these model families, enabling a fair comparison between the native pipeline and the code-generation pipeline under identical training conditions.

Within each family, multiple size variants (e.g., n, s, m, l, x/e) were evaluated to cover a broad range of parameter counts and computational budgets. The objective is not to compare YOLOv8 against YOLOv9 as competing architectures, but to assess whether the relative performance gap between the native pipeline and the code-generation pipeline remains consistent across different model complexities.

All evaluated models share the same input resolution (640 × 640) and pre-training dataset (COCO), ensuring comparable baseline conditions. Complexity metrics (parameter count, FLOPs, and model size) were obtained from the official Ultralytics documentation [53,54] and are summarized in Table 3 to facilitate transparent comparison across variants.

Table 3. Model complexity summary (640 × 640 input resolution).

Model	Parameters (M)	FLOPs (B)	Model Size (MB)
Yolov9t	2.0	7.7	4.73
Yolov9s	7.2	26.7	14.6
Yolov9m	20.1	76.8	39.0
Yolov9c	25.5	102.8	49.3
Yolov9e	58.1	192.5	112.0
Yolov8n	3.2	8.7	6.24
Yolov8s	11.2	28.6	21.5
Yolov8m	25.9	78.9	49.7
Yolov8l	43.7	165.2	83.7
Yolov8x	68.2	257.8	130.0

4. Results

This section presents the results obtained for the selected versions of YOLOv8 and YOLOv9, differentiating them according to their implementation approach: native or code generation pipelines in different NVIDIA hardware platforms. Performance metrics such as inference time, mAP, precision–recall curves, and confusion matrices across different IoU thresholds are analyzed to assess the effectiveness and efficiency of each configuration. Acronyms such as YOLOv9s or YOLOv8l used in this analysis represent standardized naming conventions for YOLO model sizes, where each suffix indicates the scale of the model, ranging from lightweight to more computationally intensive versions (e.g., n for nano, t for tiny, s for small, m for medium, l for large, x for extra large, e for extensive and c for compact).

4.1. Detection Performance

In terms of detection performance, Figures 5–8 present the Precision–Recall curves for each evaluated model, organized by YOLO version and implementation approach. Referring to the native pipeline, it can be observed that this approach consistently yields higher detection accuracy compared to the code generation pipeline. In particular, the native pipeline of YOLOv9e achieves an average precision (mAP) of 0.602 across IoU thresholds ranging from 0.5 to 0.9, which is a notably high value given the evaluation scenario. Similarly, YOLOv9m reaches 0.539 mAP in its native pipeline, while the code-generation pipeline only achieves 0.443. A comparable pattern is observed in the YOLOv8 family, where the YOLOv8m model records 0.539 mAP in the native pipeline versus 0.491 in code generation. These results reinforce the observation that native pipelines not only preserve but also enhance detection accuracy in embedded environments.

Additionally, regardless of the implementation strategy, significant variation is observed across models, suggesting that network size alone does not guarantee improved performance. For instance, models such as YOLOv9c and YOLOv8l, despite having more parameters than their smaller counterparts, do not necessarily yield better accuracy. This highlights the importance of considering architectural efficiency and domain generalization capabilities rather than model size alone when selecting neural networks for traffic light detection in embedded systems.

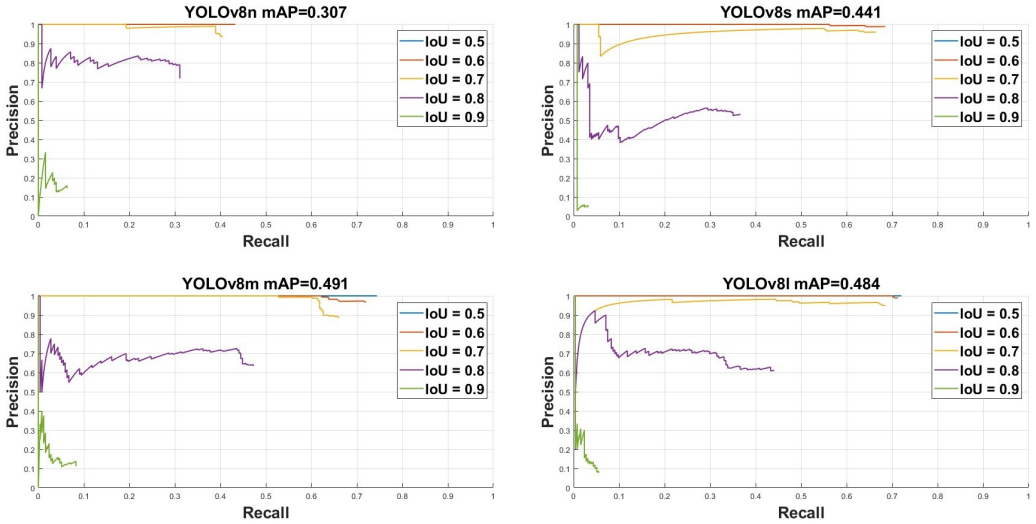

Figure 5. Precision–Recall curves of YOLOv8 code generation pipeline.

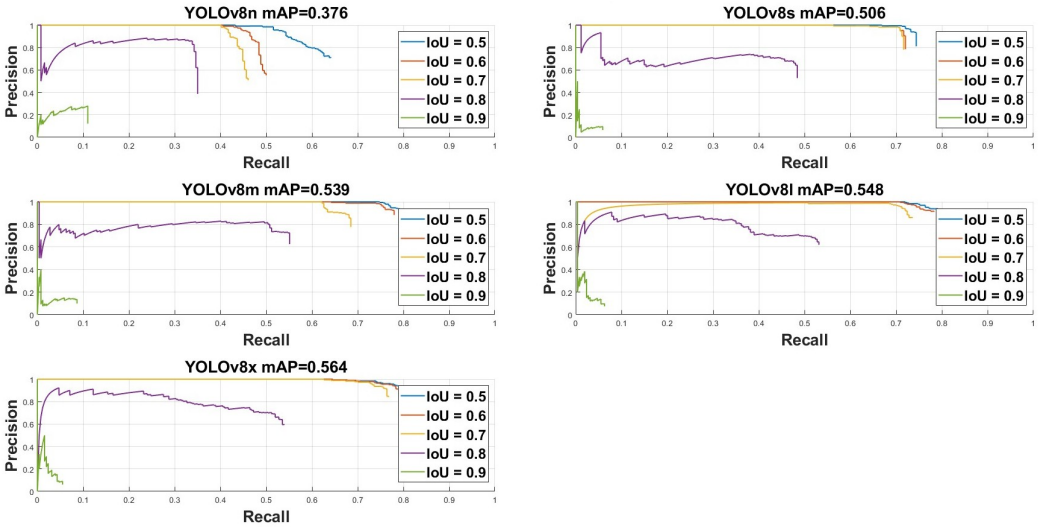

Figure 6. Precision–Recall curves of YOLOv8 native pipeline.

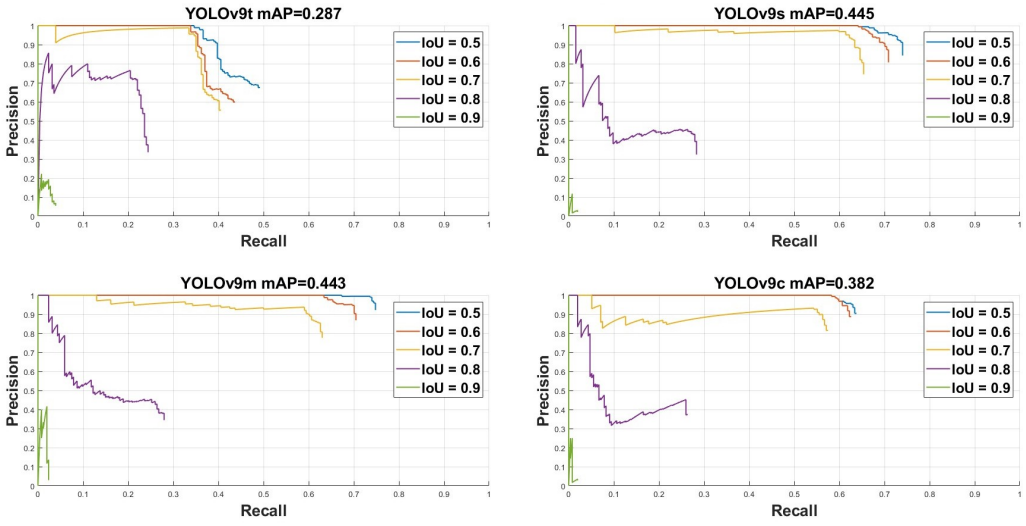

Figure 7. Precision–Recall curves of YOLOv9 code generation pipeline.

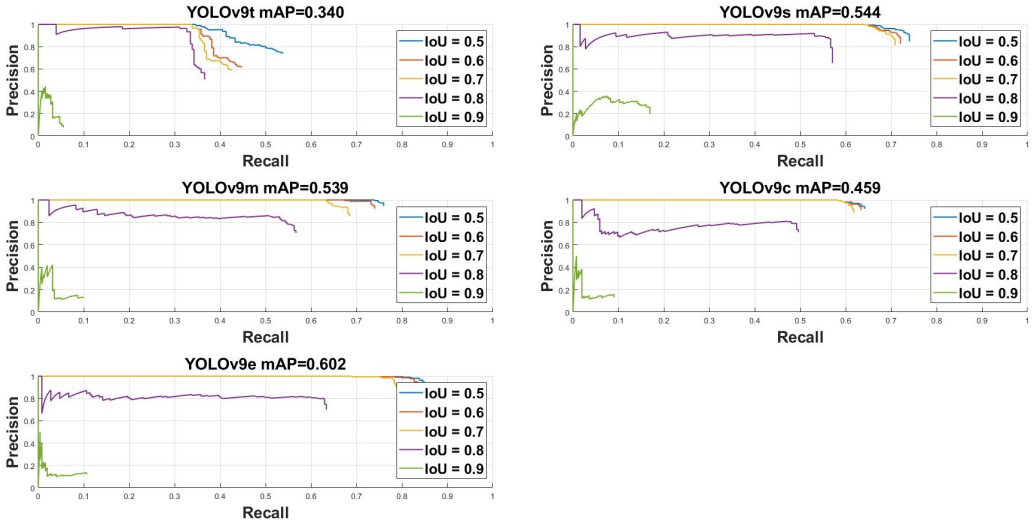

Figure 8. Precision–Recall curves of YOLOv9 native pipeline.

Tables 4 and 5 present the confusion matrices obtained for each type of implementation. These tables not only provide a detailed breakdown of classification performance across different IoU thresholds, but also support the Precision–Recall curves discussed earlier, due to the direct mathematical relationship between confusion matrix components and precision–recall metrics. The TP, FP, and FN values are computed using dataset-level aggregation. Detections are matched to ground-truth bounding boxes on a per-frame basis using one-to-one IoU-based assignment, and the final TP, FP, and FN counts are obtained by summing the results across all frames in the test sequence.

In our evaluation of traffic light detection using pre-trained YOLOv9 models on synthetic data generated from the CARLA simulator, we observed that YOLOv9c performs poorly compared to YOLOv9m, although it is a larger model. Specifically, YOLOv9c exhibits a higher rate of false negatives and a lower number of true positives, suggesting a reduced reliability in detecting traffic lights within this domain. These specific True Positive (TP) and False Negative (FN) values for YOLOv9c are highlighted in bold within the tables to emphasize the performance gap. This gap is likely attributed to a domain shift between the real-world COCO dataset on which the models were originally trained and the synthetic environment of CARLA. Larger models such as YOLOv9c are more prone to overfitting to the source domain, which can result in poor generalization when applied to unseen or synthetic data without additional fine-tuning. In particular, this issue does not arise with YOLOv9e, which demonstrates strong performance under the same conditions. This suggests that the limitation is model-specific rather than an inherent drawback of larger architectures.

A similar pattern is observed with the YOLOv8 models, where YOLOv8m and YOLOv8l exhibit comparable performance despite their architectural differences. As with YOLOv9c, YOLOv8l shows a slightly higher tendency toward false negatives, which is also highlighted in bold in the corresponding confusion matrix, supporting the hypothesis that increased model complexity does not necessarily translate to better generalization in synthetic environments when domain adaptation is not applied.

Moreover, the tables also highlight in bold the models with the best performance for each implementation strategy, YOLOv8x and YOLOv9e in native execution, and YOLOv8m and YOLOv9s in code generation. These highlights align with the mAP results previously discussed, further validating the observed trends in detection accuracy across different models and deployment approaches.

Table 4. Confusion matrix for the native pipeline YOLO models at different IoU thresholds. Boldface indicates the model variants selected as representative configurations for subsequent analysis. Abbreviations: IoU, intersection over union; TP, true positives; FP, false positives; FN, false negatives.

Model	IoU = 0.5			IoU = 0.6			IoU = 0.7			IoU = 0.8			IoU = 0.9		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Yolov9t	125	31	129	106	50	148	102	54	152	91	65	163	14	142	240
Yolov9s	188	18	66	183	23	71	180	26	74	145	61	109	43	163	211
Yolov9m	191	5	63	187	9	67	173	23	81	143	53	111	25	171	229
Yolov9c	**162**	**9**	**92**	**161**	**10**	**93**	**157**	**14**	**97**	**126**	**45**	**128**	**23**	**148**	**231**
Yolov9e	**213**	**8**	**41**	**210**	**11**	**44**	**200**	**21**	**54**	**161**	**60**	**93**	**27**	**194**	**227**
Yolov8n	150	36	104	124	62	130	115	71	139	89	97	165	28	158	226
Yolov8s	189	18	65	183	24	71	182	25	72	123	84	131	15	192	239
Yolov8m	197	11	57	194	14	60	174	34	80	140	68	114	22	186	232
Yolov8l	**196**	**12**	**58**	**193**	**15**	**61**	**184**	**24**	**70**	**134**	**74**	**120**	**16**	**192**	**238**
Yolov8x	**203**	**19**	**51**	**201**	**21**	**53**	**194**	**28**	**60**	**136**	**86**	**118**	**14**	**208**	**240**

Table 5. Confusion matrix for the code generation pipeline YOLO models at different IoU thresholds. Boldface indicates the model variants selected as representative configurations for subsequent analysis. Abbreviations: IoU, intersection over union; TP, true positives; FP, false positives; FN, false negatives.

Model	IoU = 0.5			IoU = 0.6			IoU = 0.7			IoU = 0.8			IoU = 0.9		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Yolov9t	115	43	139	103	55	151	98	60	156	61	97	193	10	148	244
Yolov9s	**188**	**18**	**66**	**180**	**26**	**74**	**166**	**40**	**88**	**72**	**134**	**182**	**5**	**201**	**249**
Yolov9m	189	7	65	178	18	76	159	37	95	71	125	183	6	190	248
Yolov9c	**161**	**11**	**93**	**158**	**14**	**96**	**145**	**27**	**109**	**66**	**106**	**188**	**5**	**167**	**249**
Yolov8n	110	0	144	110	0	144	103	7	151	79	31	175	16	94	238
Yolov8s	174	2	80	174	2	80	169	7	85	93	83	161	8	168	246
Yolov8m	**189**	**0**	**65**	**183**	**6**	**71**	**168**	**21**	**86**	**120**	**69**	**134**	**21**	**168**	**233**
Yolov8l	**183**	**0**	**71**	**181**	**2**	**73**	**174**	**9**	**80**	**112**	**71**	**142**	**14**	**169**	**240**

In addition to reporting performance across multiple IoU thresholds, Tables 6 and 7 provide mAP@0.5 (Pascal VOC standard) and mAP@0.5:0.95 (COCO-style evaluation) to facilitate comparison with existing object-detection benchmarks.

Since the task corresponds to single-class object detection, true negatives are not explicitly reported. Frames without annotated objects are included in the evaluation; any detections produced in such frames are counted as false positives. False negatives correspond to ground-truth objects that remain unmatched after one-to-one IoU-based assignment.

Table 6. Detection metrics for native pipeline.

Model	IoU = 0.5			mAP@0.5	mAP@0.5:0.95
	P	R	F1		
Yolov9t	0.8026	0.4961	0.6131	0.5136	0.3086
Yolov9s	0.9082	0.7402	0.8156	0.7368	0.5108
Yolov9m	0.9746	0.7559	0.8514	0.7595	0.5020
Yolov9c	0.9474	0.6378	0.7624	0.6402	0.4225
Yolov9e	0.9638	0.8386	0.8968	0.8488	0.5651
Yolov8n	0.8065	0.5906	0.6818	0.6210	0.3464
Yolov8s	0.9130	0.7441	0.8200	0.7427	0.4676
Yolov8m	0.9471	0.7756	0.8528	0.7972	0.5010
Yolov8l	0.9423	0.7717	0.8485	0.8000	0.5084
Yolov8x	0.9103	0.7992	0.8512	0.8132	0.5302

Table 7. Detection metrics for code generation pipeline.

Model	IoU = 0.5			mAP@0.5	mAP@0.5:0.95
	P	R	F1		
Yolov9t	0.7296	0.4567	0.5617	0.4641	0.2585
Yolov9s	0.9082	0.7402	0.8156	0.7368	0.4094
Yolov9m	0.9594	0.7441	0.8381	0.7474	0.4070
Yolov9c	0.9361	0.6339	0.7559	0.6358	0.3522
Yolov8n	1.0000	0.4331	0.6044	0.4331	0.2878
Yolov8s	0.9886	0.6850	0.8093	0.6840	0.4068
Yolov8m	1.0000	0.7441	0.8533	0.7441	0.4528
Yolov8l	1.0000	0.7205	0.8375	0.7205	0.4455

4.2. Computational Performance

Tables 8–11 show the inference time measurements for various YOLOv8 and YOLOv9 models executed on the NVIDIA Jetson Orin Nano and Jetson Orin AGX platforms, using both native and code generation pipelines. As expected, these benchmarks generally reflect scaling trends based on model size; however, several anomalies emerge, particularly in the native pipeline on the Jetson Orin AGX.

Table 8. Inference time for native pipeline in NVIDIA Jetson Orin Nano (mean latency with 95% CI over $n = 400$ frames).

Model	Avg Inference Time (ms)	95% CI Low (ms)	95% CI High (ms)	Max Inference Time (ms)	Max Time Iteration	Std Deviation (ms)	Init Time (ms)
Yolov9t	**64.28**	**63.33**	**65.23**	**215.87**	**24**	**9.71**	**2900.27**
Yolov9s	70.67	69.86	71.48	192.68	20	8.26	3063.99
Yolov9m	105.96	105.03	106.89	252.85	26	9.45	3544.03
Yolov9c	123.42	122.43	124.41	282.25	7	10.09	3869.52
Yolov9e	246.58	245.59	247.57	401.87	3	10.08	6122.36
Yolov8n	**36.64**	**35.71**	**37.57**	**173.43**	**29**	**9.46**	**2573.73**
Yolov8s	54.24	53.41	55.07	162.32	23	8.50	3141.68
Yolov8m	92.25	91.47	93.03	201.57	16	7.97	4243.88
Yolov8l	142.72	141.82	143.62	283.85	8	9.12	5674.63
Yolov8x	191.69	190.78	192.60	335.58	5	9.29	7558.80

Table 9. Inference time for code generation implementation in NVIDIA Jetson Orin Nano (mean latency with 95% CI over $n = 400$ frames).

Model	Avg Inference Time (ms)	95% CI Low (ms)	95% CI High (ms)	Max Inference Time (ms)	Max Time Iteration	Std Deviation (ms)	Init Time (ms)
Yolov9t	**63.00**	**62.03**	**63.97**	**89.21**	**89**	**9.87**	**138.10**
Yolov9s	149.28	145.88	152.68	213.05	85	34.57	229.44
Yolov9m	369.12	361.11	377.13	431.63	45	81.48	415.87
Yolov9c	616.23	601.42	631.04	727.17	43	150.58	513.44
Yolov8n	**79.94**	**77.55**	**82.33**	**122.47**	**142**	**24.29**	**123.13**
Yolov8s	197.66	193.76	201.56	233.80	42	39.71	213.77
Yolov8m	420.82	413.30	428.34	455.55	16	76.45	564.38
Yolov8l	730.50	710.59	750.41	858.67	20	202.48	689.86

Table 10. Inference time for native implementation in NVIDIA Jetson Orin AGX (mean latency with 95% CI over $n = 400$ frames).

Model	Avg Inference Time (ms)	95% CI Low (ms)	95% CI High (ms)	Max Inference Time (ms)	Max Time Iteration	Std Deviation (ms)	Init Time (ms)
Yolov9t	43.08	42.62	43.54	115.09	24	4.69	1753.63
Yolov9s	44.50	44.10	44.90	103.57	20	4.09	1853.36
Yolov9m	**35.94**	**35.48**	**36.40**	**106.13**	**26**	**4.66**	**2179.28**
Yolov9c	39.49	39.02	39.96	111.56	7	4.83	2334.73
Yolov9e	79.82	78.96	80.68	153.02	3	8.77	3487.42
Yolov8n	**20.96**	**20.49**	**21.43**	**91.19**	**29**	**4.73**	**1629.51**
Yolov8s	21.16	20.74	21.58	79.25	23	4.23	1877.52
Yolov8m	30.39	29.98	30.80	89.40	16	4.22	2306.58
Yolov8l	44.61	44.13	45.09	121.40	8	4.86	2855.56
Yolov8x	57.89	57.40	58.38	136.94	5	5.00	3817.16

Table 11. Inference time for code generation implementation in NVIDIA Jetson Orin AGX (mean latency with 95% CI over $n = 400$ frames).

Model	Avg Inference Time (ms)	95% CI Low (ms)	95% CI High (ms)	Max Inference Time (ms)	Max Time Iteration	Std Deviation (ms)	Init Time (ms)
Yolov9t	**19.94**	**19.53**	**20.35**	**33.59**	**132**	**4.20**	**78.75**
Yolov9s	36.96	36.50	37.42	46.56	96	4.66	97.97
Yolov9m	77.26	76.47	78.05	93.15	74	8.07	157.32
Yolov9c	102.30	101.28	103.32	112.87	105	10.38	186.76
Yolov8n	**20.71**	**20.16**	**21.26**	**32.28**	**124**	**5.58**	**62.04**
Yolov8s	40.26	39.60	40.92	54.05	54	6.69	89.00
Yolov8m	84.96	83.93	85.99	98.34	129	10.52	158.90
Yolov8l	151.50	149.87	153.13	189.51	55	16.59	238.77

A particularly unexpected observation involves YOLOv9m and YOLOv9c (highlighted in bold), which, when executed natively on the Jetson Orin AGX, achieve faster inference times than YOLOv9t and YOLOv9s, which, due to their smaller architectural size, had been expected to be faster. This suggests that those YOLO models may be better optimized to take advantage of specific architectural features of the AGX platform, such as memory bandwidth, tensor cores, or advanced scheduling mechanisms. It is plausible that their additional layers or structural complexity align more effectively with the GPU execution pipeline, resulting in more efficient utilization despite the increased parameter count. A similar behavior is observed within the YOLOv8 family, where YOLOv8n and YOLOv8s exhibit nearly identical inference times (both 21 ms) despite differences in model size and depth, reinforcing the idea that architectural synergy with hardware can outweigh raw model size.

The best-performing inference times for each model and implementation strategy are highlighted in bold within the tables, allowing for quick identification of optimal configurations. Among these, the fastest overall model is YOLOv9t using code generation, achieving an inference time of 19 ms. For native pipeline, the fastest models are YOLOv8n and YOLOv8s, both at 21 ms, with YOLOv8s showing unexpectedly high efficiency despite its larger architecture.

In contrast, the Jetson Orin Nano platform exhibits inference times that are unsuitable for real-time applications. For example, YOLOv9t in the code-generation pipeline runs approximately 315% slower on the Orin Nano compared to the same model on the Orin AGX. Similarly, YOLOv8n under native execution is approximately 180% slower than on the AGX platform. These figures underline the performance gap between the two hardware configurations and the challenges of deploying real-time detection systems on lower-end embedded devices.

Overall, these findings demonstrate that model size alone is not a reliable predictor of runtime performance on embedded hardware. Factors such as layer composition, operator fusion, memory access patterns, and backend-level optimizations all play a critical role in determining execution efficiency. Therefore, empirical benchmarking remains essential when selecting and deploying models on resource-constrained platforms such as the Jetson Orin family.

As shown in Tables 12–15, the native pipeline on Jetson Orin AGX consistently delivers the most favorable power–latency trade-off, achieving markedly lower mean inference times at only moderately higher average power than the Orin Nano. In particular, YOLOv8n on AGX (native) provides the strongest overall efficiency with 1.70 FPS/W at 20.96 ms, indicating that the additional power headroom of AGX is effectively converted into throughput. On the lower-power device, YOLOv8n on Nano (native) remains the best option (1.84 FPS/W at 36.64 ms), but its latency is substantially higher, reflecting tighter compute and memory constraints even in MAXN SUPER. For larger models, Nano approaches higher average power while latency increases sharply (e.g., YOLOv9e and YOLOv8x), which reduces energy efficiency.

Moreover, the tables show that the native implementation is systematically more power-efficient than the code-generation pipeline on both platforms. The code-generation approach sustains higher average power consumption while also increasing latency, most prominently on Nano (e.g., YOLOv8l), driving the lowest efficiencies. Therefore, for energy-aware real-time deployments, the preferred configuration is native inference with a lightweight model such as YOLOv8n (on AGX when minimum latency is required and on Nano when the power envelope is the primary constraint), whereas the code-generation pipeline becomes progressively less attractive as model size grows due to compounded latency and sustained power draw.

Table 12. Power and energy-efficiency metrics for the native pipeline on NVIDIA Jetson Orin Nano in MAXN SUPER mode.

Model	Avg Inference Time (ms)	P_{Avg} (W)	P_{peak} (W)	Efficiency (FPS/W)
Yolov9t	64.28	15.34	21.27	1.01
Yolov9s	70.67	15.11	21.69	0.94
Yolov9m	105.96	16.58	23.41	0.57
Yolov9c	123.42	17.06	24.08	0.47
Yolov9e	246.58	19.02	24.96	0.21
Yolov8n	36.64	14.83	20.74	1.84
Yolov8s	54.24	15.27	21.13	1.21
Yolov8m	92.25	16.04	22.39	0.68
Yolov8l	142.72	17.41	24.33	0.40
Yolov8x	191.69	18.11	24.71	0.29

Table 13. Power and energy-efficiency metrics for the code generation pipeline on NVIDIA Jetson Orin Nano in MAXN SUPER mode.

Model	Avg Inference Time (ms)	P_{Avg} (W)	P_{peak} (W)	Efficiency (FPS/W)
Yolov9t	63.00	18.24	23.62	0.87
Yolov9s	149.28	19.81	24.37	0.34
Yolov9m	369.12	21.94	24.88	0.12
Yolov9c	616.23	22.47	24.93	0.07
Yolov8n	79.94	18.63	24.05	0.67
Yolov8s	197.66	20.27	24.59	0.25
Yolov8m	420.82	21.71	24.84	0.11
Yolov8l	730.50	23.06	24.95	0.06

Table 14. Power and energy-efficiency metrics for the native pipeline on NVIDIA Jetson Orin AGX in 60 W mode.

Model	Avg Inference Time (ms)	P_{Avg} (W)	P_{peak} (W)	Efficiency (FPS/W)
Yolov9t	43.08	31.27	48.36	0.74
Yolov9s	44.50	31.94	47.85	0.70
Yolov9m	35.94	34.18	51.72	0.81
Yolov9c	39.49	33.62	50.41	0.75
Yolov9e	79.82	41.53	58.92	0.30
Yolov8n	20.96	28.11	43.79	1.70
Yolov8s	21.16	28.74	44.26	1.64
Yolov8m	30.39	31.88	48.97	1.03
Yolov8l	44.61	36.42	54.83	0.62
Yolov8x	57.89	38.67	56.91	0.45

Table 15. Power and energy-efficiency metrics for the code generation pipeline on NVIDIA Jetson Orin AGX in 60 W mode.

Model	Avg Inference Time (ms)	P_{Avg} (W)	P_{peak} (W)	Efficiency (FPS/W)
Yolov9t	19.94	29.63	44.12	1.69
Yolov9s	36.96	31.02	46.88	0.87
Yolov9m	77.26	35.24	52.33	0.37
Yolov9c	102.30	37.18	55.61	0.26
Yolov8n	20.71	29.18	43.57	1.65
Yolov8s	40.26	31.74	47.46	0.78
Yolov8m	84.96	35.91	53.04	0.33
Yolov8l	151.50	40.76	58.44	0.16

4.3. Comparative Analysis

Figures 9 and 10 provide a visual comparative analysis that combines the metrics discussed above, mAP, and inference time into a unified representation for each pipeline and hardware platform. These plots clearly highlight the overall superiority of the native pipeline over the code-generation one in terms of precision/performance trade-off. Interestingly, regardless of the implementation strategy, the results confirm that achieving higher detection accuracy generally requires more computational effort. This is particularly evident in the behavior of YOLOv8 and YOLOv9: while YOLOv8 achieves better performance in the native pipeline, it underperforms YOLOv9 under code generation.

Among all evaluated configurations, YOLOv8s (native pipeline) offers the best balance between precision and computational cost, while YOLOv9e (native pipeline) stands out as the most accurate model. Based on these findings, for the evaluated traffic-light detection micro-benchmark, we would recommend models such as YOLOv8s, YOLOv8m, or YOLOv9m under native execution on a high-performance platform like the Jetson Orin AGX.

The use of COCO-pretrained models without domain-specific fine-tuning may introduce domain shift effects when evaluated on synthetic CARLA data. Larger models may be more sensitive to such shifts, which could partially explain observed performance variations across architectures.

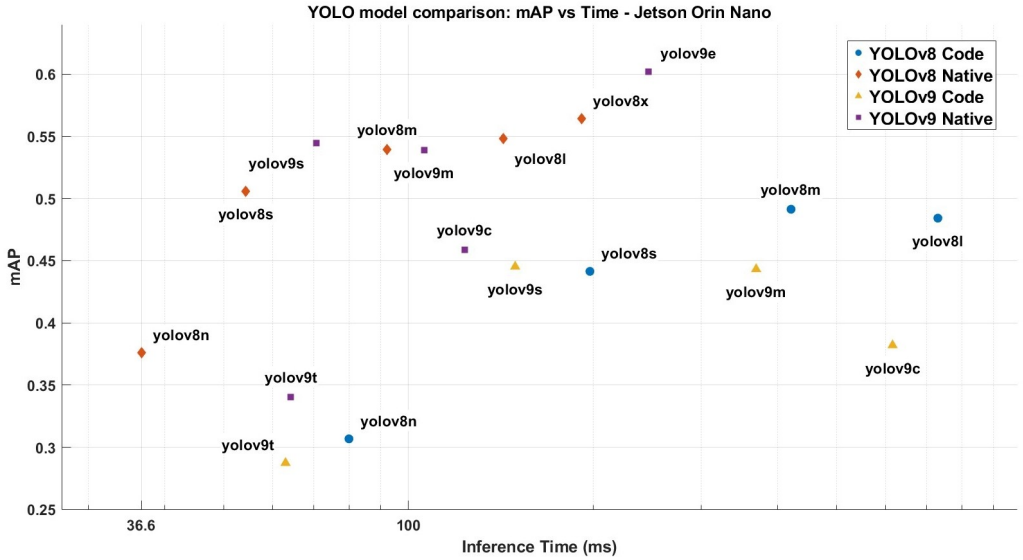

Figure 9. mAP-Inference time for NVIDIA Jetson Orin Nano.

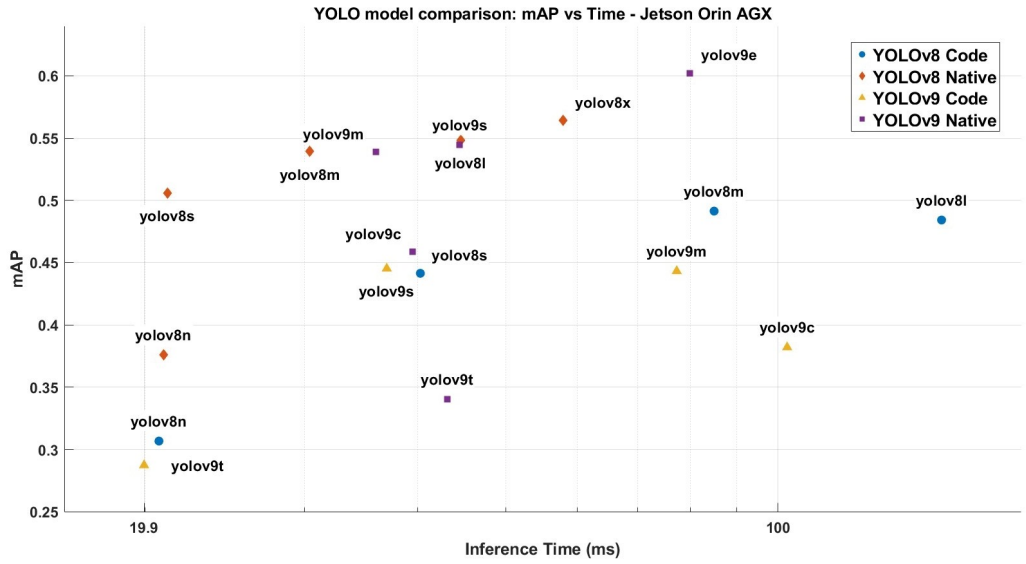

Figure 10. mAP-Inference time for NVIDIA Jetson Orin AGX.

4.4. Limitations

This study has faced several technical limitations that have influenced the scope of the evaluation and the applicability of certain models within the proposed workflow. One of the most relevant constraints has been the impossibility of evaluating the larger YOLOv8x and YOLOv9e models in embedded environments. This limitation is the result of a combination of factors. On the ond hand, prior tests conducted with smaller models such

as YOLOv8l and YOLOv9c already revealed inference times that were clearly unsuitable for real-time applications, especially when using code generation approaches. This led to the reasonable assumption that larger models such as YOLOv8x or YOLOv9e would perform even worse under the same conditions, thus offering limited practical value for the intended use case. On the other hand, in the specific case of YOLOv9e, the code generation tool used for converting the model into Simulink-compatible blocks was not capable of handling the network due to its size and internal complexity. This restriction comes from the third-party repository used for the integration of YOLOv9 into Simulink.

In addition to model-size constraints, the experimental design intentionally focuses on a controlled single-class scenario (CARLA-generated sequence, fixed input resolution, and a single object category) to isolate deployment-pipeline effects. As a result, the reported performance gap between native and code-generation pipelines should not be interpreted as a general robustness statement under diverse sensing conditions. In real deployments, absolute accuracy and latency may vary substantially with camera optics and calibration, illumination changes (day/night, glare), motion blur, sensor noise, occlusions, and multi-class scene composition, which can also alter post-processing load (e.g., NMS behavior) and end-to-end latency. These factors were not systematically varied in this work and are left for future extensions of the benchmark.

Finally, since the evaluated dataset is a short controlled sequence, it does not support statistical generalization across traffic-light designs, geographic regions, weather conditions, or sensor configurations. The primary objective is a reproducible pipeline-level comparison under fixed and documented conditions, rather than a comprehensive operational-design-domain validation.

5. Conclusions

This work quantifies the current gap between automatic code generation and native pipelines for object detection inference on embedded GPU platforms. Across YOLOv8/YOLOv9 workloads on Jetson Orin devices, the native pipeline consistently provides a better accuracy–latency trade-off, while the evaluated code-generation pipeline remains limited for larger models and shows poorer scaling with model complexity. To the best of our knowledge, relatively few studies have examined these deployment paradigms on real embedded hardware under unified and consistent evaluation criteria. These conclusions are restricted to the measured end-to-end pipelines and the stated experimental configuration (single-class traffic-light micro-benchmark, fixed input resolution, batch size 1, and default pre/post-processing settings).

In this context, we benchmarked multiple YOLOv8 and YOLOv9 model variants on Jetson Orin Nano and Jetson AGX Orin, reporting both detection-quality metrics and system-level latency measurements on a controlled single-class scenario, built from CARLA-generated data. The results highlight that runtime performance cannot be inferred from model size alone, and that the interaction between model variant, backend, and target platform can lead to non-intuitive behaviors that require further empirical validation.

From a practical perspective, our findings provide guidance for deployment decisions in real-time perception systems. For the evaluated setting, native pipeline of mid-sized models (e.g., YOLOv9s, YOLOv8m, and YOLOv9m) on Jetson AGX Orin offer the most promising balance between detection quality and computational cost. Since no domain-specific fine-tuning was performed, reported accuracy values reflect out-of-domain generalization behavior rather than optimized detection performance. Importantly, absolute accuracy and latency values should not be extrapolated to unconstrained driving conditions, where camera optics and calibration, illumination variability, motion blur, sensor noise, occlusions,

and multi-class scene composition can substantially change both detection difficulty and post-processing load (e.g., NMS behavior).

Although the MATLAB/Simulink toolchain provides ISO 26262 qualification support up to ASIL D under specific documented conditions, the perception pipeline evaluated in this study has not undergone functional safety certification. The results presented here correspond to a research-oriented benchmarking study and should not be interpreted as evidence of safety compliance for end-to-end deployment.

Finally, this study should be interpreted as a controlled traffic-light detection micro-benchmark rather than a comprehensive evaluation of full autonomous-driving or ADAS perception stacks. Broader claims would require multi-class datasets, diverse operational conditions (illumination, weather, sensor modalities), and validation on real-world data. Likewise, a broader assessment of robustness would require systematic variation of sensing conditions (e.g., optics, exposure, blur/noise) and scene complexity (e.g., multiple object categories and clutter) under a statistically grounded experimental design.

As a natural continuation of this work, several future research directions are proposed. First, a deeper analysis of the code generated by Simulink could offer valuable insights into the performance limitations observed on embedded hardware. By profiling the generated C/C++ or CUDA code, it would be possible to identify specific bottlenecks and assess the efficiency of automatic code generation in safety-critical real-time contexts.

Second, incorporating domain-specific fine-tuning on traffic-light datasets or CARLA-based annotations would allow us to quantify the impact of domain adaptation on detection performance and to evaluate whether the relative performance gap between native and code-generation pipelines persists under adapted weights. This would help distinguish between domain-shift effects and deployment-related limitations.

Third, recent work has explored machine-centric image processing techniques that prioritize information relevant for algorithmic perception rather than human visualization. For example, Wiseman [72] proposes an FDCT-based vision processing approach that reduces image data size while preserving features critical for robotic perception tasks. Integrating such machine-oriented compression or preprocessing strategies into the embedded YOLO pipeline could potentially reduce memory bandwidth usage, lower energy consumption, and improve inference latency. Evaluating the interaction between machine-centric visual encoding and deployment pipelines constitutes an interesting direction for future study.

Finally, extending the study to include additional neural network architectures or further YOLO variants would broaden the comparative analysis, enabling a more comprehensive evaluation of model–deployment interactions. Testing the proposed methodology on alternative embedded platforms would further allow assessment of portability and hardware-dependent behavior, strengthening the generality of the presented benchmarking framework.

Author Contributions: Conceptualization, M.H.M.; methodology, P.M.O., A.T. and M.H.M.; software, P.M.O.; validation, P.M.O. and A.T.; formal analysis, P.M.O. and A.T.; investigation, P.M.O., A.T. and M.H.M.; resources, A.T.; data curation, P.M.O.; writing—original draft preparation, P.M.O. and A.T.; writing—review and editing, P.M.O. and A.T.; supervision, A.T. and M.H.M.; project administration, A.T. and M.H.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding authors.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Uhrig, R. Introduction to artificial neural networks. In *Proceedings of IECON '95—21st Annual Conference on IEEE Industrial Electronics*; IEEE: Piscataway, NJ, USA, 1995; Volume 1, pp. 33–37. [CrossRef]
2. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 27–30 June 2016*; IEEE: Piscataway, NJ, USA, 2016.
3. Terven, J.; Córdova-Esparza, D.M.; Romero-González, J.A. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Mach. Learn. Knowl. Extr.* **2023**, *5*, 1680–1716. [CrossRef]
4. Sarhan, N.H.; Al-Omary, A.Y. Traffic light detection using OpenCV and YOLO. In *Proceedings of the 2022 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT);* IEEE: Piscataway, NJ, USA, 2022; pp. 604–608. [CrossRef]
5. Jensen, M.B.; Nasrollahi, K.; Moeslund, T.B. Evaluating state-of-the-art object detector on challenging traffic light data. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*; IEEE: Piscataway, NJ, USA, 2017; pp. 882–888. [CrossRef]
6. Liu, S.; Liu, L.; Tang, J.; Yu, B.; Wang, Y.; Shi, W. Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proc. IEEE* **2019**, *107*, 1697–1716. [CrossRef]
7. Batzolis, E.; Vrochidou, E.; Papakostas, G.A. Machine learning in embedded systems: Limitations, solutions and future challenges. In *Proceedings of the 2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*; IEEE: Piscataway, NJ, USA, 2023; pp. 345–350. [CrossRef]
8. Vargas, J.; Alsweiss, S.; Toker, O.; Razdan, R.; Santos, J. An Overview of Autonomous Vehicles Sensors and Their Vulnerability to Weather Conditions. *Sensors* **2021**, *21*, 5397. [CrossRef]
9. Zhang, Y.; Carballo, A.; Yang, H.; Takeda, K. Perception and sensing for autonomous vehicles under adverse weather conditions: A survey. *ISPRS J. Photogramm. Remote Sens.* **2023**, *196*, 146–177. [CrossRef]
10. Ignatious, H.A.; El-Sayed, H.; Khan, M. An overview of sensors in Autonomous Vehicles. *Procedia Comput. Sci.* **2022**, *198*, 736–741. [CrossRef]
11. Campbell, S.; O' Mahony, N.; Krpalkova, L.; Riordan, D.; Walsh, J.; Murphy, A.; Ryan, C. Sensor technology in autonomous vehicles: A review. In *Proceedings of the 2018 29th Irish Signals and Systems Conference (ISSC), Belfast, UK, 21–22 June 2018*; IEEE: Piscataway, NJ, USA, 2018; pp. 1–4. [CrossRef]
12. Ayala, R.; Khan Mohd, T. Sensors in Autonomous Vehicles: A Survey. *J. Auton. Veh. Syst.* **2021**, *1*, 031003. [CrossRef]
13. Yang, L.; He, Z.; Zhao, X.; Fang, S.; Yuan, J.; He, Y.; Li, S.; Liu, S. A Deep Learning Method for Traffic Light Status Recognition. *J. Intell. Connect. Veh.* **2023**, *6*, 173–182. [CrossRef]
14. Behrendt, K.; Novak, L.; Botros, R. A deep learning approach to traffic lights: Detection, tracking, and classification. In *Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA)*; IEEE: Piscataway, NJ, USA, 2017; pp. 1370–1377. [CrossRef]
15. Kulkarni, R.; Dhavalikar, S.; Bangar, S. Traffic light detection and recognition for self driving cars using deep learning. In *Proceedings of the 2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*; IEEE: Piscataway, NJ, USA, 2018; pp. 1–4. [CrossRef]
16. John, V.; Yoneda, K.; Qi, B.; Liu, Z.; Mita, S. Traffic light recognition in varying illumination using deep learning and saliency map. In *Proceedings of the 17th International IEEE Conference on Intelligent Transportation Systems (ITSC);* IEEE: Piscataway, NJ, USA, 2014; pp. 2286–2291. [CrossRef]
17. Wang, J.G.; Zhou, L.B. Traffic Light Recognition With High Dynamic Range Imaging and Deep Learning. *IEEE Trans. Intell. Transp. Syst.* **2019**, *20*, 1341–1352. [CrossRef]
18. Possatti, L.C.; Guidolini, R.; Cardoso, V.B.; Berriel, R.F.; Paixão, T.M.; Badue, C.; De Souza, A.F.; Oliveira-Santos, T. Traffic light recognition using deep learning and prior maps for autonomous cars. In *Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN)*; IEEE: Piscataway, NJ, USA, 2019; pp. 1–8. [CrossRef]
19. Ouyang, Z.; Niu, J.; Liu, Y.; Guizani, M. Deep CNN-Based Real-Time Traffic Light Detector for Self-Driving Vehicles. *IEEE Trans. Mob. Comput.* **2020**, *19*, 300–313. [CrossRef]
20. Wan, M.; Han, M.; Li, L.; Li, Z.; He, S. Effects of and defenses against adversarial attacks on a traffic light classification CNN. In *Proceedings of the 2020 ACM Southeast Conference*; ACM: New York, NY, USA, 2020; pp. 94–99. [CrossRef]
21. Fredj, H.B.; Chabbah, A.; Baili, J.; Faiedh, H.; Souani, C. An efficient implementation of traffic signs recognition system using CNN. *Microprocess. Microsyst.* **2023**, *98*, 104791. [CrossRef]
22. Flores-Calero, M.; Astudillo, C.A.; Guevara, D.; Maza, J.; Lita, B.S.; Defaz, B.; Ante, J.S.; Zabala-Blanco, D.; Armingol Moreno, J.M. Traffic Sign Detection and Recognition Using YOLO Object Detection Algorithm: A Systematic Review. *Mathematics* **2024**, *12*, 297. [CrossRef]

23. Rangari, A.P.; Chouthmol, A.R.; Kadadas, C.; Pal, P.; Kumar Singh, S. Deep learning based smart traffic light system using image processing with YOLO v7. In *Proceedings of the 2022 4th International Conference on Circuits, Control, Communication and Computing (I4C)*; IEEE: Piscataway, NJ, USA, 2022; pp. 129–132. [CrossRef]
24. Müller, J.; Dietmayer, K. Detecting traffic lights by single shot detection. In *Proceedings of the 2018 21st International Conference on Intelligent Transportation Systems (ITSC)*; IEEE: Piscataway, NJ, USA, 2018; pp. 266–273. [CrossRef]
25. Naimi, H.; Akilan, T.; Khalid, M.A. Fast traffic sign and light detection using deep learning for automotive applications. In *Proceedings of the 2021 IEEE Western New York Image and Signal Processing Workshop (WNYISPW)*; IEEE: Piscataway, NJ, USA, 2021; pp. 1–5. [CrossRef]
26. Gavrilescu, R.; Zet, C.; Foşalău, C.; Skoczylas, M.; Cotovanu, D. Faster R-CNN: An approach to real-time object detection. In *Proceedings of the 2018 International Conference and Exposition on Electrical And Power Engineering (EPE)*; IEEE: Piscataway, NJ, USA, 2018; pp. 165–168. [CrossRef]
27. Wu, L.; Li, H.; He, J.; Chen, X. Traffic sign detection method based on Faster R-CNN. *J. Phys. Conf. Ser.* **2019**, *1176*, 032045. [CrossRef]
28. Alhashmi, S.A.; Al-azawi, A. A Review of the Single-Stage vs. Two-Stage Detectors Algorithm: Comprehensive Insights into Object Detection. *Int. J. Environ. Sci.* **2025**, *11*, 775–787.
29. Shine, L.; Jiji, C.V. Comparative analysis of two stage and single stage detectors for anomaly detection. In *Proceedings of the 2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*; IEEE: Piscataway, NJ, USA, 2021; pp. 1–6. [CrossRef]
30. Bilous, N.; Malko, V.; Frohme, M.; Nechyporenko, A. Comparison of CNN-Based Architectures for Detection of Different Object Classes. *AI* **2024**, *5*, 2300–2320. [CrossRef]
31. Shobaki, W.A.; Milanova, M. A Comparative Study of YOLO, SSD, Faster R-CNN, and More for Optimized Eye-Gaze Writing. *Sci* **2025**, *7*. [CrossRef]
32. Terven, J.; Cordova-Esparza, D. A comprehensive review of YOLO architectures in computer vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *arXiv* **2023**, arXiv:2304.00501. [CrossRef]
33. Ren, S.; He, K.; Girshick, R.; Sun, J. Faster R-CNN: Towards real-time object detection with region proposal networks. *arXiv* **2015**, arXiv:1506.01497. [CrossRef]
34. Carranza-García, M.; Torres-Mateo, J.; Lara-Benítez, P.; García-Gutiérrez, J. On the Performance of One-Stage and Two-Stage Object Detectors in Autonomous Vehicles Using Camera Data. *Remote Sens.* **2021**, *13*, 89. [CrossRef]
35. Jiang, P.; Ergu, D.; Liu, F.; Cai, Y.; Ma, B. A Review of Yolo Algorithm Developments. *Procedia Comput. Sci.* **2022**, *199*, 1066–1073. [CrossRef]
36. Lavanya, G.; Pande, S. Enhancing Real-time Object Detection with YOLO Algorithm. *EAI Endorsed Trans. Internet Things* **2023**, 10, 12. [CrossRef]
37. Hussain, M. YOLO-v1 to YOLO-v8, the Rise of YOLO and Its Complementary Nature toward Digital Manufacturing and Industrial Defect Detection. *Machines* **2023**, *11*, 677. [CrossRef]
38. Sundaresan Geetha, A.; Alif, M.A.R.; Hussain, M.; Allen, P. Comparative Analysis of YOLOv8 and YOLOv10 in Vehicle Detection: Performance Metrics and Model Efficacy. *Vehicles* **2024**, *6*, 1364–1382. [CrossRef]
39. Alahdal, N.M.; Abukhodair, F.; Meftah, L.H.; Cherif, A. Real-time Object Detection in Autonomous Vehicles with YOLO. *Procedia Comput. Sci.* **2024**, *246*, 2792–2801. [CrossRef]
40. Azevedo, P.; Santos, V. Comparative analysis of multiple YOLO-based target detectors and trackers for ADAS in edge devices. *Robot. Auton. Syst.* **2024**, *171*, 104558. [CrossRef]
41. Quang, L.T.; Truong, T.V.; Duong Ly, P.; Nhat, P.L.; Dang, L.T. A comparative analysis of YOLOv5 and YOLOv7 object detecting models for speed-limit traffic-sign recognition. *J. Phys. Conf. Ser.* **2025**, *2949*, 012022. [CrossRef]
42. Biglari, A.; Tang, W. A Review of Embedded Machine Learning Based on Hardware, Application, and Sensing Scheme. *Sensors* **2023**, *23*, 2131. [CrossRef]
43. Malhotra, K.; Kumar, Y. Challenges to implement machine learning in embedded systems. In *Proceedings of the 2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*; IEEE: Piscataway, NJ, USA, 2020; pp. 477–481. [CrossRef]
44. Ajani, T.S.; Imoize, A.L.; Atayero, A.A. An Overview of Machine Learning within Embedded and Mobile Devices–Optimizations and Applications. *Sensors* **2021**, *21*, 4412. [CrossRef]
45. Rey, L.; Bernardos, A.M.; Dobrzycki, A.D.; Carramiñana, D.; Bergesio, L.; Besada, J.A.; Casar, J.R. A performance analysis of you only look once models for deployment on constrained computational edge devices in drone applications. *arXiv* **2025**, arXiv:2502.15737. [CrossRef]

46. Gündüz, M.Ş.; Işık, G. A new YOLO-based method for real-time crowd detection from video and performance analysis of YOLO models. *J. Real-Time Image Process.* **2023**, *20*, 5. [CrossRef]
47. Chen, Z.; Yang, J.; Li, F.; Feng, Z.; Chen, L.; Jia, L.; Li, P. Foreign Object Detection Method for Railway Catenary Based on a Scarce Image Generation Model and Lightweight Perception Architecture. *IEEE Trans. Circuits Syst. Video Technol.* **2026**, *36*, 1377–1391. [CrossRef]
48. Liu, Z.; Wang, J.; Wu, H.; Xue, F.; Qin, Z.; Sun, S.; Guo, X.; Zhao, F. Water-aware real-time detection of floating plastic debris via an enhanced YOLOv13 framework for aquatic pollution monitoring. *Expert Syst. Appl.* **2026**, *313*, 131552. [CrossRef]
49. Li, H.; Zhao, F.; Xue, F.; Wang, J.; Liu, Y.; Chen, Y.; Wu, Q.; Tao, J.; Zhang, G.; Xi, D.; et al. Succulent-YOLO: Smart UAV-Assisted Succulent Farmland Monitoring with CLIP-Based YOLOv10 and Mamba Computer Vision. *Remote Sens.* **2025**, *17*, 2219. [CrossRef]
50. Wang, H.; Yang, Z.; Liu, Q.; Zhang, Q.; Wang, H. TCE-YOLOv5: Lightweight Automatic Driving Object Detection Algorithm Based on YOLOv5. *Appl. Sci.* **2025**, *15*, 6018. [CrossRef]
51. Shlezinger, N.; Whang, J.; Eldar, Y.C.; Dimakis, A.G. Model-Based Deep Learning. *Proc. IEEE* **2023**, *111*, 465–499. [CrossRef]
52. Nikhade, Y.; Patil, C.; Patel, M.; Pande, A. Manual vs. Automated Coding: A Comparative Analysis of Productivity and Code Quality. *SSRN Electron. J.* **2025**. [CrossRef]
53. Ultralytics. YOLO v8 Model Documentation. 2025. Available online: https://docs.ultralytics.com/es/models/yolov8/ (accessed on 18 July 2025).
54. Ultralytics. YOLO v9 Model Documentation. 2025. Available online: https://docs.ultralytics.com/es/models/yolov9/ (accessed on 18 July 2025).
55. CARLA Simulator Community. CARLA: An Open Urban Driving Simulator. 2025. Available online: https://carla.org/ (accessed on 14 July 2025).
56. Padilla, R.; Netto, S.L.; da Silva, E.A.B. A survey on performance metrics for object-detection algorithms. In *Proceedings of the 2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*; IEEE: Piscataway, NJ, USA, 2020; pp. 237–242. [CrossRef]
57. The MathWorks, Inc. Simulink—Simulation and Model-Based Design. 2025. Available online: https://in.mathworks.com/products/simulink.html (accessed on 7 July 2025).
58. The MathWorks, Inc. ISO 26262 Support in MATLAB and Simulink. 2025. Available online: https://www.mathworks.com/solutions/automotive/standards/iso-26262.html (accessed on 7 July 2025).
59. Open Source Robotics Foundation. Robot Operating System (ROS). 2025. Available online: https://www.ros.org/ (accessed on 7 July 2025).
60. Šarić, M. Robot operating system (ROS). In *Studies in Computational Intelligence*; Springer: Berlin/Heidelberg, Germany, 2016. [CrossRef]
61. Maruyama, Y.; Kato, S.; Azumi, T. Exploring the performance of ROS2. In *Proceedings of the 13th International Conference on Embedded Software*; EMSOFT '16; ACM: New York, NY, USA, 2016. [CrossRef]
62. Lin, T.Y.; Maire, M.; Belongie, S.; Bourdev, L.; Girshick, R.; Hays, J.; Perona, P.; Ramanan, D.; Zitnick, C.L.; Dollár, P. Microsoft COCO: Common objects in context. *arXiv* **2014**, arXiv:1405.0312.
63. The MathWorks, Inc. Pretrained YOLOv8 Network for Object Detection. 2025. Available online: https://github.com/matlab-deep-learning/Pretrained-YOLOv8-Network-For-Object-Detection/tree/main (accessed on 14 July 2025).
64. The MathWorks, Inc. Pretrained YOLOv9 Network for Object Detection. 2025. Available online: https://github.com/matlab-deep-learning/Pretrained-Yolov9-Network-For-Object-Detection (accessed on 14 July 2025).
65. Canonical Ltd. Ubuntu Releases—Jammy Jellyfish (22.04 LTS). 2025. Available online: https://releases.ubuntu.com/jammy/ (accessed on 16 July 2025).
66. NVIDIA Corporation. JetPack SDK—NVIDIA Embedded Software. 2025. Available online: https://developer.nvidia.com/embedded/jetpack (accessed on 16 July 2025).
67. NVIDIA Corporation. Jetson Orin—Embedded Systems for Autonomous Machines. 2025. Available online: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/ (accessed on 16 July 2025).
68. PyTorch Foundation. PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration. Available online: https://pytorch.org/ (accessed on 18 July 2025).
69. Segura, M.; Poggi, T.; Bárcena, R. A generic interface for x-in-the-loop simulations based on distributed co-simulation protocol. *IEEE Access* **2023**, *11*, 5578–5595. [CrossRef]
70. Bruggner, D.; Hegde, A.; Acerbo, F.S.; Gulati, D.; Son, T.D. Model in the loop testing and validation of embedded autonomous driving algorithms. In *Proceedings of the 2021 IEEE Intelligent Vehicles Symposium (IV)*; IEEE: Piscataway, NJ, USA, 2021; pp. 136–141. [CrossRef]

71. Sarhadi, P.; Yousefpour, S. State of the art: Hardware in the loop modeling and simulation with its applications in design, development and implementation of system and control software. *Int. J. Dyn. Control* **2015**, *3*, 470–479. [CrossRef]
72. Wiseman, Y. Achieving Robotic Data Efficiency Through Machine-Centric FDCT Vision Processing. *Sensors* **2026**, *26*, 518. [CrossRef]