Orchestrating Composite Applications at the Edge

Orchestrating Composite Applications at the Edge

Antonio Calagna*,†, Stefano Ravera*, Carla Fabiana Chiasserini*†‡
*Politecnico di Torino, Italy
†CNIT, 43124 Parma, Italy
‡Chalmers University of Technology, Sweden

***Abstract*—Edge computing plays a pivotal role in enabling time-critical Machine Learning (ML) by bringing computational capabilities closer to end users. However, satisfying stringent inference latency and quality constraints under varying task complexity and limited edge resources remains challenging. We tackle this by proposing a novel architectural approach for ML-based edge deployments and introducing CARE, our orchestration framework that jointly configures composite applications and compute resources to meet inference latency and quality targets, while minimizing energy consumption. Experimental results in the context of Multi-Object Tracking (MOT) demonstrate that CARE improves inference quality by up to 50% and reduces latency by up to $2\times$ over monolithic baselines.**

***Index Terms*—Edge Computing, Orchestration, ML Tasks**

I. Introduction

Edge computing has emerged as a key enabling paradigm to support latency-sensitive applications at the network edge. By bringing computation closer to end users, this paradigm is particularly beneficial for executing complex, time-critical Machine Learning (ML) tasks, e.g., real-time video analytics and autonomous systems control, as it enables efficient processing close to data sources [1]–[3]. However, ML inference at the edge entails a fundamental trade-off between inference latency, quality, and computational load, with a direct impact on the end users' Quality of Experience (QoE). Managing this trade-off is non-trivial, especially under varying levels of task complexity and stringent resource constraints, thereby calling for efficient and adaptive orchestration strategies [4], [5].

In this context, our key observation is that current ML-based applications have a monolithic architecture, which may significantly limit both system efficiency and performance [6]. In fact, while commonly adopted, such architectures are inherently non-scalable and therefore ill-suited for edge environments, often characterized by heterogeneous resources and tight availability constraints. As a result, they frequently fail to meet QoE targets, particularly under dynamic workloads or strict inference latency and quality constraints [7], [8].

We illustrate this limitation with a practical Multi-Object Tracking (MOT) scenario implemented using Ultralytics [9] and the official YOLO nano inference model. Fig. 1 shows the number of vCPUs required to meet varying inference latency targets for increasing levels of MOT task complexity—defined by the number of tracked objects and video resolution. Each level yields a distinct inference quality value, measured as the percentage of successfully tracked objects. Results not only demonstrate that vCPU demand scales with task complexity but also that a point is eventually reached where *no feasible allocation can satisfy the latency constraints*, thus highlighting the inherent scalability limits of monolithic architectures.

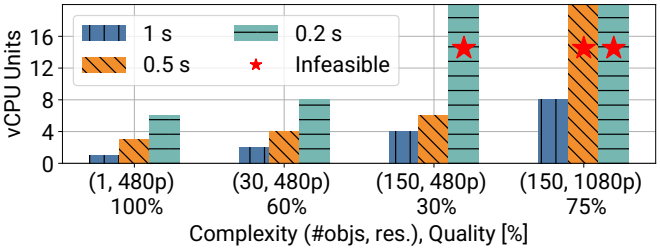


Fig. 1: Compute footprint of a monolithic MOT application vs task complexity, for varying inference latency targets (1, 0.5, 0.2 s).

To tackle this issue, we propose splitting ML applications from monolithic architectures to composite, distributed ones, composed of interconnected functional blocks, each handling a core processing task. This modular design enables independent scaling of each block and fine-grained orchestration across heterogeneous edge nodes, thus unlocking deployments that are feasible with respect to both resource and latency constraints. This vision poses several technical challenges: *(i)* determining the number of instances to deploy for each functional block, *(ii)* implementing routing mechanisms to manage inter-block communication, *(iii)* ensuring data consistency across distributed components and robustness to edge node failures, and *(iv)* allocating resources to each block instance to jointly satisfy inference latency, quality, and capacity constraints.

We address these challenges by proposing CARE, a novel orchestration framework for Composite Applications and compute Resources at the Edge, which jointly configures the application structure, data communication flow, and resource allocation to enable scalable, QoE-compliant ML inference at the edge. Unlike existing solutions [10], [11], CARE is the first to focus on composite edge applications and jointly addressing all the above challenges, i.e., by jointly computing the optimal instance placement and resource allocation policy that minimizes the overall system energy footprint, and by configuring the composite application to effectively satisfy target inference latency and quality constraints.

In the following, we formulate the optimization problem and introduce our algorithmic solution framework, CARE. Then, we experimentally validate it on our real-world testbed for MOT tasks and show that it effectively identifies the joint

This work was supported by the SNS-JU-2022 project ADROIT6G under the EUs Horizon Europe research and innovation programme under Grand Agreement No. 101095363

application and resource configuration that minimizes energy consumption while meeting strict inference latency constraints and flexibly adapting to heterogeneous edge environments.

II. System Model and Problem Formulation

This section presents our reference system architecture integrating CARE, defines the corresponding system model, and formulates the Processing- and Resource-aware ML Inference Scaling and Management (PRISM) problem.

Reference architecture. Our reference system architecture, depicted in Fig. 2, comprises a set of end devices connected via 5G/nextG BSs to an edge cluster of heterogeneous servers. Due to their limited compute capability, these devices offload ML tasks to the edge, where they are executed by a distributed composite application consisting of scalable, interconnected functional blocks. The edge cluster includes three key system components: *(i)* an application performance monitor gathering all the relevant metrics concerning the task, e.g., inference latency, quality, and energy consumption; *(ii)* a resource monitoring and management module that keeps track of resource availability across edge servers and actuates resource allocation decisions for each functional block; and *(iii)* a shared state layer, which synchronizes task-related data across block instances, ensuring seamless scalability and resilience to node failures. The core architectural component is our CARE orchestrator, which leverages both application performance metrics and edge resource availability to generate a joint application and resource control policy that allows meeting arbitrary target values of inference latency and quality.

System model. Let $\mathcal{S}$ denote the set of edge servers, where each server $s{\in}\mathcal{S}$ has a limited amount of available computational resources: C_s vCPUs and G_s GPUs. We consider a generic ML task whose complexity is described by the k-dimensional vector $\boldsymbol{\theta}{\in}\mathbb{R}^k$, which encompasses all the relevant aspects that may influence computational load and inference performance (e.g., input dimensionality and variability). This task is offloaded from an end device to the edge and carried out by a distributed composite application, which is structured as a set $\mathcal{B}$ of functional blocks, each indexed by $i{\in}\mathcal{B}$.

Given the task complexity $\boldsymbol{\theta}$ and the resource availability at each server (C_s, G_s), the first objective is to determine the number of instances of each functional block i to deploy on server s, denoted by $x_{i,s}{\in}\mathbb{Z}^+$. Let $x_i{=}\sum_{s\in\mathcal{S}} x_{i,s}$ be the total number of instances for block i. Then, to ensure consistent and predictable performance, we consider all instances of a block to be provisioned with identical resource allocations. Let $c_i{\in}\mathbb{Z}^+$ be the number of vCPUs assigned to each instance of block i, and let $g_i{\in}\{0,1\}$ be a binary decision variable indicating whether GPU acceleration is enabled for block i. Specifically, if $g_i{=}1$ each instance of block i is granted access to a GPU—whose sharing with other processes is managed by its native scheduler; otherwise all instances run on CPU only.

Each ML task is associated with a required level of user QoE, defined by target values of inference latency L and quality Q. For a generic timestamp t, we define the inference latency as $\ell_t(x_{i,s}, c_i, g_i; \boldsymbol{\theta})$ and the inference quality as

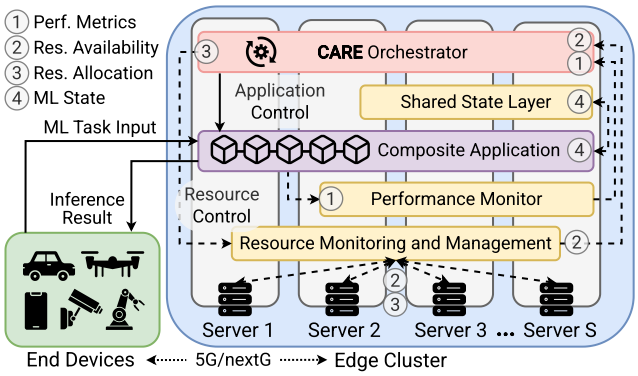


Fig. 2: Reference system architecture integrating CARE.

$q_t(x_{i,s}; \boldsymbol{\theta})$. While ℓ_t depends on both the total number of instances of each block and their resource allocation, q_t is independent of the specific allocation strategy. Finally, we denote the aggregated energy consumption of all instances of block i running on server s as $E_{i,s}(x_{i,s}, c_i, g_i; \boldsymbol{\theta})$.

Problem formulation. We now formulate the *Processing- and Resource-aware ML Inference Scaling and Management* (PRISM) as a mixed-integer nonlinear programming problem.

PRISM Problem

$$\min_{x_{i,s},\, c_i,\, g_i} \quad \sum_{i\in\mathcal{B}} \sum_{s\in\mathcal{S}} E_{i,s}\big(x_{i,s}, c_i, g_i;\, \boldsymbol{\theta}\big)$$

$$\text{s.t.}: \quad \ell_t\big(x_i, c_i, g_i;\, \boldsymbol{\theta}\big) \le L \quad \forall i, t \tag{1}$$

$$q_t\big(x_i;\, \boldsymbol{\theta}\big) \ge Q \quad \forall i, t \tag{2}$$

$$\sum_{i\in\mathcal{B}} x_{i,s}\, c_i \le C_s \quad \forall s \tag{3}$$

$$\sum_{i\in\mathcal{B}} x_{i,s}\, g_i \le G_s \quad \forall s \tag{4}$$

PRISM aims to define a joint functional block placement $x_{i,s}$ and resource allocation $(c_i,\ g_i)$ policy that minimizes the overall system's energy consumption. Constraints 1 and 2 enforce, at any time instant, the inference latency and quality constraints. Constraints 3 and 4 impose that the aggregate CPU and GPU resources allocated for all instances of all blocks running on server s do not exceed its resource availability. Specifically, when $g_i{=}0$, Constraint 4 is vacuous and the optimization finds the best CPU-only configuration; when $g_i{=}1$, the GPU constraint caps the instances count to G_s, ensuring all instances receive a GPU.

Complexity. PRISM is highly non-linear and the relation between input parameters and performance metrics is application-dependent. Such non-linearities, combined with the heterogeneity of the possible ML tasks, make the derivation of a closed-form model—and thus an exact optimal solution—impractical. We thus design an algorithmic solution to PRISM that leverages real-time metrics profiling for an online, computationally efficient execution.

III. The CARE Algorithmic Solution

This section introduces the core component of CARE, i.e., our algorithmic solution to PRISM. For simplicity and without loss of generality, we now focus on a single functional block, and, thus, omit the functional block index i.

CARE Algorithmic Solution

Require: • ML task complexity $\boldsymbol{\theta}$, • Resource availability C_s and G_s, • Target latency L, • Target quality Q
Ensure: Placement $\{x_s\}_{s\in\mathcal{S}}$ and allocation (c, g); else INFEASIBLE

$x \leftarrow 1$, $c \leftarrow 1$, $g \leftarrow 0$ ▷ start with one instance and minimal resources
while true do
  if $\nexists\, s\in\mathcal{S}$ such that $C_s \geq 1$ **and** $G_s \geq g$ **then** ▷ cluster out of resources
    return INFEASIBLE
  $\{x_s\}_{s\in\mathcal{S}} \leftarrow 0$, $C'_s \leftarrow C_s$, $G'_s \leftarrow G_s$ ▷ initialize variables
  Sort $\mathcal{S}$ by descending (G'_s, C'_s)
  success $\leftarrow$ **true** ▷ initialize placement success
  for $r=1$ **to** x **do** ▷ try to place a total of x instances
    placed $\leftarrow$ **false**
    for $s\in\mathcal{S}$ **do**
      if $C'_s \geq c$ **and** $G'_s \geq g$ **then**
        $C'_s \leftarrow C'_s - c$; $G'_s \leftarrow G'_s - g$; $x_s \leftarrow x_s + 1$;
        placed $\leftarrow$ **true**; **break**
    if not placed **then**
      success $\leftarrow$ **false**; **break**
  if success **then**
    Deploy the application with current (x, c, g) configuration
    Execute the ML task and measure actual quality $\hat{q}$ and latency $\hat{\ell}$
    if $\hat{q} < Q$ **then** ▷ quality target not met, increase instances
      $x \leftarrow x+1$, $c \leftarrow 1$, $g \leftarrow 0$; **continue**
    else if $\hat{\ell} \leq L$ **then** ▷ target latency met
      $C_s \leftarrow C'_s$, $G_s \leftarrow G'_s$ ▷ commit resource allocation
      return allocation (c, g) and placement $\{x_s\}_{s\in\mathcal{S}}$
    else ▷ target latency not met, increase CPU allocation
      $c \leftarrow c+1$; **continue**
  else if $g=0$ **then** ▷ CPU-only placement failed, try GPU acceleration
    $g \leftarrow 1$; $c \leftarrow 1$; **continue**
  else ▷ GPU allocation also failed, increase parallelism
    $x \leftarrow x+1$; $c \leftarrow 1$; $g \leftarrow 0$; **continue**

The algorithm implements a greedy, yet effective, resource allocation and placement strategy for deploying a generic functional block across heterogeneous edge servers. It incrementally explores configurations defined by the total number of instances x, vCPUs allocation per instance c, and GPU acceleration flag g, so as to meet target inference latency L and quality Q, under given resource constraints C_s and G_s.

At each iteration, the algorithm attempts to place x instances across servers sorted by descending available resources, ensuring that each instance receives exactly c vCPUs and, if enabled, access to a GPU (compatibly with C_s and G_s). The configuration is then deployed and used to experimentally measure the average quality $\hat{q}$ and latency $\hat{\ell}$ over an arbitrary time interval. If $\hat{q}<Q$, x is increased; if $\hat{\ell}>L$, c is incremented. When CPU-only configurations become infeasible, the algorithm enables GPU acceleration ($g=1$) and resets c. If both CPU and GPU allocations fail, the algorithm increases parallelism through x and retries the placement process.

The procedure ends when a feasible configuration is found or available resources are exhausted, in which case the deployment is deemed infeasible, indicating the need for higher-capacity severs or adaptation at the application level, e.g., via ML model complexity reduction. Importantly, since energy consumption grows monotonically with each escalation step—namely, adding instances, increasing vCPUs, or enabling GPU usage—*the first feasible configuration found under this greedy ordering is also the one that maximizes energy efficiency.*

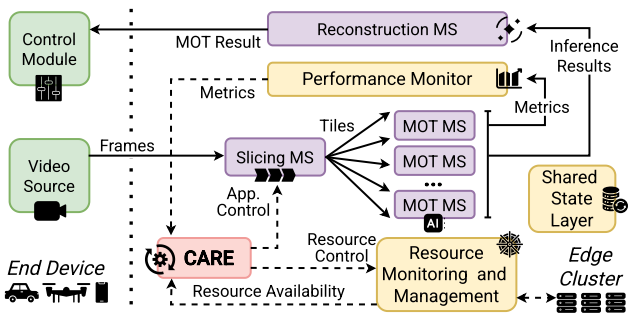


Fig. 3: Reference use case scenario for MOT tasks.

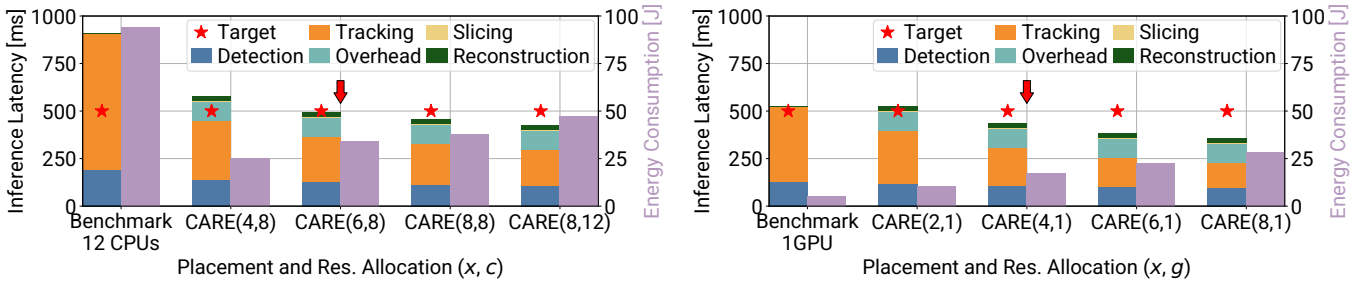
IV. Experimental Evaluation

To experimentally assess the benefits of CARE, we validate it in the real-world use case of MOT. Below, we first describe the reference scenario and the testbed we developed, and then we show the results obtained via our testbed.

Reference scenario and Testbed. We consider the practical scenario in Fig. 3, including an end device equipped with an on-board camera and an MOT task that performs inference on the camera stream, whose result is then consumed by a local control module to carry out time-critical functions (e.g., trajectory monitoring and collision avoidance). Due to the device's limited compute capacity and the need for low-latency response, the MOT task is offloaded to an edge cluster. There, it is executed by a composite application consisting of three key functional blocks: (i) a slicing microservice (MS), which partitions incoming video frames into tiles to enable parallel and scalable processing, (ii) a pool of MOT MSs, which perform object tracking on each tile, producing localized inference results, and (iii) a reconstruction MS, which aggregates and refines such results into a coherent global tracking output. Three additional cluster components support task execution: (a) a performance monitor, which collects MOT-related metrics and measures system energy consumption; (b) a shared state layer, which ensures consistency of tracking data across the distributed MSs; and (c) a resource monitoring and management module, which collects server-level resource information and enforces instances resource allocation. The CARE orchestrator runs our algorithm to (i) set the number of tiles per frame based on a given inference quality target and configure the slicing MS accordingly; (ii) compute the MOT resource allocation policy that satisfies the target inference latency while minimizing energy consumption; (iii) deploy a number of MOT instances equal to the number of tiles, with each instance assigned a specific tile to process.

Our testbed is built on top of the CrownLabs[1] bare-metal Kubernetes cluster, hosted at Politecnico di Torino, Italy. The cluster comprises 6 Dell PowerEdge R740x servers, each with

[1] crownlabs.polito.it, prometheus.io, sustainable-computing.io, redis.io, github.com/NirAharon/BOT-SORT, github.com/eclipse-zenoh/zenoh



(a) CPU-only, g=0

(b) GPU-accelerated, g=1

Fig. 4: Inference latency and energy consumption for a single video frame: CARE vs. the monolithic benchmark, across varying configurations of total number of MOT instances (x) and resource allocation (c, g). The arrow highlights the configuration selected by CARE.

an Intel Xeon Gold 5120 (28 vCPUs), 64 GB of RAM, and an NVIDIA Jetson Orin Nano GPU. It includes (i) the Kubelet, i.e., the Kubernetes agent that monitors server-level resources and enforces instance-level allocations; (ii) Prometheus[1] and Kepler[1] for fine-grained performance and energy monitoring; and (iii) a distributed shared state layer using Redis[1] (as in [12]) that ensures tracking consistency across MSs.

We developed and implemented a MOT composite application consisting of (i) a slicing MS inspired by SAHI [13], which partitions frames into tiles, (ii) an MOT MS based on YOLOv11 [9] with the official nano model and BoTSORT[1], which perform, respectively, object detection and tracking with re-identification, and (iii) a reconstruction MS, extending the Non-Max Merging (NMM) [14] method to maintain consistent object identities across tiles. Communication among MSs is managed via the efficient Zenoh communication protocol[1].

To evaluate performance under varying task complexity, we select three videos from the VisDrone dataset [15], each featuring a different average number of objects per frame (θ_{obj}) and available in multiple resolutions (θ_{res}). An FFmpeg MS processes these videos and generates an RTSP video stream with a fixed frame rate of 0.5 fps across all configurations, thus ensuring stable input-output processing without frame drops.

TABLE I: Avg. inference quality: monolithic benchmark vs CARE, for varying MOT task complexity and total inference instances x.

$(\theta_{\text{obj}}, \theta_{\text{res}})$	Benchmark	x=2	x=4	x=6
(1, 480p)	99%	99%	99%	99%
(30, 480p)	56%	68%	71%	73%
(30, 1080p)	80%	85%	87%	90%
(150, 480p)	30%	38%	43%	45%
(150, 1080p)	72%	79%	85%	88%

Results. We now validate CARE and evaluate its ability to meet inference quality and latency targets.

Target Quality. Tab. I compares the inference quality of CARE to that of the monolithic benchmark, for varying complexity $(\theta_{\text{obj}}, \theta_{\text{res}})$ and total MOT instances x. We measure inference quality as the percentage of correctly tracked objects (which is independent of the resource allocation per instance). Results show that CARE achieves comparable or greater inference quality compared to the benchmark, regardless of task complexity. Also, in dense scenarios with large θ_{obj}, such quality improves with increasing values of x. This trend aligns with the findings in [13], where frame slicing enhances object detection performance by successfully identifying small objects in large images that could not be identified otherwise. We thus conclude that, by enabling scalable MOT deployments, our composite approach permits to enhance inference quality by up to 50% with respect to the monolithic benchmark. Importantly, as detailed in our algorithm, such experimental profiling of inference quality versus x is instrumental for CARE to identify a feasible application configuration with respect to a given target.

Target Latency. Next, we focus on the most demanding scenario, i.e., we set θ_{obj}=150 and θ_{res}=1080p. Fig. 4 compares inference latency and energy consumption between the monolithic benchmark and CARE for varying total number of inference instances x, and resource allocation c and g. We set a target inference latency L=0.5 s and consider two scenarios, depending on whether GPU acceleration g is enabled or not. We separately illustrate these in, respectively, Fig. 4a and Fig. 4b. Results show that the overall inference latency is mainly determined by detection and tracking, while slicing, reconstruction, and inter-block communication overheads have a negligible impact. Fig. 4a considers as benchmark the monolithic YOLO deployed on a CPU-only server with up to 12 available vCPUs and compares it to CARE for increasing x and c. By scaling MOT instances and leveraging distributed edge resources, our solution can reduce inference latency by up to 2× compared to the benchmark. CARE indeed effectively selects the configuration that satisfies the target latency L with minimum energy consumption. Fig. 4b presents the performance of our composite approach against the same benchmark on a server that now features one GPU only. Since computation is predominantly carried out by GPU(s), we omit c, as varying it yields negligible effect on performance. Again, results highlight that our approach greatly outperforms the benchmark, enabling a latency reduction of up to 1.5× while minimizing energy consumption.

V. Conclusions

We investigated the fundamental trade-off between inference latency, quality, and resource consumption. To overcome the limitations of monolithic ML applications, we envisioned their redesign through a composite architecture of distributed independent components, thus enabling scalable and efficient resource utilization in heterogeneous edge environments. We

then introduced the CARE orchestration framework that jointly configures the application and edge resource allocation to meet target inference latency and quality, while minimizing energy consumption. Experimental results, obtained through the testbed we developed, demonstrate that CARE achieves up to 50% higher inference quality and up to $2\times$ lower inference latency compared to the monolithic benchmark.

REFERENCES

[1] J. Kaur, M. A. Khan, M. Iftikhar, M. Imran, and Q. Emad Ul Haq, "Machine learning techniques for 5G and beyond," *IEEE Access*, vol. 9, 2021.

[2] M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient acceleration of deep learning inference on resource-constrained edge devices: A review," *Proceedings of the IEEE*, vol. 111, no. 1, 2023.

[3] M. A. Altaf and M. Y. Kim, "Multiple object detection and tracking in autonomous vehicles: A survey on enhanced affinity computation and its multimodal applications," *ICT Express*, vol. 11, no. 4, 2025.

[4] T. Bai, H. Zhao, L. Huang, Z. Wang, D. I. Kim, and A. Nallanathan, "A decade of video analytics at edge: Training, deployment, orchestration, and platforms," *IEEE Communications Surveys & Tutorials*, 2025.

[5] X. Zhang and S. Debroy, "Resource management in mobile edge computing: A comprehensive survey," *ACM Computing Surveys*, vol. 55, no. 13s, Jul. 2023.

[6] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges," *IEEE Access*, vol. 11, 2023.

[7] S. Agarwal, F. Malandrino, C. F. Chiasserini, and S. De, "VNF placement and resource allocation for the support of vertical services in 5G networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 1, 2019.

[8] Z. Yu, Y. Zhao, T. Deng, L. You, and D. Yuan, "Less carbon footprint in edge computing by joint task offloading and energy sharing," *IEEE Networking Letters*, vol. 5, no. 4, 2023.

[9] G. Jocher and J. Qiu, *Ultralytics YOLO11*, 2024. [Online]. Available: github.com/ultralytics/ultralytics.

[10] Z. Huang, F. Dong, H. Zhu, *et al.*, "ADPTD: Adaptive data partition with unbiased task dispatching for video analytics at the edge," *IEEE Internet of Things Journal*, vol. 12, 2025.

[11] Y. Liang, S. Zhang, and J. Wu, "SplitStream: Distributed and workload-adaptive video analytics at the edge," *Journal of Network and Computer Applications*, vol. 225, 2024.

[12] A. Calagna, S. Ravera, and C. F. Chiasserini, "Enabling efficient collection and usage of network performance metrics at the edge," *Computer Networks*, vol. 262, 2025.

[13] F. C. Akyon, S. O. Altinuc, and A. Temizel, "Slicing aided hyper inference and fine-tuning for small object detection," *IEEE International Conference on Image Processing*, 2022.

[14] L. Kondrackis, *What is non-max merging?* 2024. [Online]. Available: blog.roboflow.com/non-max-merging/.

[15] P. Zhu, L. Wen, D. Du, *et al.*, "Detection and tracking meet drones challenge," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 11, 2021.