Wi-Fi Enabled Edge Intelligence Framework for Smart City Traffic Monitoring using Low-Power IoT Cameras

Raphael Walcher
{rawalcher}@edu.aau.at
University of Klagenfurt
Institute of Information Technology
Klagenfurt, Austria

Kurt Horvath
{kurt}.{horvath}@aau.at
University of Klagenfurt
Institute of Information Technology
Klagenfurt, Austria

Dragi Kimovski
{dragi}.{kimovski}@aau.at
University of Klagenfurt
Institute of Information Technology
Klagenfurt, Austria

Stojan Kitanov
{stojan}.{kitanov}@unt.edu.mk
Mother Teresa University
Faculty of Information Sciences
Skopje, North Macedonia

ABSTRACT

Real-time traffic monitoring in smart cities demands ultra-low latency processing to support time-critical decisions such as incident detection and congestion management. While cloud-based solutions offer robust computation, their inherent latency limits their applicability for such tasks. This work proposes a localized edge AI framework that connects low-power IoT camera sensors to a client, or applies offloading of inference to an NVIDIA Jetson Nano (GPU). Networking is achieved via Wi-Fi, enabling image classification without relying on wide-area infrastructure such as 5G, or wired networks. We evaluate two processing strategies: local inference on camera nodes and GPU-accelerated offloading to the Jetson Nano. We show that local processing is only feasible for lightweight models and low frame rates, whereas offloading enables near-real-time performance even for more complex models. These results demonstrate the viability of cost-effective, Wi-Fi-based edge AI deployments for latency-critical urban monitoring.

CCS CONCEPTS

• **Networks** → **Network performance evaluation**; *Network structure*; *Location based services*; • **Computing methodologies** → Distributed computing methodologies.

KEYWORDS

IoT Services, Computing Continuum, Edge AI, Smart City

1 INTRODUCTION

Smart cities function as intelligent systems designed to enhance quality of life by addressing key areas such as mobility and traffic monitoring [1], as well as energy optimization and waste management. Among these applications of smart cities, real-time traffic analysis is central in enabling timely decision-making [2] [19] for incident detection, congestion mitigation, and adaptive signal control. Such applications require ultra-low latency and a large amount of compute resources to be effective [10]. While traditional cloud-based solutions ensure substantial computational power, their inherent network latency and limited Quality of Service (QoS) guarantees pose significant challenges for supporting latency-critical and real-time smart city applications.

Emerging paradigms in edge computing and artificial intelligence (AI) address this challenge by bringing computation closer to the data source [13], thereby reducing latency and bandwidth consumption. However, the deployment of edge systems often assumes the availability of low-latency communication infrastructure such as 5G or fiber networks. These assumptions do not hold universally, particularly in areas with limited coverage or with dense population [10]. Wi-Fi can provide an alternative in densely populated areas [11]. Furthermore, many urban monitoring (and smart city) needs are inherently local, targeting specific intersections, pedestrian zones, or parking facilities. In such cases, wide-area connectivity may be unnecessary or even counterproductive.

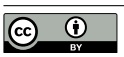
This work proposes a localized edge AI framework that uses Wi-Fi as a primary communication technology to connect **low-power IoT camera sensors** with shared edge devices to offload AI inference on vehicle classification.

Each camera node captures high-resolution images and processes them locally using its CPU or offloads them to Nvidia Jetson Nano nodes for GPU-accelerated inference. The design prioritizes nearby, high-performance processing within the coverage range of a Wi-Fi network, eliminating reliance on wide-area infrastructure.

The main contributions of this work are:

- Design and implementation of a Wi-Fi–based edge AI architecture that connects camera sensor nodes to edge devices for real-time image classification.
- Evaluation of the feasibility of local versus offloaded AI inference for vehicle classification in smart cities.
- Quantification of trade-offs among processing performance, dropped frames, and inference latency in single-node deployments.

This paper is organized as follows: Section 2 reviews related work on wireless technologies for smart cities, highlighting trade-offs between 5G and Wi-Fi in urban deployments. Section 3 presents our system's architecture and experimental setup. Section 4 defines the performance metrics used in our evaluation. Section 5 reports and discusses the results of our experiments. Section 6 discusses



outcomes and describes challenges still needing to be adressed. Section 7 concludes the paper.

2 RELATED WORK

Understanding the general characteristics of the edge communication and compute infrastructure is essential for supporting smart city applications. Therefore, the work of Chen et al. [4] contributes to understanding the role of Mobile Edge Computing (MEC) in enabling smarter cities. The authors highlight MEC's ability to support latency-sensitive services by placing computation close to the data source, emphasizing leveraging 5G networks to achieve the required bandwidth and low latency. Their architecture envisions MEC nodes at the network edge, enabling applications such as real-time video analytics, augmented reality, and vehicular communication. However, the reliance on 5G infrastructure limits its immediate applicability in areas without widespread 5G coverage or where deployment costs are prohibitive.

Similarly, Garcia et al. [6] analyze wireless technology selection for IoT deployments in smart cities, focusing on how technology choice impacts reliability and performance. They identify overcrowding in the 2.4 GHz ISM band due to the widespread use of Bluetooth, ZigBee, Wi-Fi, mobile communications, beeing a challenge for Worldwide Interoperability for Microwave Access (WiMax), causing interference, packet loss, and degraded performance. Their findings suggest that technology selection must balance application-specific requirements with interference mitigation, making Wi-Fi a viable but context-dependent alternative to cellular-based solutions.

The work of Taleb et al. [21] provides a comprehensive review of smart city architectures, identifying wireless technologies as key enablers of urban IoT services. They recognize both 5G and Wi-Fi as critical in urban deployments, noting that while 5G excels in wide-area, high-mobility scenarios, Wi-Fi offers cost-effective coverage for ultra-local services where infrastructure investment must remain minimal.

Together, these studies emphasize that although 5G promises high performance for large-scale and mobile smart city services, Wi-Fi remains a practical alternative in local or budget-constrained scenarios. Reflecting on the findings by Garcia [6] from 2018, shifting Wi-Fi into the 5/6 Ghz band could resolve many of the problems mentioned. This motivates our exploration of a Wi-Fi-based edge AI framework for real-time traffic monitoring, removing the dependency on wide-area infrastructure while meeting low-latency requirements.

3 FRAMEWORK

In this section, we introduce the framework of a Smart City use case application for performing vehicle classification directly at the roadside, proposing a low-power, low-cost solution, and propagating the information over short range to client applications via Wi-Fi.

As depicted in Figure 1, the architecture of the framework consists of three main components, client, local processing node, and remote inference node, deployed in two different configurations optimized for different objectives, namely **local** and **offloading**. The first deployment, referred to as the **local deployment**, involves a node performing vehicle classification directly on the roadside using a Raspberry Pi 4, which also acquires the input images. This setup uses CPU-only inference to achieve entirely local processing. A client device connects to the local node via Wi-Fi and requests image classification results at regular intervals. The results include annotated vehicle classifications in JSON format and image overlays (see Section 3.2), providing the most recent outcomes to the client applications.

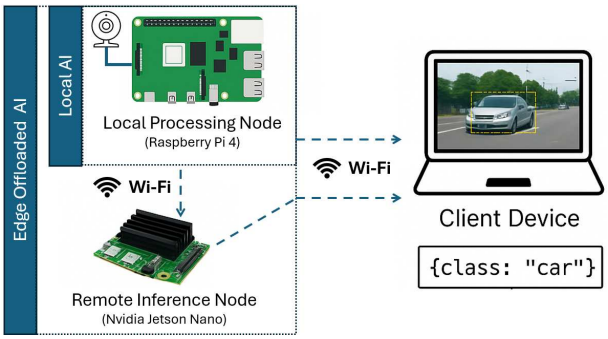


Figure 1: Overview of the system architecture

In the **offloading deployment**, inference is performed remotely on an edge node equipped with a GPU (NVIDIA Jetson Nano) for acceleration. When the client requests it, the local node sends the current image to the remote node, executing the image classification. After processing, the client device retrieves the results from the remote node.

This setup directly compares local (CPU-based) and offloaded (GPU-accelerated) edge processing, quantifying the impact of CPU and GPU inference performance in conjunction with additional networking latency introduced by the offloading process.

3.1 Workflow

This section describes how the three main components interact in the depicted deployment configurations. Based on the deployment configuration, the local processing node conducts image acquisition and inference, where the remote inference node only conducts inference on images provided by the local processing node. The components organize their communication through the following messages:

- **Hello(DeviceId)**: Announces device presence to the client
- **Control**: Comprises multiple subtypes:
  - *ConfigureExperiment*: Declares model, operational mode, and duration of the experiment
  - *ReadyToStart*: Confirms model loaded and warm-up completed
  - *BeginExperiment*: Initiates periodic pulsing
  - *Shutdown*: Performs orderly teardown
- **Pulse(TimingMetadata)**: Emitted by client every Δt
- **Frame**: Conveys image F_i from local to remote node
- **Result**: Returns inference output R_i to client

Both **Frame** and **Result** messages carry the `TimingMetadata` package initially sent by the **Pulse**. This package contains the sequence number i of the current inference cycle, the frame number, and a set of timestamps collected across devices.

The use of individual messages is described in the following sections.

https://doi.org/10.34749/3061-1008.2025.7

3.1.1 Workflow local inference. is depicted in Figure 2 and initiated by the client in a *Setup*-phase with the local processing node. In the *Setup*-phase, the client awaits `Hello` messages from the local processing node, then issues `Control::ConfigureExperiment` to configure the model, in the current deployment (local or offloading), and defines run parameters (e.g., experiment duration, fixed frame rate). The local processing node loads the model, performs a warm-up cycle, loading the model into memory and executing a dummy inference pass to prepare the runtime, and then replies to the client with `Control::ReadyToStart`.

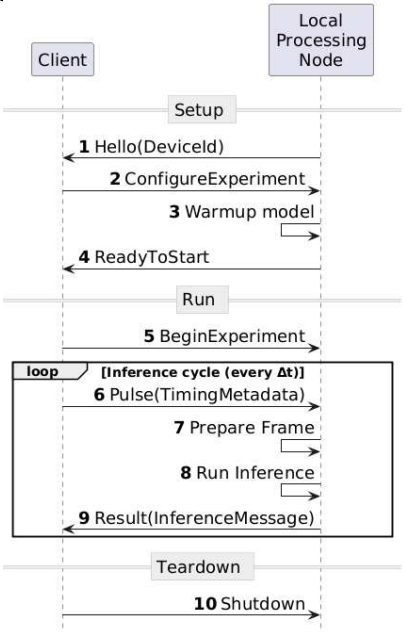


Figure 2: workflow on local deployment configuration using local inference

The *Run*-phase starts with the `Control::BeginExperiment` message indicating a new run. For each inference cycle, the client emits a `Pulse` message carrying a `TimingMetadata` package that includes the current sequence number i. Upon receiving a `Pulse`, the local processing node captures image F_i from the onboard camera. The system packs the captured frame into a `Frame`(F_i) message.Upon capturing, the node applies inference and returns the result to the client as `Result`(R_i). The client completes the workflow in the *Teardown* phase by submitting the `Shutdown` message.

3.1.2 Workflow offloaded inference. is depicted in Figure 3, and extends on the **local inference deployment**. During the *Setup*-phase, the client also registers the remote inference node by awaiting `Hello` and then sending the `Control::ConfigureExperiment` message. Both nodes respond with `Control::ReadyToStart`.

During the *Run*-phase, the local processing node transmits `Frame`(F_i) to the remote inference node. The remote inference node executes the model to obtain the inference result R_i. The remote node then returns `Result`(R_i) to the client. This deployment leverages GPU acceleration at the cost of network transmission overhead, such that the total processing time includes inference and data transfer delays.

The run continues until the configured experiment duration has elapsed, after which the client initiates the *Teardown*-phase by sending `Control::Shutdown` to both nodes, thus terminating the experiment.

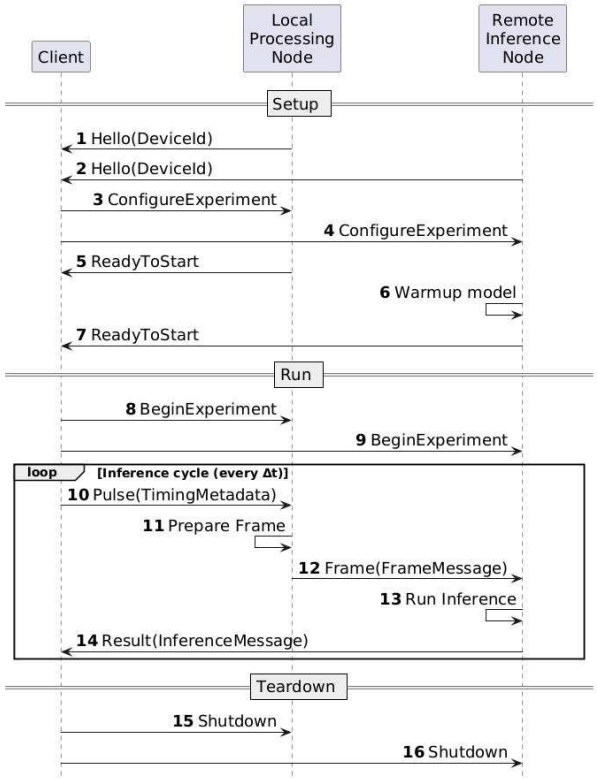


Figure 3: workflow on offloading deployment configuration applying remote inference

3.1.3 Distributed offloading. The proposed architecture is not necessarily constrained to a bijective relationship between the local processing node N^{loc} and remote inference node N^{rem}. Generalizing the queueing concept N^{rem} to support $1:k$ node relations, where k is the number of $N^{loc} = \{n_0 \dots n_{k-1}\}$ share one N^{rem} node for offloading.

With the capacity queue policy ($\forall n \in N^{loc} : Q_{\text{size}} = 1$), each local processing node supports only the most current frame to be processed.

We must demand the following condition to assign $n_i \in N^{loc}$.

$$\frac{\mu_m}{\sum_{i=0}^{k} \lambda_0(n_i)} \geq 1,$$

where μ_m is the highest sustainable processing rate for model m and λ_0 defines per-node frame generation rate.

3.2 Vehicle Classification using AI

In this section, we describe the purpose of the framework. Many traffic applications widely use these existing AI models for live traffic classification [20] [23] [16]. We also utilize *YOLOv5* [12] as object-detector. The models are trained on the COCO dataset [14, 24] and on car, truck, bus, and motorcycle categories.

https://doi.org/10.34749/3061-1008.2025.7

Three model distinguished by their size: `YOLOv5n` (nano), `YOLOv5s` (small), and `YOLOv5m` (medium). `YOLOv5n` variant provides the lowest latency but reduced precision, while `YOLOv5m` achieves higher accuracy at greater computational cost.

This selection enables systematic evaluation on both the *Local Processing Node* (CPU-only) and the *Remote Inference Node* (GPU-accelerated), showing a trade-off between processing locality. In terms of model size, their parameter counts and corresponding in-memory weights differ substantially, about *7 MB* for the nano-size model, *29 MB* for the medium, and *85 MB* at FP32 precision. The official YOLOv5 model table [12] reports the parameter counts that we use to derive these values, applying the relation of params × 4 bytes for FP32 storage.

Figure 4 compares the outputs of `YOLOv5n` and `YOLOv5m` on identical frames. Both models detect multiple vehicles, but they also misclassify some objects. For instance, they identify roadside electrical boxes as cars, and in the case of `YOLOv5m`, they also misclassify traffic lights as pedestrians, as highlighted in red in Figures 4a, 4c, and 4e. `YOLOv5n` further fails to detect larger objects such as the truck and excavator (highlighted in blue; Figure 4e), whereas the medium variant succeeds in recognizing them and generally identifies more vehicles overall. These examples illustrate the trade-off between accuracy and model size [17]. Smaller models are computationally efficient but prone to missed detections (Figure 4a), while larger models achieve broader coverage at the expense of higher computational cost (Figure 4f).

(a) Frame 30175; YOLOV5n (b) Frame 30175; YOLOV5m

(c) Frame 24418; YOLOV5n (d) Frame 24418; YOLOV5m

(e) Frame 10614; YOLOV5n (f) Frame 10614; YOLOV5m

Figure 4: comparison of `YOLOv5n` and `YOLOv5m` on frames from the multi-view traffic intersection dataset [18]

3.3 Implementation

The solution implements a distributed edge computing architecture with four main software components implemented in Rust and Python. The code applies the actual inference in Python. The complete implementation is available on GitHub on our rust-traffic-watch repository[1]

The implementation supports automated benchmarking across multiple AI model variants (YOLOv5n/s/m), configurable frame rates (1-15 frames per second), and supports local- and offloading deployment configurations established in the following software components:

- **Controller**: Orchestrates experiments, coordinates device communication, and collects performance metrics across all distributed nodes
- **Pi Sender**: Captures frames and performs local PyTorch inference or forwards data for offloading to remote compute nodes
- **Jetson Receiver**: Provides accelerated TensorRT inference for offloaded workloads with optimized GPU utilization
- **Shared Library**: Common networking protocols, binary serialization, and type definitions for inter-device communication

All experiments generate detailed CSV logs containing end-to-end latency, processing overhead, network latency, and detection accuracy metrics for reproducible analysis across 24 configurations.

3.3.1 Hierarchical Time Measurement. Accurate time measurements across distributed computing nodes always present difficulties since clocks tend not to be fully synchronized. This paragraph describes the method we use to avoid this problem.

We measure time hierarchically across nodes $\mathcal{N} = \{N_c, N_p, N_r\}$, where N_c is the client, N_p a local processing node, and N_r a remote inference node triggered by N_p. Each node N_i maintains a local clock. The client N_c initializes the ordered timestamp structure $\mathcal{T}$ (`TimingMetadata`):

$$\mathcal{T} = \langle t_1, \ldots, t_n \rangle,$$

During execution, N_p appends timestamps for local processing and triggers N_r, which appends inference timestamps. We compare only timestamps acquired from the same local clock to avoid aligning the local clocks (C_c, C_p, C_r). So we define a duration between to moments τ_a, τ_b on node N_i as

$$\Delta T_i(t_a, t_b) = C_i(\tau_b) - C_i(\tau_a),$$

with $t_a = C_i(\tau_a)$ and $t_b = C_i(\tau_b)$. A valid duration requires:

$$\Delta T_i(t_a, t_b) \quad \text{only if} \quad t_a, t_b \leftarrow C_i,$$

i.e. to calculate the duration of a run is calculated as follows:

$$t_{\text{start}} = C_c(\tau_{\text{start}}),$$
$$t_{\text{end}} = C_c(\tau_{\text{end}}),$$
$$\Delta T_{\text{run}} = t_{\text{end}} - t_{\text{start}}.$$

4 EVALUATION DESIGN

In our evaluation, we systematically vary the capture frame rate *CFR* from 1 to 15 frames per second (FPS) and measure the impact of different inference models on total processing time, accuracy, and system responsiveness. These parameters allow us to analyze trade-offs between computation location, network usage, and classification performance, providing insights into the design of smart

[1] https://github.com/rawalcher/rust-traffic-watch

https://doi.org/10.34749/3061-1008.2025.7

city edge AI systems, considering the two different implementations of the vehicle classification smart city application.

4.1 Evaluation Metrics

Table 1 summarizes the notation used throughout the evaluation. This table serves as a reference for the various symbols and terms employed in the evaluation.

Table 1: Notation used in describing the experiments

Symbol	Description	Unit
CFR	Capture frame rate	frames/s
DFR	Dropped frame rate	frames/s
OTR	On-time processing rate	frames/s
RPI	Relative performance improvement	-
$\mathcal{Z}_{\text{total}}$	Total nominal frames captured during experiment	frames
$\mathcal{Z}_{\text{dropped}}$	Amount of frames dropped during experiment	frames
ΔT_{total}	Total experiment duration	s
$\Delta T_{\text{proc}}(i)$	Processing time for frame i	s
μ_m	Max. sustainable processing rate for model m	frames/s
λ_0	Per-node frame generation rate	frames/s
Q_{size}	Capacity queue size	frames

4.1.1 Performance Metrics. The target metric in our evaluation to assess image inference is the *capture frame rate* (*CFR*), defined as:

$$CFR = \frac{\mathcal{Z}_{\text{total}}}{\Delta T_{\text{total}}}.$$

To handle varying processing speeds, we implement a *capacity-bounded queue* with size $Q_{\text{size}} = 1$, ensuring only the most recent frame is processed. When a new frame arrives, it replaces any waiting frames in the queue during ongoing processing. This policy prioritizes the timeline over completeness.

We define the *dropped frame rate* (*DFR*) as the ratio of frames discarded due to processing delays:

$$DFR = \frac{\mathcal{Z}_{\text{dropped}}}{\mathcal{Z}_{\text{total}}} \times 100\%.$$

A frame is considered dropped when:

(1) It arrives while the previous frame is still being processed $\Delta T_{proc}(i-1) > 1/CFR$, and
(2) It gets replaced by a newer frame before processing begins.

The *on-time processing rate* (*OTR*) represents successfully processed frames, $OTR = 100\% - DFR$.

4.1.2 Processing Time. The *average processing time per frame* T_{proc} is computed as:

$$\Delta T_{proc} = \frac{1}{n} \sum_{i} \Delta T_{proc}(i).$$

Where n represents the number of frames that the system successfully processes without dropping, while i indicates the index of the current frame, which determines the processing time ΔT_{proc} for that frame, how processing time is cummulated depends on the deployment configuration. $\Delta T_{proc} \in \{\Delta T^{\text{loc}}_{proc}, \Delta T^{\text{off}}_{proc}\}$

For the **local-deployment**, processing time ΔT^{loc}_{proc} accumulates Image acquisition from the camera, model inference, and result serialization.

For the **offloading-deployment** the processing time $\Delta T^{\text{off}}_{proc}$ includes additional components: network transmission t_{in} to the remote node, the queuing delay at the remote node, model inference ΔT_{inf} and network transmission of results ΔT_{out}.

4.1.3 Performance Comparison. To assess the benefit of GPU offloading, we define the *relative performance improvement* (*RPI*):

$$RPI = \frac{DFR_{\text{local}} - DFR_{\text{offload}}}{DFR_{\text{local}}} \times 100\%.$$

If $RPI > 0$, offloading reduces frame drops,
$RPI < 0$ shows that network overhead outweighs acceleration benefits. $RPI = 0$, both approaches work identically.

4.1.4 Model-Specific Processing Capacity. Each model variant $m \in \{\texttt{YOLOv5n}, \texttt{YOLOv5s}, \texttt{YOLOv5m}\}$ exhibits a maximum sustainable processing rate μ_m for each type t processing node and corresponding hardware capabilities. The processing capacity without drops demands:

$$CFR \leq \mu_m.$$

Due to the capacity queue implementation, the system can operate at higher capture rates but will drop frames according to:

$$DFR(m) \approx \max\left(0, (1 - \frac{\mu_m}{CFR})\right) \times 100\%.$$

This approximation assumes steady-state operation and uniform processing times.

5 RESULTS

We evaluate two deployment strategies: *local AI* inference directly on the roadside node (Raspberry Pi 4, CPU-only) and *remote inference* offloaded to a Jetson Nano (GPU-enabled) over a Wi-Fi link. Performance is quantified using the metrics introduced in Section 4: the *Dropped Frame Rate* (*DFR*), the *Capture Frame Rate* (*CFR*), and the *Inference Latency* (ΔT_{inf}).

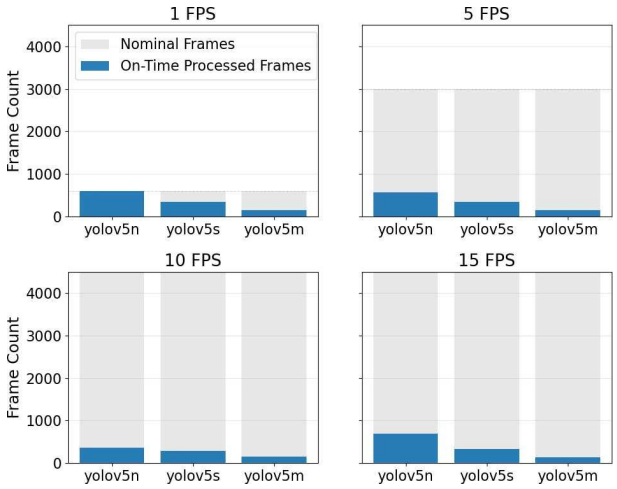


Figure 5: OTR over DFR using local deployment configuration (y-axis limited to 4000 frames)

Reviewing the results of local deployment configuration shows that sustained operation without frame drops ($DFR = 0$) is only achievable with the smallest model, YOLOv5n, at $CFR = 1$. This

https://doi.org/10.34749/3061-1008.2025.7

setting satisfies the condition $OTR \geq CPR$, so the next frame arrives only after the previous one is fully processed. Increasing the capture rate to 5 FPS or 15 FPS results in a significant reduction of successful processings, with DFR exceeding 80% for YOLOv5s and 95% for YOLOv5m. In these cases, the high DFR reflects the condition $OTR \ll CFR$, where most frames cannot be processed in time.

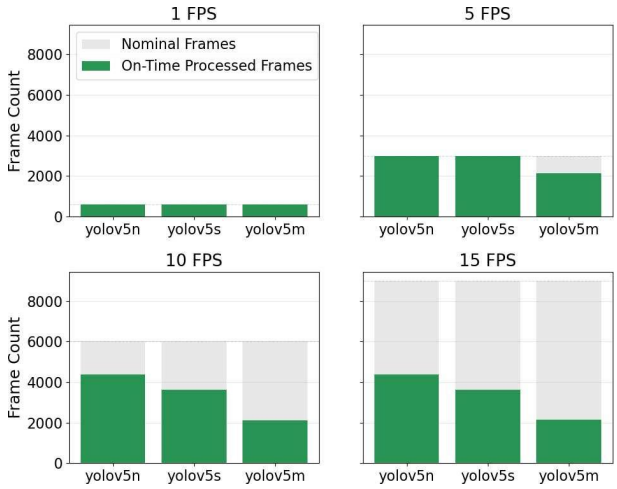


Figure 6: OTR over DFR using offloading deployment configuration

On the other hand, the situation changes quite a bit when offloading. With $CFR = 1$, we stay at $DFR \approx 0$, and we only start dropping frames at $CFR = 5$ with the bigger YOLOv5m model. Starting at $CFR = 10$, all models perform worse across the board with DFR ranging from $\approx 27\%$ (YOLOv5n) up to $\approx 64\%$ (YOLOv5m). At $CFR = 15$, we can see the same results as observed at $CFR = 10$, facing hardware processing limits on the remote inference node.

These findings confirm that for single-node deployments, local inference is feasible only for lightweight models and low capture rates. In contrast, offloading to GPU-equipped edge nodes enables near-real-time performance even for heavier models at higher capture rates, as long as OTR remains sufficiently close to CFR to keep DFR within the acceptable bound DFR^{max}. Furthermore, on lower $CFR \leq 5$ resource capabilities of the offloading node suffice to process data of multiple nodes.

6 DISCUSSION

Based on the experimental evaluation, we found that processing the video stream on a single node can be highly restrictive. Many real-world deployments of edge AI involve multiple processing nodes sharing one offloading node. Such a configuration introduces several open challenges, like resource contention, adaptive scheduling, and reliable network performance despite the actual multi-node distribution of service capabilities.

Multi-node scalability allows us to optimize resource usage. Increasing the number of camera nodes will increase the total data rate and computational demand on the AI node [7], potentially overloading its GPU or saturating the Wi-Fi network (see Section 3.1.3). Understanding the limits of performance degradation and identifying optimal load-balancing and queuing strategies will be critical. This also demands long-term reliability testing in outdoor urban environments with variable interference, integrating more advanced AI models for real-time object detection and tracking, and applying compression techniques to reduce bandwidth requirements without compromising classification accuracy.

Finally, *network optimization* techniques will be necessary to mitigate interference and congestion in dense Wi-Fi environments. This includes channel allocation strategies, use of Wi-Fi 7 [5] features like Multi-Link Operation (MLO) [3] and 4K Quadrature Amplitude Modulation (4K-QAM) [8], and integration with mesh networking to extend coverage without sacrificing performance or going beyond those aspects of 6G.

7 CONCLUSION

This paper introduced a Wi-Fi-based Edge AI framework for real-time traffic image classification in smart city environments. We implemented a system where low-power camera nodes either perform on-device inference using their CPU or offload the processing to an NVIDIA Jetson Nano for GPU-accelerated inference. By defining formal performance metrics such as inference latency and dropped frame rate, we could assess the feasibility of both local and offloaded processing strategies.

Our results demonstrate that purely local processing is only practical for lightweight AI models at low frame rates. For instance, YOLOv5n at 1 FPS processed all frames without drops. However, increasing the frame rate or the model complexity quickly caused dropped frame rates to exceed operational thresholds up to 98% in some configurations. Conversely, GPU offloading to the Jetson Nano maintained adequate processing at low to moderate frame rates across all tested models, with only minimal performance degradation for heavier models at higher frame rates.

These findings underscore the value of GPU-accelerated offloading over Wi-Fi for achieving reliable, low-latency performance in urban traffic monitoring scenarios [22] [15] without the need for costly 5G or wired infrastructure [9]. The approach is particularly attractive for **ultra-local deployments** where coverage is contained within a single Wi-Fi cell, offering a cost-effective yet high-performance alternative to more infrastructure-heavy solutions. Beyond the specific traffic monitoring use case, the proposed framework demonstrates the broader viability of Wi-Fi-based edge computing for other urban sensing applications, provided the computational load and network conditions are appropriately managed. This positions localized Wi-Fi edge AI as a practical enabler for smart city services in settings where traditional wide-area edge or cloud deployments are impractical.

ACKNOWLEDGMENT

This work received funding from the OeAD agency of the Austrian Ministry for Education, Science and Research and the Macedonian Ministry for Education and Science under project number MK10/2024.

REFERENCES

[1] Md Eshrat E Alahi, Arsanchai Sukkuea, Fahmida Wazed Tina, Anindya Nag, Wattanapong Kurdthongmee, Korakot Suwannarat, and Subhas Chandra Mukhopadhyay. 2023. Integration of IoT-enabled technologies and artificial intelligence (AI) for smart city scenario: recent advancements and future trends. *Sensors* 23, 11 (2023), 5206.

https://doi.org/10.34749/3061-1008.2025.7

[2] Johan Barthélemy, Nicolas Verstaevel, Hugh Forehead, and Pascal Perez. 2019. Edge-computing video analytics for real-time traffic monitoring in a smart city. *Sensors* 19, 9 (2019), 2048.

[3] Marc Carrascosa-Zamacois, Lorenzo Galati-Giordano, Francesc Wilhelmi, Gianluca Fontanesi, Anders Jonsson, Giovanni Geraci, and Boris Bellalta. 2024. Performance Evaluation of MLO for XR Streaming: Can Wi-Fi 7 Meet the Expectations?. In *2024 IEEE 29th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. IEEE, 1–6.

[4] Ning Chen, Tie Qiu, Laiping Zhao, Xiaobo Zhou, and Huansheng Ning. 2021. Edge Intelligent Networking Optimization for Internet of Things in Smart City. *IEEE Wireless Communications* 28, 2 (2021), 26–31. https://doi.org/10.1109/MWC.001.2000243

[5] Cailian Deng, Xuming Fang, Xiao Han, Xianbin Wang, Li Yan, Rong He, Yan Long, and Yuchen Guo. 2020. IEEE 802.11 be Wi-Fi 7: New challenges and opportunities. *IEEE Communications Surveys & Tutorials* 22, 4 (2020), 2136–2166.

[6] Laura García-García, Jose M Jiménez, Miran Taha Abdullah Abdullah, and Jaime Lloret. 2018. Wireless technologies for IoT in smart cities. *Network Protocols and Algorithms* 10, 1 (2018), 23–64.

[7] Raj Hakani and Abhishek Rawat. 2024. Edge computing-driven real-time drone detection using YOLOv9 and NVIDIA Jetson nano. *Drones* 8, 11 (2024), 680.

[8] Roger Pierre Fabris Hoefel. 2024. Effects of Phase Noise and Frequency Offset on the Performance of 4K-QAM and 16K-QAM in 802.11 be and 802.11 bn WLANs. In *2024 IEEE Wireless Communications and Networking Conference (WCNC)*. 1–6.

[9] Kurt Horvath, Dragi Kimovski, Stojan Kitanov, and Radu Prodan. 2025. Enhancing Traffic Safety with AI and 6G: Latency Requirements and Real-Time Threat Detection. In *2025 10th International Conference on Information and Network Technologies (ICINT)*. IEEE, 129–136.

[10] Kurt Horvath, Shpresa Tuda, Blerta Idrizi, Stojan Kitanov, Fisnik Doko, and Dragi Kimovski. 2025. 6G Infrastructures for Edge AI: An Analytical Perspective. *arXiv preprint arXiv:2506.10570* (2025).

[11] Monica Karel Huerta, Jessica Garizurieta, Rubén González, Luis-Ángel Infante, Melina Horna, Renato Rivera, and Roger Clotet. 2023. A long-distance wifi network as a tool to promote social inclusion in southern veracruz, mexico. *Sustainability* 15, 13 (2023), 9939.

[12] Glenn Jocher, Ayush Chaurasia, Jirka Borovec, and Ultralytics. 2020. YOLOv5. https://github.com/ultralytics/yolov5.

[13] Dragi Kimovski, Roland Mathá, Josef Hammer, Narges Mehran, Hermann Hellwagner, and Radu Prodan. 2021. Cloud, Fog, or Edge: Where to Compute? *IEEE Internet Computing* 25, 4 (2021), 30–36. https://doi.org/10.1109/MIC.2021.3050613

[14] Zhiming Luo, Frederic Branchaud-Charron, Carl Lemaire, Janusz Konrad, Shaozi Li, Akshaya Mishra, Andrew Achkar, Justin Eichel, and Pierre-Marc Jodoin. 2018. MIO-TCD: A new benchmark dataset for vehicle classification and localization. *IEEE Transactions on Image Processing* 27, 10 (2018), 5129–5141.

[15] Aale Luusua, Johanna Ylipulli, Marcus Foth, and Alessandro Aurigi. 2023. Urban AI: Understanding the emerging role of artificial intelligence in smart cities. *AI & society* 38, 3 (2023), 1039–1044.

[16] Madhusri Maity, Sriparna Banerjee, and Sheli Sinha Chaudhuri. 2021. Faster r-cnn and yolo based vehicle detection: A survey. In *2021 5th international conference on computing methodologies and communication (ICCMC)*. IEEE, 1442–1447.

[17] Gaurav Menghani. 2023. Efficient deep learning: A survey on making deep learning models smaller, faster, and better. *Comput. Surveys* 55, 12 (2023), 1–37.

[18] Andreas Møgelmose. 2019. Multi-view Traffic Intersection Dataset. Kaggle. https://www.kaggle.com/datasets/andreasmoegelmose/multiview-traffic-intersection-dataset

[19] Zhaolong Ning, Jun Huang, and Xiaojie Wang. 2019. Vehicular fog computing: Enabling real-time traffic management for smart cities. *IEEE Wireless Communications* 26, 1 (2019), 87–93.

[20] Héctor Rodríguez-Rangel, Luis Alberto Morales-Rosales, Rafael Imperial-Rojo, Mario Alberto Roman-Garay, Gloria Ekaterine Peralta-Peñuñuri, and Mariana Lobato-Báez. 2022. Analysis of statistical and artificial intelligence algorithms for real-time speed estimation based on vehicle detection with YOLO. *Applied Sciences* 12, 6 (2022), 2907.

[21] Tarik Taleb, Sunny Dutta, Adlen Ksentini, Muddesar Iqbal, and Hannu Flinck. 2017. Mobile Edge Computing Potential in Making Cities Smarter. *IEEE Communications Magazine* 55, 3 (2017), 38–43. https://doi.org/10.1109/MCOM.2017.1600249CM

[22] Radosław Wolniak and Kinga Stecuła. 2024. Artificial intelligence in smart cities—applications, barriers, and future directions: a review. *Smart cities* 7, 3 (2024), 1346–1389.

[23] Yu Zhang, Zhongyin Guo, Jianqing Wu, Yuan Tian, Haotian Tang, and Xinming Guo. 2022. Real-time vehicle detection based on improved yolo v5. *Sustainability* 14, 19 (2022), 12274.

[24] Muhammad Azhad Bin Zuraimi and Fadhlan Hafizhelmi Kamaru Zaman. 2021. Vehicle detection and tracking using YOLO and DeepSORT. In *2021 IEEE 11th IEEE symposium on computer applications & industrial electronics (ISCAIE)*. IEEE, 23–29.

  https://doi.org/10.34749/3061-1008.2025.7