Split Computing for Vehicle Detection in Smart Parking

Milena F. Silva[1], **Heitor H. da Silva**[1], **Mateus C. Oliveira**[1], **Carlos Senna**[1], **Allan M. de Souza**[1], **Juliana F. Borin**[1], **Luiz F. Bittencourt**[1]

[1]Instituto de Computação (IC) – University of Campinas (UNICAMP)
Campinas – SP – Brazil

{m299743, h178291, m230437}@dac.unicamp.br,

{crsenna, allanms, jufborin, bit}@unicamp.br

Abstract. *The increasing complexity of deep learning models like YOLO challenges the capabilities of low-cost edge devices in smart parking systems. To address this gap, we propose a Split Computing architecture applied to vehicle detection, where the neural network is partitioned between an edge client and a central server. This design offloads intensive computation, reducing local inference time and resource usage while preserving model accuracy. We evaluated distinct partitioning configurations, assessing computational costs and communication latency. Results demonstrate the solution's effectiveness, highlighting its scalability for multi-tenant environments and sustainability potential by enabling the repurposing of low-cost TV set-top boxes as functional edge nodes.*

1. Introduction

The 2030 United Nations Agenda for Sustainable Development [Nations 2015] establishes a global mandate to transform cities into inclusive, safe, and sustainable environments. As part of this global effort, initiatives such as the "Smart Campus" at Unicamp[1] are leveraging the Internet of Things (IoT) to foster more efficient and intelligent urban infrastructures [Baggio et al. 2020]. One of the primary targets of these innovations is urban mobility, particularly the persistent challenge of parking management. It is estimated that up to 30% of traffic congestion in urban areas is caused by drivers searching for available parking spaces, a process that results in wasted travel time, increased fuel consumption, and higher greenhouse gas emissions [Shoup 2021]. A cost-effective approach to parking management involves vision-based smart parking systems, which employ cameras and deep learning to monitor real-time availability. As documented in the Unicamp parking project's 10-year evolution [da Luz et al. 2025a], system effectiveness heavily relies on the capability of these underlying models. Progressing from early architectures like GoogLeNet to modern YOLO variants has substantially improved detection accuracy, with the latest deployment achieving 98.65% using a TensorFlow Lite YOLOv11m model [da Luz et al. 2025a].

However, this pursuit of accuracy has created a significant and pressing 'computational gap'. While modern hardware such as the Raspberry Pi 5 is available, its high cost often prevents large-scale deployment in urban infrastructures. Therefore, this work focuses on low-cost edge devices, such as the Raspberry Pi 3[2]. (released in 2016),

[1]https://smartcampus.prefeitura.unicamp.br/
[2]https://www.raspberrypi.com/products/raspberry-pi-3-model-b/

which better represents the constraints of legacy infrastructure and the reality of smart city projects focused on the circular economy. These devices possess limited processing power, 1GB of RAM, and often run on 32-bit systems, restricting compatibility with modern software libraries. As documented in the 10-year evolution of the Unicamp parking project, the 8-second inference time for the deployed YOLOv11m model pushes this hardware to its operational limits, leading to increased resource utilization and energy consumption [da Luz et al. 2025a]. When scaling to larger infrastructures, network stack latency becomes critical, as the saturated CPU struggles to handle the communication overhead required to synchronize data across distributed nodes. Furthermore, bandwidth saturation arises as a bottleneck; the aggregate data volume from simultaneous video streams causes severe network congestion, leading to packet loss and unpredictable delays. This situation forces a difficult trade-off: developers must either sacrifice accuracy by adopting smaller models or invest in more expensive hardware, neither of which is a sustainable or scalable solution.

This research proposes bridging the identified computational gap by leveraging Split Computing [Kang et al. 2017], a distributed inference paradigm derived from the architecture originally proposed by [Gupta and Raskar 2018]. Instead of executing a monolithic deep learning model on a resource-constrained edge device, Split Computing partitions the neural network into two segments. The initial, lightweight layers are executed locally on the constrained device, while the resulting intermediate representations, commonly referred to as "smashed data", are transmitted to a more powerful server to complete the computationally intensive portion of the inference. This approach significantly reduces the computational load, memory footprint, and energy consumption on the edge, enabling the deployment o state-of-the-art models without requiring costly hardware upgrades.

One possible solution to these gaps would be a distributed structure dividing client processing with servers that have greater computing power. However, this introduces another critical challenge focused on identifying the ideal "cut-off layer", the specific point in the neural network architecture at which the model is partitioned between the client and the server. As [Marinova and Rakovic 2024] highlights, this decision involves a direct trade-off between on-device computation and network communication [Yao 2023]: cutting too late overloads the constrained device, while cutting too early forces the transmission of large intermediate feature maps, increasing communication overhead and network congestion.

Therefore, a central contribution of this work is the analysis of the communication-computation trade-off through the evaluation of different cut layers and three distinct model sizes. We assess these configurations to identify the optimal balance that minimizes inference latency while maximizing resource efficiency on constrained devices. By lowering these on-device computational requirements, our Split Computing approach has immediate practical implications. It not only extends the viable lifespan of existing hardware such as the Raspberry Pi 3, but it also unlocks the feasibility of using ultra-low-cost, repurposed hardware. This includes devices such as the TV boxes seized by the Federal Revenue Service, a potential application identified by the Unicamp Smart Campus initiative [da Luz et al. 2025b, Sato et al. 2024].

The remainder of this paper is organized as follows: Section 2 provides the back-

ground on Split Computing and the architectural concepts utilized. Section 3 discusses related work in edge intelligence and distributed inference. Section 4 details the proposed methodology, including the model partitioning strategy, communication security, and experimental setup. Section 5 presents the quantitative results regarding inference latency, resource usage, and network payload. Finally, Section 6 concludes the study and outlines directions for future research.

2. Split Learning and Split Computing

Split Computing forms the backbone of this work, serving as the architectural enabler for deploying state-of-the-art deep learning models on resource-constrained edge hardware [Eshratifar et al. 2019, Kang et al. 2017, Chakroun et al. 2021]. By shifting the execution paradigm from monolithic local inference to a collaborative client-server workload, it overcomes the hardware limitations of low-cost devices such as TV boxes. Understanding this paradigm is therefore crucial to appreciating how the proposed framework manages to bridge the gap between high detection accuracy and limited computational power.

Split Learning (SL) is a distributed machine learning paradigm originally designed to enable the collaborative training of deep neural networks (DNNs) on resource-constrained devices without direct sharing of raw data. Introduced by [Gupta and Raskar 2018], SL addresses a fundamental challenge in edge computing scenarios, where devices such as IoT sensors or the TV boxes used in this study lack the computational resources, both memory and processing capability, required to execute large, state-of-the-art ML models entirely on-device.

Unlike traditional centralized approaches that require transmitting raw data to a powerful server, thereby raising significant privacy and bandwidth concerns, SL partitions the neural network itself into multiple distinct sections. The fundamental operational flow involves dividing a deep neural network into at least two partitions: a device-side model (Head), which resides on the edge device, and a server-side model (Tail), which is hosted on a central server. The point of division is referred to as the cut layer.

The process begins on the client device, which feeds raw data (such as images) through the Head model. The device executes forward propagation only up to the cut layer, producing a set of intermediate activations known as smashed data. This smashed data serves as a compressed, abstract representation of the original input and is transmitted over the network to the server. Upon reception, the server uses these activations as input for the Tail model, completing the forward propagation to generate the final output (e.g., bounding box coordinates).

Since this work targets real-time inference rather than model training, only the forward propagation phase is executed, and the paradigm is henceforth referred to as Split Computing [Kang et al. 2017].

3. Related Work

Smart parking systems have increasingly adopted deep learning-based vehicle/parking space detection at the edge to reduce latency and dependence on cloud connectivity. Recent work evaluates modern YOLO variants and region-of-interest strategies

to improve robustness and scalability in real deployments, reinforcing that edge-side vision is feasible, but still constrained by device compute and networking conditions [P C P da Luz et al. 2026, da Luz et al. 2025a].

To overcome compute limitations, DNN execution is often partitioned between constrained devices and capable servers. Early systems like Neurosurgeon pioneered layer-level partitioning to reduce latency and energy by balancing local compute with transfer costs [Kang et al. 2017]. Recently, adaptive approaches (e.g., Autodidactic Neurosurgeon) learn optimal split points on-the-fly under changing network and server loads, benefiting mobile or variable wireless environments [Zhang et al. 2021].

For complex object detectors, Matsubara and Levorato highlight practical bottlenecks such as the large activation tensors produced by early backbone layers and the sensitivity of end-to-end delay to where the split occurs [Matsubara and Levorato 2020]. Their later survey systematizes split computing and early-exiting as intermediate points between fully on-device inference and full offloading, emphasizing the bandwidth/latency trade-offs that dominate in real deployments [Matsubara and Levorato 2022]. For object detection specifically, Chakroun *et al.* propose a distributed architecture for edge-assisted detection and quantify the benefits and costs of distributing the pipeline across networked resources [Chakroun et al. 2021]. Neduchal *et al.* provide a focused comparison of split computing scenarios using YOLOv8 as baseline, showing how architectural choices translate into communication overhead and performance differences [Neduchal et al. 2024].

Scalability with multiple concurrent cameras/clients is another under-addressed aspect in Split Computing for edge vision. In training-oriented split learning, sequential client interaction is known to cause straggler effects and high end-to-end latency when the number of participants grows; recent work proposes parallelization and resource management over wireless networks [Wu et al. 2023]. While our focus is inference, the same systems bottleneck (shared server compute, uplink contention, variable radio conditions) appears when many cameras stream to a shared edge server.

Existing Split Computing works for object detection typically emphasize single-client optimization or static split points, and fewer studies connect the design to multi-client scalability and MEC/5G-ready deployment assumptions. Our work targets this gap by (i) experimentally characterizing the compute/communication bottleneck for YOLO-based Split Computing in smart parking, (ii) analyzing multi-client contention at the edge server, and (iii) structuring the system to be network-agnostic, laying the groundwork for future evaluation under variable bandwidth conditions such as 5G/MEC deployments. Unlike prior works that assume homogeneous or simulated environments, our study is conducted on real repurposed consumer hardware (TV boxes), directly validating the feasibility of circular-economy deployments, a dimension absent from existing Split Computing benchmarks for smart parking.

4. Split Computing Framework

This section details the proposed Split Computing framework, structured to address the computational constraints of edge environments while maintaining high detection accuracy. We first present the general solution architecture, which establishes a device-agnostic blueprint for distributed inference. Subsequently, we describe the specific model partitioning strategies and the experimental implementation used to validate this architec-

ture in a real-world smart parking scenario.

4.1. Proposed Solution Architecture

To overcome the limitations of monolithic edge inference, we propose a three-tier hierarchical architecture designed for campus-wide scalability. The system is composed of: (i) the Edge Client, responsible for sensing and the Head Partition; (ii) the Fog Node, which hosts the Tail Partition and local application logic; and (iii) the Central Aggregator, responsible for global data visualization and system orchestration, where *aggregation* refers to the consolidation of occupancy metrics from distributed edge nodes, and is not related to model-parameter aggregation as in federated learning scenarios. As illustrated in Figure 1, this architecture decouples the sensing and initial feature extraction from the computationally intensive deep learning tasks.

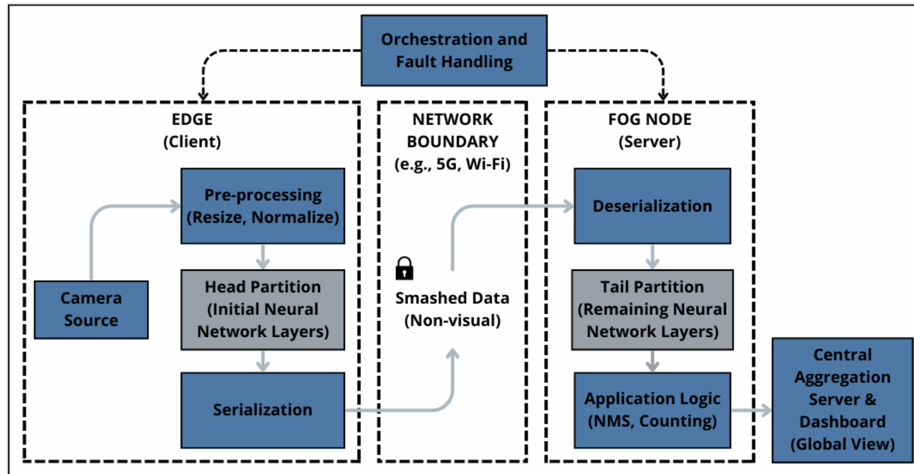

Figure 1. Smart Parking Application with Split Computing Strategy.

The workflow consists of the following modular components:

1. **Data Acquisition & Pre-processing (Edge):** The client device continuously captures video frames from the connected camera. Basic pre-processing operations (resizing and normalization) are performed locally to match the model's input requirements.
2. **Head Partition (Edge):** Instead of executing the full neural network, the client runs only the initial layers (the "Head"). This partition acts as a feature extractor, transforming raw high-dimensional images into compressed intermediate representations, referred to as "smashed data."
3. **Serialization & Transmission (Network):** The smashed data is serialized into a binary payload and transmitted over the network boundary. This layer is designed to be agnostic to the underlying transport technology, supporting diverse protocols (e.g., Wi-Fi, 5G, or Ethernet).
4. **Tail Partition (Fog Node):** A nearby Fog Node (FN) receives the payload, deserializes the tensor, and feeds it into the remaining layers of the model (the "Tail"), completing the server-side forward propagation close to the data source.
5. **Final Inference & Application Logic (Fog Node):** The Fog Node generates the final detection output (e.g., bounding boxes). Post-processing steps, such as Non-Maximum

Suppression (NMS) and vehicle counting, are executed at this layer to centralize complex logic and reduce the burden on the edge while keeping low latency.

6. **Global Aggregation & Real-time Visualization (Central Server):** The processed occupancy metrics are forwarded to a Central Aggregation Server, which maintains a campus-wide view and provides dashboards or external client applications with real-time updates on available parking spots, completing the end-to-end service loop.

A critical advantage of this architecture is its inherent scalability. As depicted in Figure 2, the system is designed as a one-to-many topology, where a single Server acts as an inference aggregator for multiple distributed Edge Clients. By centralizing the most intensive computing resources (Tail Partitioning), the system allows for cost-effective expansion of the sensitive mesh. In this way, new parking lots can be added simply by deploying low-cost edge nodes, without the need for dedicated high-performance hardware at each location.

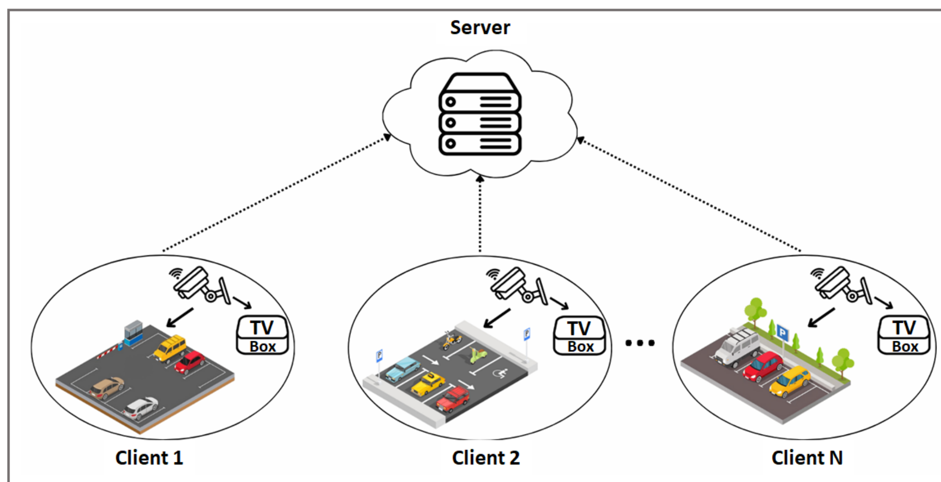

Figure 2. Proposed scenario for a multi smart parking.

Crucially, the architecture is designed to be hardware-agnostic. By decoupling the lightweight ’Head’ partition, the framework supports a broad spectrum of edge devices beyond the evaluated TV Boxes, including standard SBCs (e.g., Raspberry Pi) and accelerators (e.g., NVIDIA Jetson). This flexibility enables heterogeneous deployments that allow operators to mix ultra-low-cost hardware with high-performance nodes based on site-specific needs.

4.2. Privacy-by-Design and Data Obfuscation

The proposed architecture establishes an inherent layer of privacy, aligning with the Brazilian General Data Protection Law (LGPD). By performing initial feature extraction at the edge, the system transmits only abstract representations, known as smashed data. Unlike traditional surveillance systems that stream raw video, our server receives only intermediate tensors that are visually unrecognizable to human observers or potential interceptors. This process ensures technical anonymization at the source, as sensitive visual information (e.g., faces or license plates) is filtered before leaving the edge device, fulfilling the principles of data minimization and security-by-design.

4.3. YOLOv8 Architecture

To evaluate the proposed architecture, we implement the three partitioning scenarios defined by [Neduchal et al. 2024], as visualized in Figure 3. These configurations, referred to hereafter as Early Split (ES), Middle Split (MS), and Late Split (LS), cut the YOLOv8 architecture at varying depths within the model's backbone. By selecting these specific cut points, we aim to analyze the trade-off between the computational burden placed on the TV Box and the bandwidth required to transmit the smashed data. These specific layers were selected to create distinct scenarios of increasing computational load on the client device (TV Box), while maintaining a fixed network overhead. Unlike typical split configurations where deeper cuts reduce transmission size, the specific split points chosen for this study produce intermediate tensors (smashed data) of identical dimensions. Consequently, the payload size and network communication time remain constant across the Early, Middle, and Late split scenarios, as can be seen in Section 5.4. This unique configuration eliminates the bandwidth variable, allowing us to isolate and precisely evaluate the impact of the client's increasing processing burden on the total system latency.

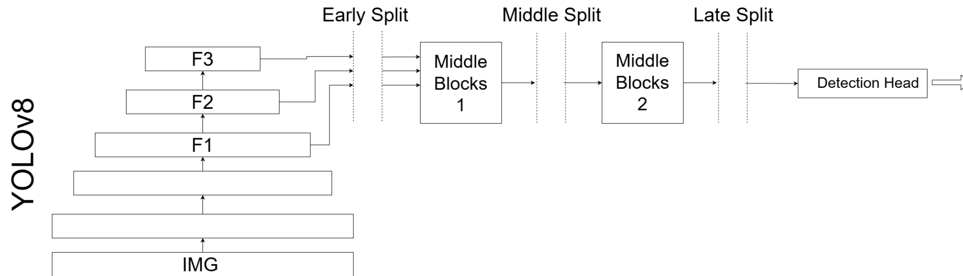

Figure 3. Split points in the YOLOv8 structure.

To evaluate the performance of our split-computing approach under significant computational demands, we selected three large pre-trained variants of YOLOv8: Medium (m), Large (l), and Extra Large (x)[3]. These models represent a high range of computational loads; specifically, the larger variants (m, l, x) require substantial floating-point operations (FLOPs) and memory bandwidth, which render them typically infeasible for direct deployment on resource-constrained edge devices without significant optimization or quantization [Okano et al. 2025, Rey et al. 2025]. The key specifications for these selected models can be seen in Table 1. It is crucial to distinguish between storage requirements and runtime memory constraints. The file sizes listed in Table 1 represent the serialized model weights (specifically, the standard PyTorch .pth format)[4]. However, the actual runtime memory footprint is larger, as the inference process requires additional allocation for uncompressed weight tensors, activation maps, and framework overhead.

To contextualize the storage requirements of the proposed Split Computing architecture, we compare the partition sizes with the monolithic baselines established in the work of [da Luz et al. 2025a]. For the YOLOv8x model, the standard monolithic weights require 136.9 MB of storage. In our ES configuration, the client-side partition occupies 118.2 MB, offering a modest reduction in on-device storage footprint. However, for deeper partitions, the storage requirement increases significantly; the LS for the

[3]Detailed specifications and performance benchmarks for each model variant are available in the official Ultralytics repository: `https://github.com/ultralytics/ultralytics`

[4]`https://pytorch.org/docs/stable/notes/serialization.html`

Model Variant	Total Layers	Full Model Load (GFLOPs)	Edge Partition Size (MB)		
			ES	MS	LS
YOLOv8m	169	79.3	45.4	55.0	84.5
YOLOv8l	209	165.7	75.8	98.6	145.5
YOLOv8x	209	258.5	118.2	153.9	227.1

Table 1. Model configurations and partition sizes for the three split scenarios defined by [Neduchal et al. 2024].

same model reaches 227.1 MB, surpassing the size of the original monolithic file. This increase is attributed to the overhead of serializing the split architecture as independent PyTorch modules, which lack the compression optimizations of a unified state dictionary. This comparison highlights that while ES optimizes both computational load and storage, deeper splits trade storage efficiency for reduced transmission bandwidth.

4.4. Implementation and Experimental Environment

To validate the proposed architecture, we deployed a concrete implementation emulating a multi-tenant smart parking environment.

The client-side role was assigned to low-cost TV Boxes featuring an Amlogic S905X2 SoC (Quad-core ARM Cortex-A53 @ 2.0 GHz) with 2 GB of RAM, running the Armbian operating system. These devices represent the extreme resource constraints typical of large-scale IoT deployments. The server was powered by an AMD Ryzen 5 7430U CPU with 36 GB of RAM, running the Ubuntu operating system and capable of handling concurrent inference requests.

The communication layer was implemented using gRPC (Google Remote Procedure Call). Unlike traditional REST APIs, gRPC utilizes Protocol Buffers (Protobuf), a binary serialization format that is significantly more compact and faster to parse than JSON. This choice minimizes the serialization overhead identified as a bottleneck in split computing literature. The system operates over a standard campus Wi-Fi network, introducing realistic jitter and bandwidth variations (average uplink: 6 Mbps).

To validate the comparisons statistically, each split-versus-baseline pair was matched by image name ($n = 10$ per scenario). Normality of paired differences was assessed via the Shapiro–Wilk test; paired t-tests or Wilcoxon signed-rank tests were applied accordingly ($\alpha = 0.05$). Confidence intervals (95%) were estimated by bootstrap resampling (20,000 iterations, seed $= 0$).

5. Results

This section discusses the results of the evaluation performed for the proposed Split Computing architecture. The evaluation was conducted to analyze three dimensions of performance: (i) Total Inference Latency, composed of client processing, network transmission, and server inference; (ii) Client-side Resource Usage, specifically CPU and RAM footprint; and (iii) Network Payload Size.

These metrics were selected to demonstrate the advantages of the split architecture. Latency analysis verify if the system meets the near real-time requirements of smart parking applications (typically under 10 seconds). Resource usage is analyzed to confirm

that offloading computationally intensive layers prevents the edge device from reaching saturation, thereby avoiding thermal throttling and ensuring system stability. Finally, payload analysis assesses the impact on network bandwidth, ensuring the solution is viable even in constrained wireless environments.

5.1. Inference and Communication Time

The lowest client-side inference times averaged around 3.5 seconds using the YOLOv8m model and an Early Split. Figure 4 illustrates the distribution and variance of client-side inference times across different model variants and split points. In general, more complex models incurred a multiplicative impact on inference duration, with YOLOv8l ranging from 6.5 to 13 seconds and YOLOv8x from 10 to 20 seconds, depending on the cut layer. Earlier cuts push more computation to the server, resulting in faster client-side execution.

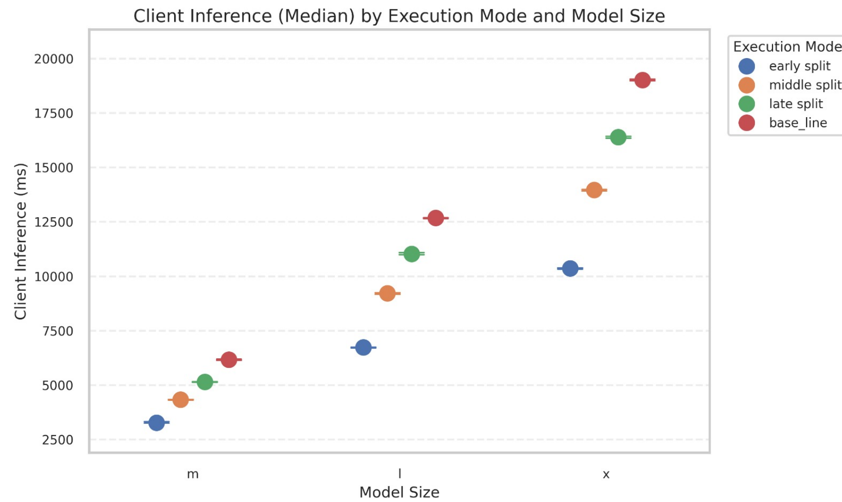

Figure 4. Average client inference time by split layer and model size.

Client inference dominated the total inference time, accounting for 73.5% to 86.2% of the total duration across all variants, while server inference was negligible, contributing less than 2.5%. Table 2 details this breakdown, showing the specific time allocation for computation versus the transmission of smashed data. The data confirms that larger models (l, x) impose a higher burden on the client, as their increased complexity results in a steep rise in processing delay that far outpaces the server-side costs.

Model	Client Inference		Communication		Server Inference		Total
	(ms)	(%)	(ms)	(%)	(ms)	(%)	(ms)
YOLOv8m	4440.30	73.53%	1440.29	23.85%	132.28	2.19%	6038.37
YOLOv8l	9462.90	84.02%	1645.53	14.61%	134.04	1.19%	11262.99
YOLOv8x	13568.29	86.20%	1916.34	12.18%	235.65	1.50%	15739.94

Table 2. Average inference time and component contribution per model.

For the purpose of this analysis, Communication Time is defined as the total latency incurred during the offloading process, encompassing two distinct phases: (1) the serialization of the intermediate tensor (smashed data) into a Protocol Buffer message, and (2) the network transmission of this payload via the gRPC channel. This metric was

calculated by measuring the duration of the client-side transmission request and subtracting the server-side processing time, effectively isolating the network and serialization overhead from the inference computation.

It is also worth highlighting the shifting impact of client-to-server communication in Split Computing scenarios. While the absolute time required to transmit data increases with model size, its relative percentage decreases significantly, dropping from 23.85% for YOLOv8m to 12.18% for YOLOv8x. This indicates that for larger models, the bottleneck is strictly computational. However, for the medium model, communication remains a major factor (consuming nearly a quarter of the runtime).

Additionally, to mitigate bandwidth saturation from feature expansion, a encoder-decoder architecture [Neduchal et al. 2024] could drastically compress intermediate feature maps before transmission. This approach reduces payload size without degrading inference accuracy. Furthermore, while gRPC provides a robust standard, investigating zero-copy serialization protocols could further lower the client's CPU burden from packet marshalling. These optimizations are targeted for future iterations, as this research aims to establish a baseline performance using standard, cost-effective technologies.

5.2. Baseline Deployment x Split Computing Strategy

To precisely quantify the benefits of the proposed Split Computing architecture, we evaluated our results against a direct baseline deployment where the full, monolithic YOLOv8 models were executed locally on the client edge devices (TV Boxes). As detailed in Table 3, the standalone, monolithic execution of the YOLOv8x model required 19.17 seconds to complete a single inference on the edge device. In contrast, our Split Computing implementation achieved a total inference time of 15.74 seconds, yielding a latency reduction of approximately 17.9% using the exact same hardware and model framework.

Similar trends were observed across the other model variants. The monolithic execution of YOLOv8l took 12.90 seconds, whereas the split architecture reduced this to 11.26 seconds (a 12.7% improvement). For the medium-sized variant (YOLOv8m), the monolithic baseline achieved an inference time of 6.26 seconds, which the split implementation improved to 6.04 seconds.

It is important to note that these split computing totals represent configurations that heavily favor client-side processing. By shifting to an Early Split, the client's computational burden is minimized even further, dropping the client-side inference phase to as low as 3.5 seconds for the YOLOv8m model. Ultimately, these direct comparisons prove that the Split Computing strategy effectively mitigates hardware limitations, enabling the deployment of complex models that would otherwise suffer from severe latency bottlenecks on standalone edge devices.

Model Variant	Monolithic Baseline	Split Computing Total	Latency Reduction
YOLOv8m	6258.2 ms	6038.4 ms	3.5%
YOLOv8l	12901.9 ms	11263.0 ms	12.7%
YOLOv8x	19171.7 ms	15739.9 ms	17.9%

Table 3. Performance comparison between Monolithic Baseline and Split Computing

Statistical analysis confirms that all nine scenarios yield significant latency reductions (all $p < 0.01$), with strictly negative 95% bootstrap confidence intervals (CIs), indicating that the true mean latency reduction falls below zero with high confidence. Gains range from -348 ms (YOLOv8m, Late Split) to $-7{,}350$ ms (YOLOv8x, Early Split), consistently demonstrating that Split Computing reduces end-to-end inference latency regardless of model size or partition point.

5.3. Client Resource Usage

Our inference framework implemented in this research, including the PyTorch and Ultralytics libraries as well as other necessary components, were able to be loaded into memory on both clients. Figure 5 illustrates the average client RAM usage during inference. The objective of this measurement is to validate the feasibility of deploying these models on highly constrained edge nodes, specifically the repurposed TV boxes which are limited to 2 GB of RAM. As shown, the YOLOv8m model demonstrated the lowest memory footprint at 600MB across all cut layers. Even under the most demanding scenario, the YOLOv8x model executing a late split, the memory consumption peaked at 900MB.

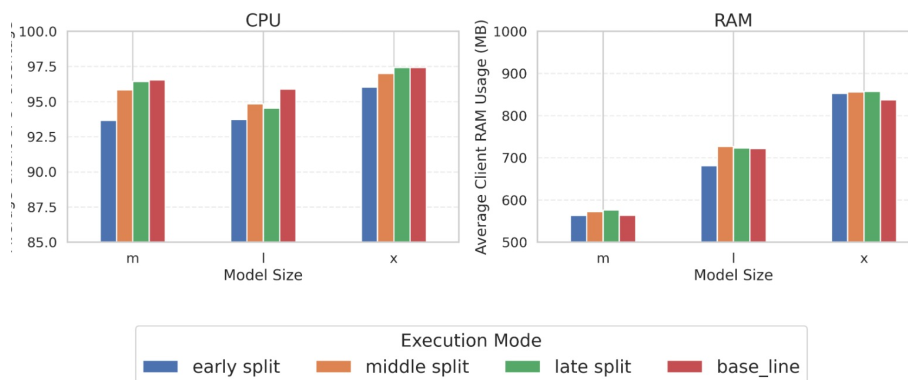

Figure 5. Edge/Client resource usage comparison.

Interestingly, the baseline monolithic executions occasionally exhibited slightly lower RAM usage than their partitioned counterparts. This counterintuitive behavior is attributed to the structural overhead introduced by the split architecture. When the neural network is serialized into independent PyTorch modules for the split, it lacks the compression optimizations of a unified state dictionary. Furthermore, the actual runtime memory footprint must temporarily expand to allocate uncompressed weight tensors, framework overhead, and the large activation maps (smashed data) generated before transmission. Despite this serialization overhead, this demonstrates a key contribution of the Split Computing architecture: it allows complex, memory-intensive deep learning frameworks and their dependencies to operate safely within the strict 2 GB hardware limits of ultra-low-cost devices without triggering out-of-memory errors or requiring swap space.

On the other hand, client CPU usage during the experiments were mostly the same across models and split layers, with the late split naturally incurring slightly more processing load, as can be seen in Figure 5. From the data, it appears that the model size does not influence the CPU load on the clients, as all models registered themselves to be in the range of 95% to 100% throughout. Because the Amlogic S905X2 SoC is highly resource-constrained, any deep learning workload immediately saturates its capabilities.

The slightly lower CPU usage observed for YOLOv8l is attributed to OS-level scheduling variability, which can introduce small variances in reported process utilization at near-saturation loads.

Crucially, while Split Computing offloads the execution of deeper neural network layers, it introduces a new CPU burden on the client: the serialization of the intermediate tensor (smashed data) into a Protocol Buffer message and its subsequent transmission. Consequently, the true computational benefit of offloading is not reflected as a drop in the CPU utilization percentage, which remains effectively maximized, but rather as the reduction in total inference execution time demonstrated in Section 5.2.

5.4. Payload Size

Finally, we analyzed network payload sizes during the experiments. To quantify this, the server intercepts the incoming gRPC transmission and records the exact byte size of the serialized Protocol Buffer message containing the smashed data.

Our measurements show that the payload size increases linearly with the model scale, recording 6.33 MB for YOLOv8m, 8.13 MB for YOLOv8l, and 10.16 MB for YOLOv8x. Crucially, these values remain constant across all three split points (Early, Middle, and Late) for each respective model. This occurs because, following the partitioning strategy defined by Neduchal et al. [Neduchal et al. 2024], the selected cut layers were specifically chosen to produce intermediate tensors of identical spatial dimensions. Consequently, in this specific experimental design, the choice of cut layer does not impact the bandwidth consumption, allowing us to isolate the computational trade-offs.

6. Conclusion and Future Work

This work presented the design, implementation, and experimental evaluation of a Split Computing–based inference approach using YOLOv8 models for vehicle detection in smart parking scenarios. The proposed method demonstrates clear advantages over fully client-side inference by offloading the most computationally intensive portions of the model to a server, thereby reducing the computational and energy burden on resource-constrained edge devices while avoiding the transmission of raw visual data. As a result, the approach enables the deployment of larger and more accurate detection models in environments where client devices alone would be incapable of executing them efficiently.

The experimental results indicate that Split Computing provides a practical and scalable solution for smart parking systems operating on low-powered IoT hardware, particularly in multi-tenant scenarios. By carefully selecting the model partition point, the framework balances computation and communication overhead, making state-of-the-art object detection feasible without requiring costly hardware upgrades at the edge.

Future work will focus on three directions. First, expanding the statistical evaluation to larger sample sizes ($n > 10$) to increase the power of significance tests and improve generalizability. Second, incorporating energy consumption metrics to quantify the sustainability gains of offloading on ultra-low-cost devices such as the repurposed TV boxes. Third, evaluating the architecture under variable network conditions, including 5G/MEC deployments and congested Wi-Fi scenarios with multiple concurrent clients, to assess the robustness of the identified computation-communication trade-offs beyond the controlled campus environment.

Acknowledgments

This work was supported by the S ao Paulo Research Foundation (FAPESP), grant 2021/00199-8 (CPE SMARTNESS), processes 2025/03795-1 and 2025/02093-3, and by the National Council for Scientific and Technological Development (CNPq).

References

Baggio, J. V., Gonzalez, L. F. G., and Borin, J. F. (2020). Smartparking a smart solution using deep learning. https://api.semanticscholar.org/CorpusID:227991127.

Chakroun, I., Vander Aa, T., Wuyts, R., and Verachtert, W. (2021). Distributing intelligence for object detection using edge computing. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 681–687.

da Luz, G., Sato, G., Bannwart, T., Gonzalez, L., and Borin, J. (2025a). 10 years of deep learning for vehicle detection at a smart parking : What has changed? In *Anais do IX Workshop de Computação Urbana*, pages 127–140, Porto Alegre, RS, Brasil. SBC.

da Luz, G. P. C. P., Sato, G. M., Gonzalez, L. F. G., and Borin, J. F. (2025b). Repurposing of tv boxes for a circular economy in smart cities applications. *Scientific Reports*, 15(1):22638.

Eshratifar, A., Abrishami, A., and Pedram, M. (2019). Bottlenet: A deep learning architecture for intelligent mobile cloud offloading. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE.

Gupta, O. and Raskar, R. (2018). Distributed learning of deep neural network over multiple agents. *Journal of Network and Computer Applications*, 116:1–8.

Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T. N., Mars, J., and Tang, L. (2017). Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of ASPLOS '17*, pages 615–629.

Marinova, M. and Rakovic, V. (2024). Accelerating convergence in split learning for time-varying and resource-limited environments. In *2024 IEEE 22nd Mediterranean Electrotechnical Conference (MELECON)*, pages 13–18.

Matsubara, Y. and Levorato, M. (2020). Split computing for complex object detectors: Challenges and preliminary results. In *Proceedings of the 4th ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges (VideoAI '20)*.

Matsubara, Y. and Levorato, M. (2022). Split computing and early exiting for deep learning applications: Survey and research challenges. *ACM Computing Surveys*.

Nations, U. (2015). Transforming our world: the 2030 agenda for sustainable development. https://wedocs.unep.org/20.500.11822/9824.

Neduchal, P., Straka, J., Sieber, M., and Gruber, I. (2024). Comparison of split computing scenarios for object detection. In *IFAC-PapersOnLine (18th IFAC Conference on Intelligent Manufacturing Systems)*, pages 120–125.

Okano, M. T., Lopes, W. A. C., Ruggero, S. M., Vendrametto, O., and Fernandes, J. C. L. (2025). Edge AI for industrial visual inspection: YOLOv8-based visual conformity detection using raspberry pi. *Algorithms*, 18(8).

P C P da Luz, G., Massuyoshi Sato, G., Fernando Gomez Gonzalez, L., and Freitag Borin, J. (2026). Smart parking with pixel-wise ROI selection for vehicle detection using yolov8, yolov9, yolov10, and yolov11. *Internet of Things*, 36:101858.

Rey, L., Bernardos, A. M., Dobrzycki, A. D., Carramiñana, D., Bergesio, L., Besada, J. A., and Casar, J. R. (2025). A performance analysis of you only look once models for deployment on constrained computational edge devices in drone applications. *Electronics*, 14(3).

Sato, G. M., da Luz, G. L. N., Gonzalez, L. F. P., and Borin, J. (2024). Reaproveitamento de tv boxes para aplicação de contagem de pessoas na borda em cidades inteligentes. *Anais do VIII Workshop de Computação Urbana (CoUrb 2024)*.

Shoup, D. (2021). Pricing curb parking. *Transportation Research Part A: Policy and Practice*, 154:399–412.

Wu, W., Li, M., Qu, K., Zhou, C., Shen, X., Zhuang, W., Li, X., and Shi, W. (2023). Split learning over wireless networks: Parallel design and resource management. *IEEE Journal on Selected Areas in Communications*, 41(4):1051–1066.

Yao, J. (2023). Split learning for image classification in internet of drones networks. In *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*, pages 52–55.

Zhang, L., Chen, L., and Xu, J. (2021). Autodidactic neurosurgeon: Collaborative deep inference for mobile edge intelligence via online learning. In *Proceedings of The Web Conference (WWW) 2021*, pages 3111–3123.