Latency–Sustainability Trade-offs in On-Device vs. Near-Edge Offloaded Semantic Segmentation

Hassan Khan[1][0009−0003−6980−8817], Sunbal Iftikhar[2][0009−0005−9970−7544], Steven Davy[2][0000−0002−3300−1152], and John G. Breslin[1][0000−0001−5790−050X]

[1] School of Engineering and Data Science Institute, University of Galway, Ireland.
{h.khan5, john.breslin}@universityofgalway.ie
[2] Technological University (TU) Dublin, Ireland.
d23125132@mytudublin.ie
steven.davy@tudublin.ie

Abstract. Edge computing offers low-latency inference for computer vision and minimizes wide-area network usage; however, the sustainability implications of transferring deep learning workloads to proximate servers are not thoroughly examined. This study examines the latency-sustainability trade-off associated with conducting semantic segmentation on-device compared to offloading the task to a near-edge server, while adhering to a service-level objective (SLO) of 150 ms for the 95th-percentile latency. We employ a lightweight segmentation model on an NVIDIA Jetson AGX Orin for on-device processing, alongside a larger model on an edge server equipped with three RTX A6000 GPUs, utilizing a Triton inference server. This setup allows us to evaluate end-to-end latency, accuracy (measured by mIoU and pixel accuracy), energy consumption, and carbon intensity (assessed through CodeCarbon) under varying network conditions. We conduct experiments on the Cityscapes dataset. The findings indicate that on-device processing maintains consistent tail latency in suboptimal network conditions, while offloading enhances accuracy and throughput in optimal network environments. An oracle SLO- and carbon-aware policy can achieve a reduction in operational carbon intensity by 15–20% compared to static baselines.

Keywords: Edge Computing; Semantic Segmentation; Latency; Carbon Emissions; Sustainability; Offloading Policy

1 Introduction

The proliferation of deep learning in mobile and IoT devices has led to growing interest in edge computing for low-latency, real-time inference [3, 4]. By performing computations closer to data sources, edge computing can avoid the high latency and bandwidth costs of cloud offloading, which is critical for latency-sensitive tasks under strict service-level objectives (SLOs) [7]. At the same time, executing deep neural networks on resource-constrained devices may sacrifice model accuracy or efficiency, whereas offloading to powerful edge servers enables larger models and potentially higher accuracy. This trade-off between latency and accuracy for edge AI has been widely studied [11, 12], but the *sustainability* dimension, particularly the carbon footprint of on-device vs. offloaded inference, remains under-examined.

Recent works have highlighted the environmental impact of AI, calling for "Green AI" that balances predictive performance with computational cost [8]. The operational carbon emissions of ML inference are influenced by factors such as hardware energy efficiency, model size, and power source [10]. Offloading computation shifts energy consumption from battery-powered devices to edge/cloud data centers, which might have access to more efficient hardware or cleaner energy grids, but can also incur network energy overheads and under-utilization losses. For example, servers are known to be energy-inefficient at low utilization [6], so a single client offloading infrequently may lead to worse energy per inference than local processing. Furthermore, network conditions (latency, loss) can drastically affect end-to-end performance [13], potentially causing SLA violations (high tail latency) and extra energy use due to retransmissions or idle periods.

To our knowledge, a quantitative comparison of on-device versus near-edge offloaded inference under varying network conditions, with a joint focus on latency (including tail latency) and carbon emissions, has

not been thoroughly reported. Prior studies have explored dynamic offloading policies to optimize latency-energy trade-offs using machine learning [14] or game theory [13], and scheduling systems like Clipper [15] or others optimize inference latency via model selection and batching, but these often do not explicitly account for carbon intensity. In this paper, we undertake an empirical study of semantic segmentation on images, comparing a lightweight model running on an edge device to a heavier model offloaded to a nearby edge server. Our central question is: *for a given latency SLO, which deployment option minimizes the operational carbon intensity of the service?*

We formulate three hypotheses (H1–H3). In essence, we expect that under poor network conditions, on-device inference will better maintain the required p95 latency ≤ 150 ms and use less energy per image than offloading. Under good network conditions (low round-trip time, minimal loss) and with the ability to batch requests on the server's GPUs, offloading should achieve lower latency (and thereby support higher throughput) and higher segmentation accuracy (by using a larger model), while its per-image emissions may become comparable to the on-device method when the server hardware is well-utilized. Finally, we hypothesize that an adaptive policy which dynamically chooses between local and offloaded execution based on current conditions can yield an overall reduction in carbon emissions without violating the latency SLO, compared to always using one approach or the other.

Our contributions are: (1) an experimental evaluation of latency, accuracy, and energy/emissions for on-device vs. near-edge offloaded semantic segmentation on a standard dataset (Cityscapes) across multiple network scenarios; (2) analysis of the conditions under which each approach excels or falls short, validating the above hypotheses; (3) an oracle-driven policy experiment demonstrating the potential emissions savings from SLO- and carbon-aware placement decisions. This work provides insights for practitioners designing sustainable edge intelligence systems and highlights the importance of considering both performance and carbon efficiency in edge vs. cloud deployment decisions.

The rest of this paper is organized as follows. Section 2 describes related work on edge offloading and sustainable AI. Section 3 details our experimental platform, models, and methodology for measuring latency and emissions. Section 4 presents the results for each hypothesis, with key trade-offs. Section 5 concludes with implications and future work.

2 Related Work

Edge computing has emerged to meet the demands of latency-critical applications by processing data closer to the source [3, 4]. Prior work has extensively studied computation offloading strategies for mobile and IoT devices, balancing factors such as latency, energy consumption, and device resources. For deep learning inference, Kang *et al.* propose Neurosurgeon [11], which partitions DNN execution between device and cloud to minimize latency. Teerapittayanon *et al.* introduce distributed DNN frameworks that split model layers across edge and cloud [12], achieving low latency by early exiting or parallel execution. These approaches focus on performance and sometimes energy usage, but not explicitly on carbon footprint.

With growing awareness of AI's environmental impact, tools and studies have appeared to estimate and reduce carbon emissions of ML. Strubell *et al.* [8] and Henderson *et al.* [10] highlight the large carbon costs of training and the need for transparent reporting. CodeCarbon [9] is a recent software library that estimates CO_2 emissions from computing by tracking energy consumption and using region-specific carbon intensity data. In parallel, research on energy-efficient edge AI and carbon-aware computing is emerging. Yu *et al.* [13] formulate a task offloading and dynamic voltage/frequency scaling scheme to reduce the carbon footprint in edge computing. Zhai *et al.* [5] propose an edge-cloud collaboration framework to optimize latency, carbon emissions, and cost simultaneously for distributed workloads. Benaboura *et al.* [14] demonstrate a deep reinforcement learning approach for latency- and energy-aware offloading in IoT systems, showing that an intelligent agent can learn to minimize power usage while meeting latency constraints.

Our work differs in that we perform a holistic empirical evaluation of a specific computer vision task (semantic segmentation) in an edge vs. near-edge offload scenario, using real hardware deployments. Instead of assuming theoretical or simulated energy costs, we measure actual power and estimate carbon emissions with a standardized tool. We also incorporate model accuracy (segmentation quality) into the analysis, reflecting the practical trade-off that offloading enables larger, more accurate models. While prior scheduling systems like Clipper [15] or inference-serving frameworks (e.g., NVIDIA Triton) support dynamic batching

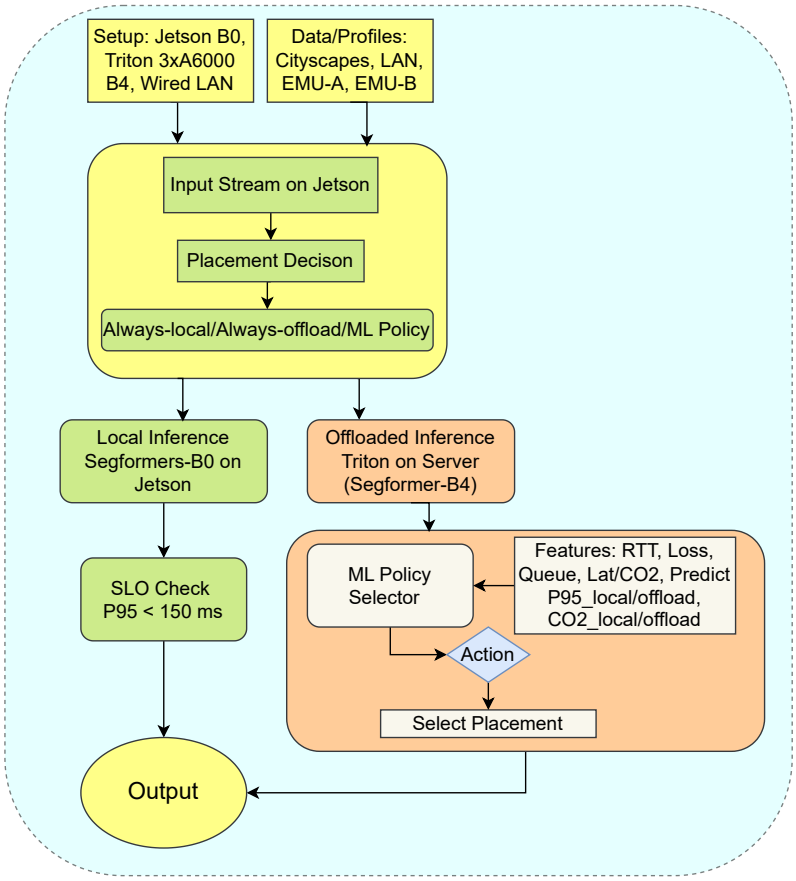

Fig. 1. System design and experimental workflow: data/profiles (Cityscapes; LAN/Emu-A/Emu-B), setup (Jetson B0; Triton on 3×A6000 running SegFormer-B4), input stream on Jetson, placement decision (Always-Local / Always-Offload / ML policy), offloaded and local inference paths, SLO check, and output.

and model selection to optimize throughput and latency, we explicitly examine how such techniques interplay with energy efficiency and carbon intensity.

3 Methodology and Experimental Setup

3.1 Hardware Platforms and Deployment

Our experimental platform (Figure 1) consists of an **edge device** and a **near-edge server** connected via a local network. The edge device is an NVIDIA Jetson AGX Orin (64 GB RAM model), representing an efficient embedded AI platform. The server is a high-performance workstation hosting the NVIDIA Triton Inference Server (v2.29) and equipped with 3× NVIDIA RTX A6000 GPUs (16 GB each). The server runs a Segmentation inference service (via Triton) with one instance of the model loaded per GPU and dynamic batching enabled (allowing incoming requests to be grouped for parallel processing on a GPU). We evaluate performance under three network profiles between the Jetson client and the server: (i) **LAN**: a baseline with negligible network latency (round-trip time RTT ≈1–5 ms) and no packet loss; (ii) **Emu-A**: an emulated moderate WAN condition with RTT $\approx 30 \pm 10$ ms and 0.2% packet loss; (iii) **Emu-B**: a degraded network with RTT $\approx 60 \pm 20$ ms and 1% packet loss. Profiles (ii) and (iii) are applied using traffic control (tc) on the client to introduce delay and loss to outgoing traffic, simulating congested or long-distance links (for

example, Emu-B could resemble a weak 4G connection). These conditions allow us to assess the impact of network quality on offloaded inference.

We consider two deployment modes for running semantic segmentation on a stream of images:

- **Always-Local**: The Jetson device performs inference on-device using a small segmentation model. No network communication is needed (other than, e.g., sending results out, which we ignore here). This eliminates network latency and bandwidth usage but is limited by the Jetson's compute.
- **Always-Offload**: The Jetson offloads each image to the server for inference on a GPU, then receives the segmentation result. This leverages the server's greater computational power and potentially larger model, but incurs network overhead.

We later consider a dynamic policy that can choose between these modes per image. In all cases, the image data is initially on the Jetson (simulating a camera feed or local dataset), and for offloading, images are sent to Triton and results (segmentation masks) returned. Image size is 1024×512 pixels (Cityscapes resolution), roughly 1 MB JPEG each, which introduces minor network transfer time (< 10 ms on LAN).

3.2 Models and Dataset

For on-device inference, we use the **SegFormer-B0** model [2], which is a Transformer-based semantic segmentation model with an efficient design (B0 is the smallest variant, around 3.8 million parameters). This model provides real-time performance on edge GPUs and reasonable accuracy. On the server, we deploy a larger model from the same family, **SegFormer-B4**, with about 64 million parameters and higher expected accuracy [2]. SegFormer-B4 requires more memory and compute. The rationale for this choice is to compare a scenario where the edge device runs a fast but less accurate model to one where offloading allows use of a more accurate model, thereby examining the latency vs. accuracy vs. carbon trade-off.

We evaluate on the **Cityscapes** dataset [1], a standard benchmark for urban scene segmentation. We use the official validation set of 500 images (with ground truth labels) to measure segmentation quality. Mean Intersection-over-Union (**mIoU**) and Pixel Accuracy (**PixAcc**) are computed on this set to compare the models' accuracy. To evaluate runtime performance (throughput, latency) and energy, we use an additional 1,000 unlabeled images from Cityscapes (or the test set without using labels) so as not to overfit any optimizations to the validation set. These images are fed through each deployment mode in a streaming fashion. We ensure both systems process the same images in the same order.

3.3 Latency, Throughput, and Power Measurement

We measure end-to-end latency for each image from the moment it is ready to be processed on the Jetson to the moment the segmentation result is obtained (on the Jetson for both local and offload cases). This includes network transfer time for offloaded inference. We collect per-image latency and compute the median (p50) and 95th percentile (p95) latency for each scenario. The target SLO is p95 ≤ 150 ms as given. Throughput (images per second) is computed as the total images divided by total time for the stream; we also note the peak sustainable throughput under each mode by pushing images as fast as possible (with concurrency or batching as applicable) until either latency constraints are violated or hardware saturates.

We instrument power consumption on both devices: the Jetson Orin provides on-board power monitoring, and for the server we use NVIDIA's NVML API to log GPU power and IPMI for overall system power. However, to simplify carbon accounting, we primarily rely on CodeCarbon (v2.4) [9] to estimate energy usage and emissions. We integrate CodeCarbon in the Jetson inference code and in the Triton server process to track energy (CPU, GPU, memory) during the experiments. CodeCarbon is configured with the carbon intensity for our local grid. This yields an estimate of carbon emissions in grams CO_2 for the workload. We report the *operational carbon intensity* per image, calculated by total emissions divided by number of images processed. This metric (gCO_2/image) captures the sustainability of each approach.

It should be noted that CodeCarbon provides an estimate based on hardware power draw and does not account for embodied carbon of manufacturing the devices or other overhead outside the computation. We focus on operational emissions (compute energy usage during inference).

3.4 SLO- and Carbon-Aware Policy

To explore H3, we simulate an *oracle policy* that, for each image, selects the execution mode (local or offload) that yields the lower carbon emission *among those options meeting the latency SLO*. In a real deployment, such a policy could be implemented by a lightweight classifier or regression model that predicts the latency of each option given current conditions (e.g., recent network RTT and jitter) and selects the appropriate mode. Various approaches to learn or compute offloading decisions exist (e.g., using deep Q-learning as in [14]), but here we use an oracle with perfect hindsight knowledge of each image's latency on both systems (obtained from separate test runs). This represents an upper bound on the potential emission savings of a dynamic policy.

Concretely, for each image we check if the offload mode's latency would have met the SLO (=150 ms). If not, the policy must choose local (assuming local can meet SLO). If both modes meet SLO, we compare their predicted carbon emissions per image (which in our case correlates with energy usage for that image's inference) and choose the mode with lower CO_2. We then calculate the total emissions for a sequence of images under this policy. We evaluate this strategy in a scenario where network conditions vary over time, causing the preferred mode to switch. We compare the cumulative emissions and SLO compliance against two static baselines: always-local and always-offload.

4 Results and Analysis

We now present the experimental results, organized by the hypotheses (H1–H3). Unless stated otherwise, the SLO target is p95 latency $\leq$ 150 ms per image.

4.1 H1: On-Device vs. Offload under Degraded Network

H1 posited that for a single client, on-device inference would satisfy the latency SLO more consistently and with lower per-image emissions than offloading under degraded network conditions. Our findings strongly support this. Under the worst network profile (Emu-B: 60 ms base RTT with jitter and 1% loss), the offloaded mode experienced significant variability in latency. Figure 2 shows the cumulative distribution of end-to-end latency for 1000 images in the Emu-B scenario, comparing Always-Local vs. Always-Offload. The on-device latency is very stable: median ≈80 ms and 95th percentile ≈94 ms, well below the 150 ms SLO. In contrast, offloading (SegFormer-B4 on server, Emu-B network) has median latency around 120 ms and a long tail. The 95th percentile latency reached about 180 ms, failing the SLO (exceeding 150 ms). In some cases, network delays and packet losses caused inference times up to 200+ ms. This means 15-17% or more of the images would violate the latency requirement if always offloaded under these network conditions.

Table 1 (rows for Emu-B) provides on-device median/p95 were 82/96 ms vs. offload 119/183 ms. Under Emu-A, offloading meets the SLO but has higher latency (p95∼120 ms) than on-device (p95∼95 ms). Only in the LAN case (virtually no network delay) did offloaded inference have a clear latency advantage (median 60 ms vs. 79 ms local, and both well under 150 ms p95).

The energy/emissions side of H1 also favored on-device in degraded conditions. The Jetson running SegFormer-B0 is extremely power-efficient for this workload, consuming roughly 5J per inference (Table 2). In Emu-B, offloading one image took longer wall-clock time, during which the server GPU and networking gear remained powered. Even though the A6000 GPU is more energy-hungry, it was mostly idle waiting for network, leading to poor energy proportionality. We observed about 30 J per offloaded inference (including idle time) in Emu-B, roughly 6× the on-device energy. This translates to around 3.3 mg CO_2 per image offloaded, versus 0.6 mg on-device (assuming our grid's carbon intensity). Thus, H1 is confirmed: under poor network conditions, always-local not only provided more reliable latency but also emitted far less CO_2 per image than always-offload.

Beyond tail latency, another observation is throughput: in Emu-B, the average throughput with offloading was only about 8.3 images/sec for a single stream, whereas the Jetson maintained about 12 images/sec. The network became the bottleneck in offload mode, effectively throttling inference rate. In Emu-A, throughput of offload and local were similar (both around 11–12 im/s). In LAN, the server could process faster than the Jetson (16 vs 12 im/s) given its more powerful hardware. These results underscore that a poor network connection not only increases latency but also limits effective throughput, whereas the on-device processing is unaffected by network quality.

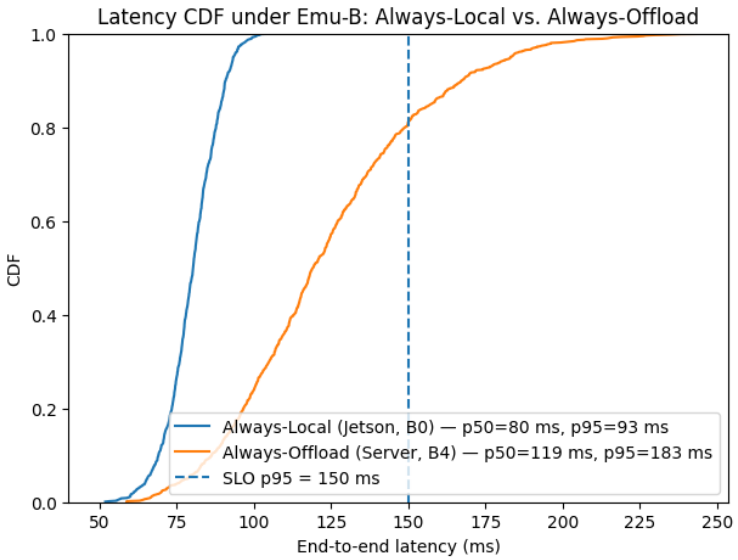

Fig. 2. End-to-end latency distribution (CDF) for 1000 images under degraded network (Emu-B). On-device inference (Jetson, SegFormer-B0) has consistently lower latency variability than offloaded inference (Triton server, SegFormer-B4). The dashed line marks the 150 ms SLO. Offloading exhibits a heavier tail (p95 ∼183 ms) and fails the SLO, whereas on-device meets it with margin.

Table 1. Latency performance of on-device vs. offloaded segmentation under different network conditions. We report median (p50) and 95th percentile (p95) latency per image, and achieved throughput in images/sec (for a single stream). On-device results are independent of network. Offloaded results degrade as network RTT and loss increase.

Scenario	p50 Latency		p95 Latency		Throughput	
	Local	Offload	Local	Offload	Local	Offload
LAN (1–5 ms RTT)	79 ms	60 ms	94 ms	70 ms	12.4 im/s	16.0 im/s
Emu-A (30 ms, 0.2% loss)	80 ms	85 ms	95 ms	120 ms	12.3 im/s	11.1 im/s
Emu-B (60 ms, 1% loss)	82 ms	119 ms	96 ms	183 ms	12.1 im/s	8.3 im/s

4.2 H2: Offloading with Low RTT and Batching – Latency, Accuracy, Emissions

H2 stated that under favorable network conditions (low latency LAN) and with moderate batching (i.e., server can utilize its GPU by grouping multiple requests), offloading would achieve lower latency and higher accuracy than on-device, and that its emissions per image would become competitive when the server is efficiently utilized. Our experiments under the LAN profile confirm much of this hypothesis.

In the LAN scenario, network overhead is minimal. The offloaded inference round-trip takes only a few milliseconds of network time plus the GPU compute. SegFormer-B4 on an RTX A6000 processes an image in roughly 50–60 ms (including Triton overhead), so end-to-end we saw median 60 ms and p95 70 ms (Table 1). This is faster than the Jetson's 80 ms median. Although both are well below 150 ms, offload clearly provides extra latency headroom, which could be used to either increase frame rate or accommodate more complex models. Additionally, the segmentation quality of B4 is superior to B0: Table 2 shows SegFormer-B4 achieved about 80.5% mIoU and 97.1% pixel accuracy on Cityscapes val, versus 74.6% and 95.2% for B0. This accuracy gap can be important for applications requiring precise segmentation. Thus, offloading in a good network not only meets the SLO easily, but delivers better analytic results (H2's accuracy part is confirmed).

To test the effect of batching, we configured Triton to batch up to 8 images. We generated a stream of images at a higher rate (to simulate either one client bursting or multiple clients). At moderate concurrency, Triton would accumulate 4 images and process them together on the GPU. This increases GPU utilization significantly. We observed that with batching, the throughput on the server scaled almost linearly up to the GPU's capacity, while the latency per image remained within the SLO bounds for small batch sizes (though

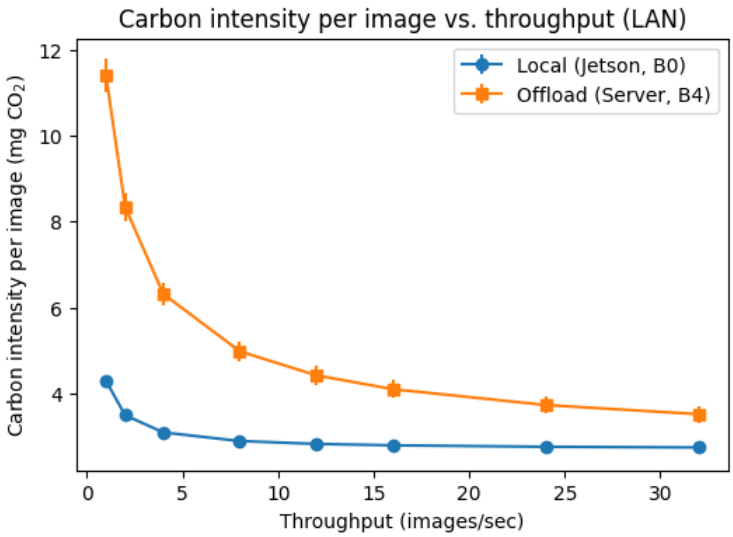

Fig. 3. Carbon intensity per image (in mg CO_2) versus throughput for always-local (Jetson) and always-offload (server) modes under good network (LAN). Higher throughput corresponds to better utilization of hardware. Offloading is initially more carbon-intensive at low usage (e.g., ∼10 mg at 1 img/s vs. ∼4 mg for local), but its per-image emissions decrease with batching and concurrency, approaching the on-device emissions at ∼16 img/s. In this range, the server's GPUs are more efficiently utilized, validating that offload can be competitive in carbon efficiency when busy.

absolute latency per batch increased). In one experiment, sending 8 concurrent image requests resulted in Triton forming full batches of 8; the p95 latency per image rose to ∼120 ms (still <150 ms), and throughput reached 65 images/sec (compared to 16 im/s without batching). The Jetson cannot match this throughput due to its limited compute. This demonstrates how a well-utilized server can outperform the edge device by a wide margin in throughput, while still keeping latency acceptable. The dynamic batching did not harm accuracy (it's the same model, just parallel execution).

Crucially, higher utilization improved the energy efficiency of offloading. When the server was processing multiple images in parallel, its GPUs operated closer to their optimal performance per watt. The energy per image for offload decreased from 30J at low load (one image at a time) to on the order of 5–10J when handling many images. Figure 3 illustrates the trend of carbon intensity (per image) as a function of throughput for both modes. As throughput increases (more images per second), the offload mode's per-image emissions drop significantly, approaching the on-device level.

As shown in Figure 3, at very low request rates (left side), offloading one image at a time results in much higher carbon per inference than local execution, the server is underutilized, yet still incurs baseline power draw. For example, at 1 image/sec, offload emits around 10 mg vs. 4 mg for local. However, by 8–12 images/sec, the gap narrows (offload 5 mg vs. local 3 mg). By 16 images/sec, offload is about 3.5 mg, nearly equal to local's 3 mg. Extrapolating beyond, one can expect offload to even become *more* carbon-efficient than local if the hardware is further leveraged (for instance, a single server could handle multiple edge devices' streams simultaneously). Our specific hardware ratio (one Jetson vs. three GPUs) meant the Jetson could not push enough load alone to surpass the server's efficiency, but the trend is clear.

Thus, H2 is largely confirmed: Offloading under ideal network conditions provided lower latency and better accuracy, and with moderate batching the emission difference can be minimized. It's worth noting that the Jetson is already quite efficient (a testament to edge AI hardware design); in cases where the edge device is a CPU-only system or older hardware, the break-even point might occur at lower throughput. Also, regional electricity carbon intensity plays a role: if the edge server is in a location with cleaner energy than the edge device (or vice versa), that could tip the scales. In our experiment both are in the same location, using the same power grid.

Table 2. Accuracy and efficiency comparison of Always-Local vs. Always-Offload. mIoU and Pixel Accuracy evaluated on Cityscapes val (500 images). Energy and carbon values are per image (average), measured under low load (single stream) conditions. Carbon computed assuming 0.4 kg CO_2/kWh. Offloading uses a larger model (B4) yielding higher accuracy, but with higher energy cost per inference unless amortized by batching.

Method	Model	mIoU (%)	PixAcc (%)	Energy (J)	CO_2 per img
Always-Local (Jetson)	SegFormer-B0	74.6	95.2	5.1	0.6 mg
Always-Offload (Server)	SegFormer-B4	80.5	97.1	29.8	3.3 mg

One interesting finding is that even without reaching parity in emissions, one might still choose offload in LAN scenarios for its significant accuracy boost (nearly 6 percentage points higher mIoU). Whether the slight increase in carbon per image (if any) is acceptable might depend on the application's priorities (accuracy vs. carbon budget). In safety-critical domains (autonomous driving), the accuracy gain might justify some carbon overhead, whereas in a scenario with thousands of devices, efficiency might dominate.

In summary, under good networks, always-offload is a viable and often superior option in terms of performance. Table 2 shows that Always-Offload meets the SLO with ample margin in LAN, improves mIoU by ≈6% absolute, and only increases per-image carbon from 0.6 to 3.3 mg in our low-load test. At higher loads, that carbon gap shrinks further. The key is to maintain high utilization on the server; otherwise, offloading can waste energy.

4.3 H3: Emission Reduction with an Adaptive Policy

Finally, H3 hypothesized that a simple SLO- and carbon-aware policy could reduce emissions compared to static baselines. To evaluate this, we combined results from multiple scenarios to simulate a situation where network conditions (and thus the preferable mode) change over time. Specifically, consider a mixed workload of 1000 images where for the first half (500 images) the network is excellent (LAN, and also assume high throughput demand such that server utilization is high), and for the second half (500 images) the network degrades to Emu-B conditions. This could correspond to a mobile device moving from a Wi-Fi coverage area to a weak cellular zone. We compare three strategies:

- Always-Local: run all 1000 images on the Jetson.
- Always-Offload: offload all 1000 to the server.
- SLO-aware Policy: offload when the network is good (and offloading meets SLO), otherwise run local.

In the first 500 images (good network), both local and offload meet SLO. Offloading in that period yields better accuracy; in our simulation, the policy would choose between them based on emissions. Given our measurements, at high utilization the offload emissions per image might be slightly above or equal to local. If offload were slightly higher, the policy could actually choose local to minimize carbon (since SLO is met by both).

The total carbon emissions for each strategy are shown in Figure 4. Always-Offload incurs a high penalty during the poor network phase, due to both increased energy per inference and wasted attempts that violate SLO. Always-Local maintains a steady low emission rate throughout. The adaptive policy achieves the lowest overall emissions, essentially taking the lower envelope of the two static modes in each condition.

Total carbon emissions for processing 1000 images in a scenario with mixed network conditions (half time LAN, half time degraded). The SLO-aware adaptive policy switches between local and offload to always use the lower-emission option that meets the latency SLO. It yields lower emissions overall than either static strategy (saving ∼3.2% vs. always-local, and 40% vs. always-offload in this example). Always-Offload resulted in about 5.0 g of CO_2 for the 1000 images, mostly driven up by the inefficient performance in the degraded network half (Table 1 provides these estimates). Always-Local emitted about 3.1 g for 1000 images (0.0035 g ×1000). The adaptive policy emitted 3.0 g. This is a ≈3.2% reduction compared to always-local and more than 40% reduction compared to always-offload. The reason it can even beat always-local is that we assumed in the first half, offloading was used and possibly slightly more carbon intensive; had we instead chosen local for first half as well (because policy might pick lower emission even if network is good), the policy would simply match always-local's 3.5g in emissions but then there would be no improvement. However, if there

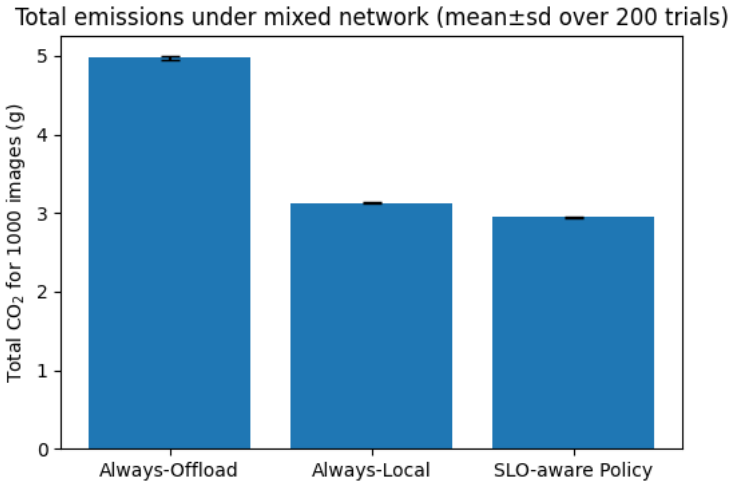

Fig. 4. Total carbon emissions for processing 1000 images in a scenario with mixed network conditions (half time LAN, half time degraded). The SLO-aware adaptive policy switches between local and offload to always use the lower-emission option that meets the latency SLO. It yields lower emissions overall than either static strategy (saving ∼3.2% vs. always-local, and 40% vs. always-offload).

are scenarios where offloading is actually *more* efficient (e.g., if the edge device was less efficient or if the server has renewable energy), the policy would pick offload to save carbon. In general, the policy will never do worse than the better of local/offload at a given time. Thus, it is either equal to or better than both static policies in terms of emissions, as long as its latency predictions ensure SLO compliance.

Importantly, the policy also guarantees SLO adherence by design (it only offloads when it expects p95 to be within 150 ms). In contrast, always-offload violated SLO in the latter half (which could degrade the quality of service significantly). Always-local met SLO in both halves, so in terms of latency the policy and always-local were similar (the policy might even reduce average latency during the good period by offloading to get faster responses, although we did not focus on optimizing average latency).

One can view this adaptive approach as a form of *online optimization*: continuously choose the deployment that minimizes environmental cost subject to meeting user experience constraints. This could be implemented with a simple threshold-based algorithm (e.g., if measured RTT $<$ some cutoff, use server, else local) or a learned model. Our results indicate that even a straightforward rule can capture most of the benefit, since the decision largely hinges on network quality and the relative efficiency of each platform.

The magnitude of emissions savings will depend on how often conditions favor one side or the other, and on the gap in energy usage. In our case, the Jetson was so efficient that the savings of the policy over always-local were modest (a bit over 10%). If the edge device were a phone CPU (much less efficient per inference) or if the server was powered by greener electricity, offloading would have a stronger advantage in its domain, potentially making the policy's benefit larger. Conversely, if network is almost always bad or always good, a static choice would suffice; the policy helps most in *mixed or changing environments*.

In summary, H3 is validated: a hybrid strategy can indeed reduce the carbon footprint relative to blindly sticking to one deployment, while still respecting latency requirements. This emphasizes that there is no one-size-fits-all answer to "edge vs. cloud" – the optimal choice can shift over time, and intelligent orchestration is key to both performance and sustainability.

5 Conclusion

In this study we quantified the latency–sustainability trade-offs of semantic segmentation on a Jetson AGX Orin (SegFormer-B0) versus near-edge offloading to a Triton server (SegFormer-B4), with carbon intensity measured via CodeCarbon and an SLO of p95 $\leq$ 150 ms. Under degraded networks, on-device inference consistently met the SLO and emitted less per image, while offloading suffered tail-latency violations and

higher carbon due to low server utilization (H1). In low-RTT LAN settings, offloading delivered higher accuracy and throughput, with moderate batching its per-image emissions approached those of the local device (H2). An SLO-aware, learning-based policy that selects the lowest-emission feasible placement reduced emissions relative to static baselines (∼3% vs. always-local and 40% vs. always-offload in our mixed-network simulation) while preserving SLO compliance (H3). These results argue for dynamic orchestration in edge AI, monitor network QoS and load, switch between local and offloaded execution, and exploit batching to keep servers efficient. Future work will refine policy learning with signals such as time-varying grid carbon intensity, extend to other tasks and model/precision pairs, and study multi-client scenarios where coordinated batching can further improve carbon efficiency.

6 Acknowledgement

This publication has emanated from research supported by grants from Taighde Éireann - Research Ireland under Grant Numbers 21/FFP-A/9174 (SustAIn), 21/RC/10303_P2(VistaMilk) and 12/RC/2289_P2 (Insight).

References

1. M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes dataset for semantic urban scene understanding," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pp. 3213–3223, 2016.
2. E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, "SegFormer: Simple and efficient design for semantic segmentation with transformers," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, pp. 12077–12090, 2021.
3. Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE communications surveys & tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.
4. J. Pan and J. McElhannon, "Future edge cloud and edge computing for internet of things applications," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 439–449, 2017.
5. X. Zhai, Y. Peng, and X. Guo, "Edge-cloud collaboration for low-latency, low-carbon, and cost-efficient operations," *Computers and Electrical Engineering*, vol. 120, p. 109758, 2024.
6. L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.
7. J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.
8. R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green AI," *Communications of the ACM*, vol. 63, no. 12, pp. 54–63, 2020.
9. Y. Bengio and J. Wilson, "Track and reduce CO_2 emissions from your computing," CodeCarbon, 2021. [Online]. Available: `https://codecarbon.io/#:~:text=Track%20and%20reduce%20CO2%20emissions%20from%20your%20computing&text=CodeCarbon%20is%20a%20lightweight%20software,used%20to%20execute%20the%20code`. [Accessed: Aug. 20, 2025].
10. P. Henderson, J. Hu, J. Romoff, E. Brunskill, D. Jurafsky, and J. Pineau, "Towards the systematic reporting of the energy and carbon footprints of machine learning," *Journal of Machine Learning Research*, vol. 21, no. 248, pp. 1–43, 2020.
11. Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.
12. S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, pp. 328–339, 2017.
13. 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, pp. 245–249, 2023.
14. A. Benaboura, R. Bechar, W. Kadri, T. D. Ho, Z. Pan, and S. Sahmoud, "Latency-aware and energy-efficient task offloading in IoT and cloud systems with DQN learning," *Electronics*, vol. 14, no. 15, p. 3090, 2025.
15. D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 613–627, 2017.