SMOOTH: Scalable Multitask Offloading with Backbone Sharing

Wei Geng*, Xiang Su†, Nitinder Mohan‡, Jörg Ott*, Pan Hui§¶†

*Technical University of Munich, Munich, Germany

†University of Helsinki, Helsinki, Finland

‡Delft University of Technology, Delft, Netherlands

§The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China

¶The Hong Kong University of Science and Technology, Hong Kong, China

wei.geng@tum.de, xiang.su@helsinki.fi, n.mohan@tudelft.nl, ott@in.tum.de, panhui@ust.hk

***Abstract*—Intelligent mobile applications are often constrained by limited on-device hardware and by the latency and bandwidth overhead of cloud offloading. Offloading computation-intensive deep learning tasks to edge servers can potentially mitigate these challenges. However, existing systems struggle to scale as the numbers of tasks and users grow. This limitation stems from a fundamental conflict: requests in edge settings are typically sparse, heterogeneous, and demand immediate processing to minimize latency. However, GPUs operate most efficiently on dense, homogeneous batches, a condition that edge traffic rarely provides. Most existing approaches, including cloud-oriented solutions like Multi-Instance GPU, fail to resolve this tension because edge devices typically lack the advanced features available in high-end cloud GPUs. This paper contributes SMOOTH, a scalable multitask offloading system that introduces a *Sparsity-to-Density Abstraction* to resolve this conflict. By decomposing deep learning models into a shared backbone and lightweight task-specific heads, SMOOTH transforms sparse, heterogeneous request streams into dense, homogeneous computation blocks amenable to efficient batching. This allows highly efficient cross-task batching without prohibitive accumulation delays. To further optimize responsiveness, SMOOTH employs a compute-lightweight ($\mathcal{O}(1)$), sparsity-aware adaptive scheduler that dynamically balances inference throughput and end-to-end latency based on queue dynamics. Our evaluations demonstrate up to 2.21× higher throughput, 45% lower memory usage, and up to 82% reduced latency across diverse arrival patterns compared to baselines.**

I. Introduction

Offloading computationally intensive tasks to edge servers typically yields lower latency than cloud offloading [1], [2]. However, edge deployments lack the redundant resources to maintain separate pipelines for diverse tasks. Furthermore, an edge server typically serves a localized geographic area and a limited user base [3], [4], so the request volume for any individual task is inherently thin and temporally sparse. This creates a fundamental conflict: GPUs require dense, homogeneous batches of data to achieve high throughput [5], [6], but localized edge workloads naturally manifest as sparse, heterogeneous streams scattered across multiple tasks. Batching is also expensive in terms of latency, as it requires waiting for multiple requests to arrive, and edge arrival patterns are often unpredictable. This mismatch between how requests arrive and how the hardware consumes them efficiently is a key obstacle to scaling edge offloading.

ISBN 978-3-903176-82-9 © 2026 IFIP

Existing systems typically fall into the *one-model-per-task* paradigm, loading a separate monolithic model for each task. However, this does not suit edge scenarios where requests are scattered across tasks and users. In datacenters, spatial partitioning mechanisms such as Nvidia Multi-Instance GPU (MIG) or Multi-Process Service (MPS) can isolate concurrent models on a single GPU. On edge hardware, these mechanisms are usually unavailable. MIG requires high-end server GPUs absent from edge-class devices (e.g., the Nvidia Jetson series), and MPS on edge GPUs with unified memory causes severe memory exhaustion when multiple memory-hungry models are loaded simultaneously. Moreover, even where such mechanisms are available, they do not address the sparsity problem. Without hardware-level partitioning, the system is forced to process the sparse stream of each task sequentially, incurring frequent Compute Unified Device Architecture (CUDA) context switches. As we show empirically in Section II-B, this caps GPU utilization at $\sim$40% under high-demand scenarios, because the hardware rarely sees dense, homogeneous instruction blocks. Furthermore, sparse streams incur unavoidable batch forming delays, which reduce the service quality to a large extent. Prior works partially mitigate this problem. CoEdge [4] collects same-task requests from multiple nodes for batching, and OffloaDNN [7] partitions Deep Neural Networks (DNNs) across the edge-cloud continuum. Yet none of these solutions addresses the fundamental challenge of redundancy of isolated models and sparsity of per-task request streams.

Inspired by multi-task learning (MTL), using a shared backbone with task-specific heads offers benefits beyond parameter efficiency. Rather than managing K independent models with infrequent requests, a shared backbone processes the union of all task-specific streams, so K sparse, heterogeneous arrival patterns collapse into a single dense stream. We term this structural conversion the *Sparsity-to-Density Abstraction*: temporally sparse, heterogeneous requests become temporally dense, homogeneous compute blocks that the GPU executes efficiently. The abstraction addresses the conflict by reshaping the workload to match the GPU's efficient execution pattern,

instead of relying on hardware partitioning features unavailable on edge GPUs.

Realizing this abstraction in practice, however, necessitates solving a concrete systems challenge: *how to simultaneously maximize GPU utilization and minimize end-to-end (E2E) latency under dynamically varying workloads*. Larger batch sizes improve throughput but inflate per-request wait times. Smaller batches reduce latency but leave the GPU underutilized. On edge hardware lacking datacenter-grade spatial partitioning, there is no hardware mechanism to resolve this tension, meaning the system must manage it entirely in software. Furthermore, the scheduler itself must be compute-lightweight, as an expensive optimization solver would compete with inference for the same constrained CPU and memory budget, thereby defeating the purpose of edge-native design.

To address this, we propose `SMOOTH`, a scalable multi-task offloading system built around the *Sparsity-to-Density Abstraction*. `SMOOTH` realizes the abstraction through three tightly integrated design components:

1) Unified Memory-Sharing Abstraction: `SMOOTH` decomposes monolithic deep learning (DL) pipelines into a shared preprocessing stage, a shared computing-heavy transformer backbone, and lightweight task-specific heads, physically eliminating memory redundancy (Section III-B).

2) Cross-Task Batching and Pipelining: By grouping heterogeneous requests into unified execution blocks for the shared backbone, `SMOOTH` converts sparse per-task streams into dense GPU memory blocks. CPU-GPU pipeline parallelism further overlaps preprocessing with inference, reducing blocking. (Section III-C).

3) Sparsity-Aware Adaptive Scheduler: Rather than expensive optimization solvers, `SMOOTH` employs a compute-lightweight ($\mathcal{O}(1)$) heuristic scheduler that evaluates the bounded batch-size configuration space in constant time based on queue dynamics, optimizing throughput-latency trade-offs without competing for constrained edge resources (Section III-D).

To the best of our knowledge, this is the first work to resolve the fundamental conflict between task sparsity and GPU efficiency in edge offloading. We implement a proof-of-concept prototype[1] of `SMOOTH`. The key contributions of this paper are threefold.

I. We identify a fundamental conflict between request sparsity and GPU execution efficiency on edge devices. To resolve this, we formalize the *Sparsity-to-Density Abstraction*: the insight that sharing a backbone across heterogeneous tasks structurally converts sparse, per-task request streams into dense, batchable workloads.
II. We design cross-task batch inference and CPU-GPU pipeline parallelism mechanisms that realize the abstraction, along with a compute-lightweight adaptive scheduler ($\mathcal{O}(1)$ complexity) that dynamically optimizes the throughput-latency trade-off without imposing solver overhead.
III. We develop a complete proof-of-concept system and conduct a comprehensive evaluation across multiple metrics (resource utilization, throughput, and latency) and workload distributions (Uniform, Poisson, and log-normal), demonstrating up to $2.21\times$ throughput improvement and up to 82% latency reduction over conventional approaches.

[1] https://github.com/ViGeng/smooth

The remainder of this paper is organized as follows. Section II empirically motivates this work by analyzing the limitations of conventional task-specific offloading. Section III presents the design of `SMOOTH`. Section IV describes the evaluation setup and the results. Section V discusses related work. Finally, Section VI concludes the paper and outlines future directions.

II. Motivating Study

A. Scenarios

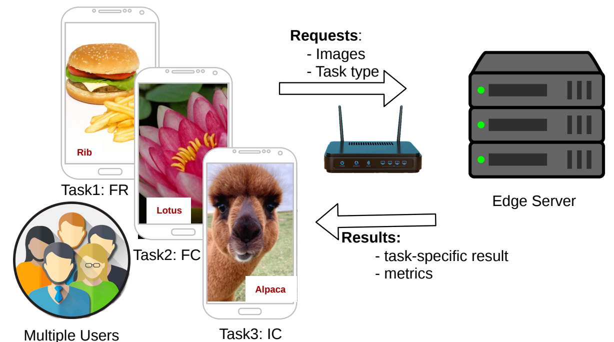


Fig. 1: Multi-task, multi-user edge offloading scenario.

As shown in Figure 1, multiple users concurrently offload distinct tasks to an edge server, including food recognition (FR) [8], flower classification (FC) [9], and general image classification (IC) [10]. While IC assigns a coarse object label, FR and FC require fine-grained categorization (e.g., pizza vs. burger, rose vs. tulip). Despite differing in output semantics, these tasks share similar deployment contexts and visual pipelines. Conventionally, each task is served by a separate model (Figure 2(a)). MTL, by contrast, enables sharing the same backbone (e.g., Swinv2 [11]) across lightweight task-specific heads (Figure 2(b)). All three tasks share similar computational profiles (processing time and memory footprint). Visual classification is deliberately chosen as a representative workload. The shared-backbone principle extends naturally to other task families (e.g., object detection and pose estimation sharing a detection backbone), though we leave such extensions to future work. Our focus is demonstrating how the *Sparsity-to-Density Abstraction* improves the scalability and latency of edge offloading.

We assume that each user sends requests at a fixed rate of 10 requests per second (RPS), processed independently by the edge server. In this setting, we conduct the following motivating study.

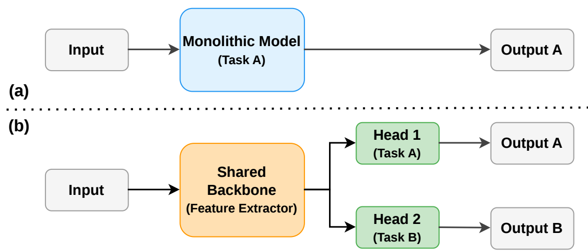


Fig. 2: (a) Separate monolithic models & (b) shared-backbone MTL.

B. Experiment Study

As discussed in Section I, the conventional monolithic approach loads a separate model per task into the same GPU. Without datacenter-grade spatial partitioning (MIG/MPS), edge devices fall back to sequential processing, which can neither utilize more parallel processing units in GPUs nor make full use of GPU memory for runtime data and intermediate results. When it comes to multiple task offloading, they further suffer from frequent context switching, exposing severe limitations in resource utilization, leading to low throughput and high latency. We quantify these limitations under three deployment schemes as follows.

(i) **3Tasks-Base**: Three separate models loaded concurrently into GPU memory (simulating MPS-style concurrent deployment on a server GPU, or sequentially multiplexed on an edge GPU). microsoft/swinv2 [11] weights are employed.

(ii) **IC-Base**: Only one IC task, using the base size model. All requests are homogeneous IC task requests.

(iii) **IC-Tiny**: Same as homogeneous IC-Base, but using a tiny size model (microsoft/swinv2).

Experiments run on an Nvidia RTX 3090Ti GPU. Each simulated user sends requests at 10 RPS, generating 200 requests per task. Figure 3 presents the averaged results, followed by our key observations.

▷ *Limited Resource Utilization.* As shown in Figure 3a, both GPU compute and memory utilization plateau early as concurrent users increase. Further increasing load yields no additional utilization because the monolithic sequential-inference approach leaves GPU resources underutilized regardless of demand.

▷ *Low Throughput.* As shown in Figure 3b, the throughput of base variants (3Tasks-Base and IC-Base) is comparable (≈ 52 RPS). Tiny model throughput is slightly higher due to its lower computation complexity, hovering between $52 \sim 80$ RPS.

▷ *High Latency.* Limited throughput causes request queuing that dominates E2E latency. As shown in Figure 3c, only six concurrent users achieve acceptable latency ($<$500 ms); beyond 18 users, E2E latency surges past 2000 ms, almost entirely due to queue waiting time.

Summary — Independent models subject to sparse request streams force the GPU into an inefficient operating mode. Even when models are co-located on the same GPU (effectively equivalent to Nvidia MIG), the lack of cross-task batching means requests are processed with small batch sizes (often $B = 1$). GPU utilization then plateaus around $\sim$40%, not for lack of compute demand, but because the hardware rarely sees dense, homogeneous instruction blocks. If the sparse requests from different tasks could be fundamentally treated as the same computational workload, batch processing could be effectively utilized for higher throughput without the prohibitive accumulation latency.

III. System Design

The motivating study (Section II) confirms that the monolithic one-model-per-task paradigm leaves GPUs starved of dense work. We now present SMOOTH, which realizes the *Sparsity-to-Density Abstraction* introduced in Section I through three tightly integrated design components (Figure 4). The system targets the following goals.

I. *Resource-efficient Multitask Support*: Support multiple concurrent tasks with a lower memory footprint than monolithic deployments, minimizing GPU memory when idle and maximizing utilization under load.

II. *High Scalability with Low Latency*: Serve a relatively larger number of concurrent users without jams or latency spikes, maintaining stable end-to-end response times.

III. *Adaptivity to Workload Dynamics*: Dynamically select the batch size that minimizes E2E latency while meeting the required throughput under varying load.

We realize the abstraction in three progressive steps. First, we structurally unify the model architecture, replacing K separate models with a single shared backbone to merge heterogeneous request streams at the architecture level (Section III-B). Second, we convert the merged stream into dense GPU work through cross-task batching and CPU-GPU pipelining (Section III-C). Third, we dynamically control the batch density via a lightweight adaptive scheduler that adjusts the throughput-latency trade-off (Section III-D).

A. System Overview

SMOOTH follows a standard client-server model (Figure 4).

▷ **Client Side.** Clients on mobile devices send offloading requests (containing task type, image data, and user metadata) to the edge server over a local wireless network and receive processed results.

▷ **Server Side.** The edge server comprises four components: (i) a preprocessing engine that converts raw images into model-compatible tensors, (ii) a shared backbone that extracts features, (iii) task-specific heads that produce final outputs, and (iv) an adaptive scheduler that dynamically selects the batch size to optimize E2E latency. The following subsections detail each component.

B. Sharing Backbone for Multitask Support

The first step in realizing the *Sparsity-to-Density Abstraction* is to structurally unify the model: replacing K separate monolithic models with a single shared backbone merges K sparse, heterogeneous request streams into one unified stream

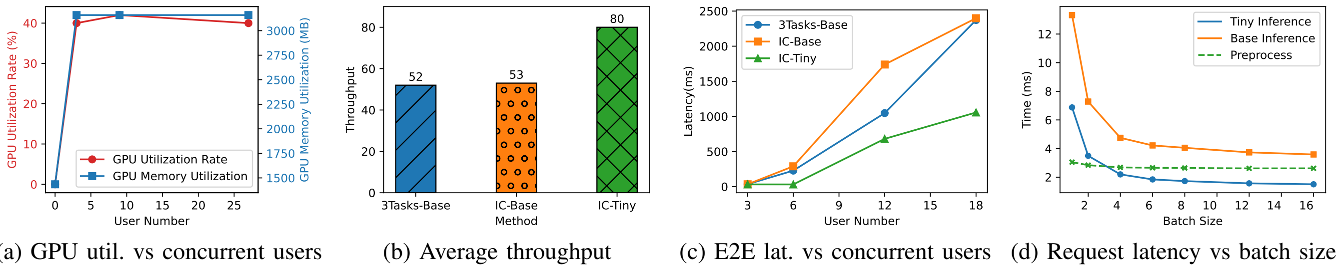


Fig. 3: Monolithic system experiment results.

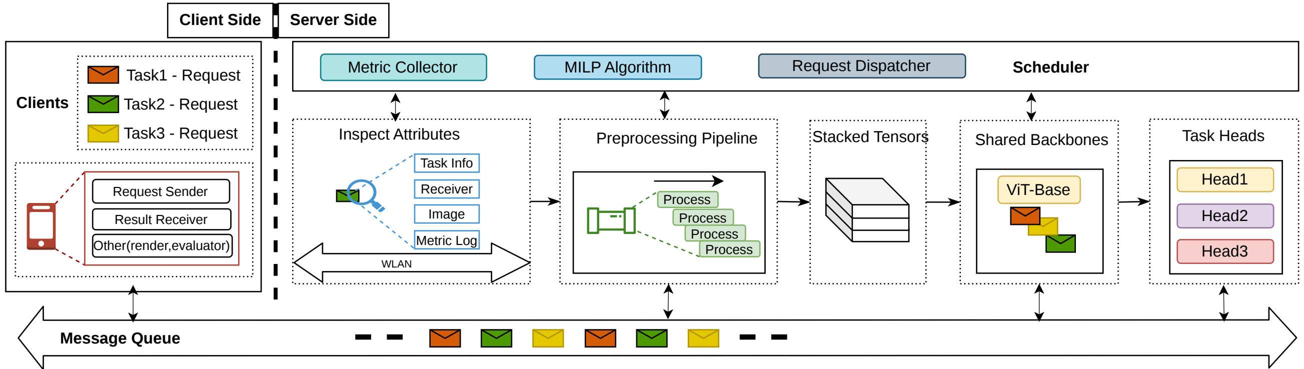


Fig. 4: SMOOTH system architecture. Clients send offload requests over a local network. The server preprocesses images, batches tensors through the shared preprocessor, shared backbone extracts visual features and dispatches them to task-specific heads. An adaptive scheduler selects the batch size at runtime.

at the architecture level. This eliminates both the memory redundancy of loading K separate models and the scheduling fragmentation that forces per-task sequential processing.

TABLE I: Performance of Task-Specific Heads on Datasets

Dataset	Test Loss	Test Accuracy (%)
Food101 [8]	0.8872	77.24
Flowers102 [9]	0.3286	91.58
CIFAR100 [10]	0.4984	84.32

Concretely, we employ a frozen pretrained Swinv2 [11] backbone (`swinv2-base-patch4-window8-256`, pre-trained on ImageNet-1K [12]) as the shared feature extractor. All requests regardless of task type route through this single backbone and are dispatched to lightweight, task-specific classification heads for their respective tasks. Only heads are trained; backbone weights remain frozen. Because training these task-specific heads is not the primary contribution of our work, we briefly report their resulting performance in Table I here for context, rather than dedicating space to it in the formal evaluation section. Using a frozen shared backbone may yield slightly lower per-task accuracy than fully fine-tuned task-specific models (quantified in Section IV-A) due to the inherent domain shift between ImageNet and downstream tasks. We justify this choice for two reasons: 1) a universal pipeline is more robust and easier to maintain from a system perspective than a collection of fragile, task-specific models, especially given that the accuracy drop is modest; 2) edge-side offloading is not intended to achieve state-of-the-art accuracy per task, but to handle routine inference locally while routing

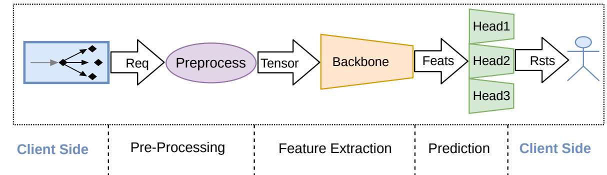


Fig. 5: SMOOTH pipeline: preprocessing, shared backbone feature extraction, and task-specific head dispatch.

difficult cases to the cloud, an extension we sketch in the future work directions (Section VI).

As shown in Figure 5, the SMOOTH pipeline loads the preprocessor, shared backbone, and task-specific heads on the server. Because the most resource-intensive component, the backbone, is shared, the system loads only one model for all K tasks, directly addressing the memory redundancy identified in Section II. However, unifying the model alone is necessary but not sufficient: the merged stream still arrives sequentially without further mechanisms to batch and pipeline the work.

C. Cross-Task Batching and Pipelining

With the model unified, heterogeneous requests now share a single execution path through the backbone. The second step in realizing the *Sparsity-to-Density Abstraction* is to exploit this structural unification: by batching requests across tasks for the shared backbone, we convert the merged stream into dense GPU compute blocks. We pair this with CPU-GPU pipeline parallelism to ensure that the backbone is continuously fed.

BatchSize	Ave Latency	Batch Latency	Theoretical Thr.
1	15.93	15.93	62.77
2	9.59	19.19	104.22
4	7.38	29.55	135.36
6	7.08	42.49	141.21
8	6.64	53.19	150.40
12	6.41	76.95	155.95
16	6.24	99.99	160.02

TABLE II: Batch processing time and theoretical throughput.

Note: Latency unit is ms, throughput unit is req/s.
On RTX3090Ti GPU, swinv2 base size model with IC head.
Average latency is calculated by $BatchLatency/BatchSize$.
Theoretical throughput is calculated by $1000ms/AverageLatency$.

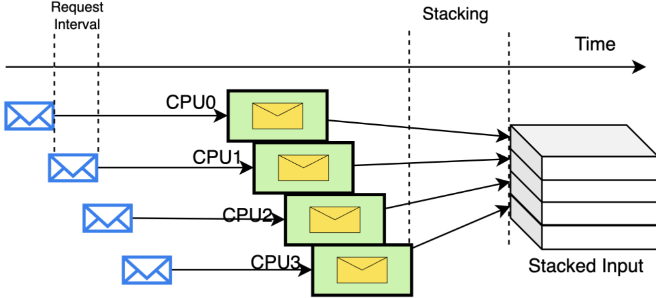


Fig. 6: Multi-core preprocessing pipeline. Requests are distributed across CPU cores; preprocessed tensors are stacked and forwarded to the backbone.

1) Cross-Task Batch Inference: Batch processing amortizes GPU kernel launch overhead and makes better use of CUDA cores: as shown in Table II, increasing batch size (B) from 1 to 16 yields up to ∼2.55× higher theoretical throughput. The shared backbone is what makes cross-task batching possible (Figure 2(b)). In monolithic deployments, each task accumulates batches only from its own sparse stream (Figure 2(a)); under `SMOOTH`, the backbone receives the union of all K task streams, reducing batch accumulation time by a factor of K. Formally, given U_k users per task at rate R_k for task $k \in \{1, \ldots, K\}$, the accumulation time for a batch of B requests is $t_{\text{wait}} = B / \sum_k U_k \cdot R_k$.

2) CPU-GPU Pipeline Parallelism: Cross-task batching requires a steady supply of preprocessed tensors. Image preprocessing (resizing, normalization) runs on the CPU, where per-image time is ∼3 ms and total batch time scales linearly (Figure 3d). For example, a batch of 16 takes 40–50 ms on a single core. To prevent this CPU-bound stage from starving the GPU, we distribute preprocessing across multiple CPU cores via a multi-process pool (Figure 6). Preprocessed tensors are stacked in a concurrent-safe queue and forwarded to the backbone. Though inter-process overhead is non-negligible at ∼3 ms, more importantly, it ensures the shared backbone is never idle, enabling continuous batch accumulation, thereby reducing the average latency.

D. Sparsity-Aware Adaptive Scheduler

The unified backbone and cross-task batching produce a dense stream whose optimal batch size varies with load. However, workload fluctuations create a direct tension: if throughput falls below the request arrival rate, request jams and latency spikes ensue (Figure 3c); if the system over-provisions via large batches, accumulation delays and processing times inflate E2E latency unnecessarily. The final step in realizing the abstraction is to dynamically control this density: selecting the batch size that minimizes E2E latency while meeting throughput demand.

We continuously monitor the aggregate request arrival rate Thr_{load} using a sliding time window. For a given batch size B, the system's theoretical throughput $Thr_{\text{theo}}(B)$ is bounded by the slowest stage in the pipeline:

$$Thr_{\text{theo}}(B) = \min\left(\frac{1}{t_{\text{pre}}}, \frac{B}{t_{\text{B}}}, \frac{1}{t_{\text{head}}}\right) \tag{1}$$

where t_{pre}, t_{B}, and t_{head} are the profiled execution times for preprocessing, backbone inference (for batch size B), and head execution.

The expected E2E latency t_{e2e} consists of network transmission time t_{net}, processing time t_{proc}, and batch accumulation wait time t_{wait}. Assuming uniformly distributed arrivals, $t_{\text{wait}} = B/Thr_{\text{load}}$, yielding:

$$t_{\text{e2e}} = t_{\text{net}} + \underbrace{(t_{\text{pre}} + t_{\text{B}} + t_{\text{head}})}_{t_{\text{proc}}} + \underbrace{\frac{B}{Thr_{\text{load}}}}_{t_{\text{wait}}} \tag{2}$$

While real-world traffic exhibits burstier patterns (Section IV-E), this expectation provides a highly effective and tractable baseline.

Our objective is to find a batch size B from a predefined operational set $B_{\text{set}} = \{1, 2, 4, 6, 8, 12, 16\}$ (Table II) that minimizes latency while satisfying the throughput constraint:

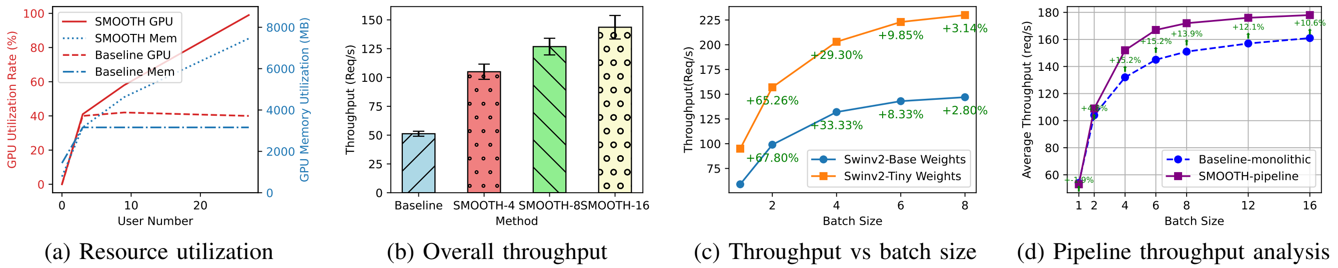
$$\begin{aligned} \min_{B \in B_{\text{set}}} \quad & t_{\text{e2e}} \\ \text{subject to:} \quad & Thr_{\text{theo}}(B) \geq Thr_{\text{load}} \end{aligned} \tag{3}$$

Because B_{set} is small and fixed, the scheduler (Algorithm 1) simply enumerates all feasible batch sizes in $\mathcal{O}(1)$ time (< 1 ms) to find the global optimum. This lightweight heuristic avoids the overhead of expensive continuous optimization solvers that would otherwise compete with inference for edge CPU cycles. Our evaluation (Section IV-D) shows this concise approach achieves 9%–82% latency reduction compared to static or greedy batch sizing.

IV. Evaluation

A. Implementation Setup and Considerations

We implement `SMOOTH` in `Python` following the architecture in Section III. Client-server communication uses RabbitMQ [13] via Advanced Message Queuing Protocol (AMQP) for request/result transport and WebSockets for control signaling. CPU-GPU pipeline parallelism is realized via Python's `multiprocessing` module (bypassing the Global Interpreter Lock (GIL)). The backbone and task heads are loaded from HuggingFace pre-trained weights [14] on PyTorch. The adaptive scheduler runs in a separate thread with negligible overhead (<1 ms per invocation); it re-evaluates periodically rather than per-request, so we exclude it from pipeline latency measurements.



(a) Resource utilization (b) Overall throughput (c) Throughput vs batch size (d) Pipeline throughput analysis

Fig. 7: Resource utilization and throughput. (a) Paired curves illustrate GPU core utilization (red, left axis) and memory footprint (blue, right axis) for SMOOTH versus 3Tasks-Base across varying user counts. (b) box plots of throughput per batch size; (c) curves of distinct color denote different maximum batch sizes plotted against user count; (d) pipeline-enabled SMOOTH versus monolithic baseline under saturated load.

Algorithm 1 Sparsity-Aware Adaptive Batch Sizing

Input: Arrival rate Thr_{load}, profiled stage times $t_{\text{pre}}, t_{\text{head}}, t_{\text{B}}[B]$, and t_{net}
Output: Optimal batch size B_{opt}

1: $B_{\text{set}} \leftarrow \{1, 2, 4, 6, 8, 12, 16\}$ ▷ candidate sizes
2: $t_{\text{e2e}}^{\min} \leftarrow \infty$ ▷ best so far
3: **for** each $B \in B_{\text{set}}$ **do**
4: $Thr_{\text{bbone}} \leftarrow B / t_{\text{B}}[B]$ ▷ backbone thr.
5: $Thr_{\text{theo}} \leftarrow \min(1/t_{\text{pre}}, Thr_{\text{bbone}}, 1/t_{\text{head}})$ ▷ bottleneck
6: **if** $Thr_{\text{theo}} \geq Thr_{\text{load}}$ **then**
7: $t_{\text{wait}} \leftarrow B / Thr_{\text{load}}$ ▷ batch fill time
8: $t_{\text{e2e}} \leftarrow t_{\text{net}} + t_{\text{pre}} + t_{\text{B}}[B] + t_{\text{head}} + t_{\text{wait}}$
9: **if** $t_{\text{e2e}} < t_{\text{e2e}}^{\min}$ **then**
10: $t_{\text{e2e}}^{\min} \leftarrow t_{\text{e2e}}$ ▷ update best
11: $B_{\text{opt}} \leftarrow B$
12: **end if**
13: **end if**
14: **end for**
15: **return** B_{opt}

Testbed. The server runs Ubuntu 20.04 LTS with an RTX 3090Ti GPU and an i9-12900K CPU. This exceeds typical edge devices such as the Nvidia Jetson Orin, but serves two deliberate purposes: (i) it ensures a *fair comparison*, since the monolithic baseline requires loading three Swinv2-base models simultaneously; and (ii) the phenomena we measure, namely the ~40% utilization cap, context-switching overhead, and batch-accumulation trade-off, are *architectural* behaviors that persist across GPU tiers, arising from kernel dispatch patterns rather than absolute compute capacity. On more constrained embedded hardware that cannot load three Swinv2-base models simultaneously, SMOOTH's memory savings and batch efficiency would yield even greater relative benefits, so our results represent a *conservative lower bound*. Validation on Jetson-class devices is a dedicated follow-up direction.

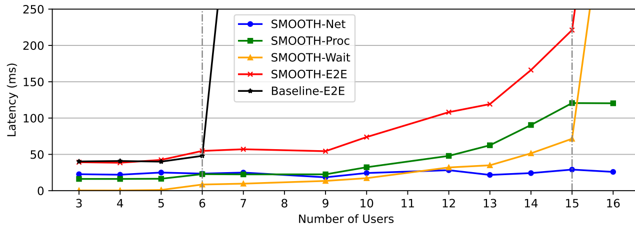
The client runs OS X Sonoma 14.4 (M2 Pro, 16 GB RAM); multiple client processes emulate concurrent user load. The network uses a Redmi AX5400 (WiFi 6) router and a 10 Gbps switch. Middleware: RabbitMQ 3.13.3, Erlang 26.2.5.1, Docker Swarm 26.1.1. DL stack: PyTorch 2.3.0, CUDA 12.3, HuggingFace Transformers 4.41.2 [14]. Datasets: Food101 (FR), Flowers102 (FC), CIFAR100 (IC).

Baseline Configuration. Throughout the evaluation, we compare SMOOTH against the 3Tasks-Base baseline introduced in Section II. In this configuration, three separate monolithic models are loaded concurrently into GPU memory to process incoming tasks. Technically, this setup operates closely to how spatial partitioning mechanisms like NVIDIA MIG or software multiplexers like MPS function in datacenters: multiple independent models reside in memory simultaneously to serve distinct request streams. However, because edge hardware uniformly lacks MIG's hardware-level isolation, these models are forced into sequential execution, sharing CUDA cores via temporal multiplexing. This makes 3Tasks-Base the most direct and realistic representation of standard multi-model serving in edge environments. In particular, it is functionally representative of NVIDIA Triton and TorchServe with dynamic batching enabled: those frameworks batch *within* a single model (i.e., aggregate same-task requests) but neither performs *cross-task* batching across independent model instances, because there is no shared parameter surface through which to consolidate heterogeneous tasks. 3Tasks-Base therefore captures the same structural ceiling these frameworks impose, minus their commercial-level engineering tuning. Without cross-task structural sharing, even advanced serving frameworks cannot resolve the memory redundancy and per-task sparsity inherent to the K-models-for-K-tasks paradigm.

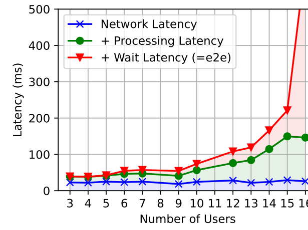
Accuracy Trade-off. The shared frozen Swinv2-base backbone yields 77.24% on Food101, 91.58% on Flowers102, and 84.32% on CIFAR100. Fully fine-tuned Swinv2 models [11] can exceed these numbers by several points (commonly reported upper bounds approach ~90% on Food101 and above 95% on CIFAR100). This gap is the cost of freezing the backbone and is *orthogonal* to SMOOTH's system-level contributions: per-task accuracy can be recovered independently via fine-tuning, adapter-based heads, or deeper heads, without modifying the pipeline, scheduler, or cross-task batching mechanisms.

B. Resource Efficiency

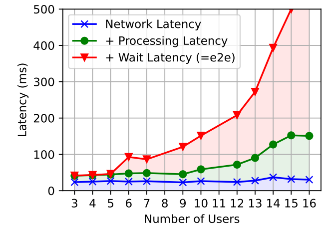
We measure resource utilization with *nvidia-smi* across varying user counts, comparing SMOOTH against the 3Tasks-Base baseline (Figure 7a).



(a) E2E latency and stage component latency comparison vs. concurrent users.



(b) Latency stack (shared backbone)



(c) Latency stack (separate backbones)

Fig. 8: Latency analysis. (a) Black triangle: baseline E2E; red cross: SMOOTH E2E; green square: SMOOTH process; yellow triangle: SMOOTH wait, all versus concurrent users. (b)–(c) stacked latency components (wait, preprocess, backbone, head) for SMOOTH with a shared versus separate per-task backbones.

▷ *Idle State.* Even when idle, the system must retain model weights in GPU memory. The shared backbone reduces the idle memory footprint by 45% compared to the baseline (1435 MB $\rightarrow$ 780 MB), freeing memory for other workloads.
▷ *Memory under Load.* As user count grows, SMOOTH's scheduler increases the batch size accordingly, and memory usage scales linearly, since each additional batch slot is used for inference. The baseline, by contrast, cannot translate additional memory capacity into throughput because it processes at $B{=}1$.
▷ *GPU Core Utilization under Load.* The baseline processes requests sequentially at $B{=}1$, capping GPU utilization at $\sim$40% due to context-switching overhead rather than compute limits. SMOOTH batches requests into dense memory blocks, enabling concurrent GPU core execution and lifting this cap.

Takeaway — The shared backbone compresses idle memory (45% reduction) while enabling productive memory scaling under load. Combined with cross-task batching, SMOOTH bypasses the 40% utilization cap inherent to sequential monolithic processing, achieving efficient use of both GPU memory and compute cores across all load levels.

C. Scalability Analysis

Parameter Ranges. We sweep maximum batch size $\in \{4, 8, 16\}$ and concurrent user count $n \in [1, 15]$. The batch set mirrors the bounded configuration space $\{1, 2, 4, 6, 8, 12, 16\}$ of the $\mathcal{O}(1)$ adaptive scheduler (Section III-D), so the three upper bounds directly probe the scheduler's design knob. The user range spans the meaningful comparison regime on a single edge server: idle at $n{=}1$, baseline acceptable-latency boundary around $n{=}6$, and SMOOTH saturation around $n{=}15$. Beyond this range both systems enter sustained overload, which is orthogonal to the scaling question we study.

We measure throughput under increasing concurrent users ($n{=}5$ repetitions). Throughput is computed via Equation (4), where $N_{90\%}$ and $N_{10\%}$ denote the cumulative requests completed by the 90th and 10th percentile timestamps ($T_{90\%}$, $T_{10\%}$), respectively. Results are shown in Figure 7b.

$$T = \left\lfloor \frac{N_{90\%} - N_{10\%}}{T_{90\%} - T_{10\%}} \right\rfloor \quad T_{avg} = \left\lfloor \frac{1}{n} \sum_{i=1}^{n} T_i \right\rfloor \tag{4}$$

▷ *Throughput Improvement.* Our method, tested with different parameters (max batch size $\in \{4, 8, 16\}$), consistently outperforms the baseline method (*3Tasks-Base*). The throughput improvement is substantial, with our method achieving up to $\sim 2.21\times$ higher throughput (average = 143.6 req/s) compared to the baseline (average = 51.2 req/s).
▷ *Throughput Improvement Trend.* Maximum throughput increases with batch size but with diminishing returns beyond $B{=}4$, as indicated by the flattening curves in Figure 7c and the overlapping distributions in Figure 7b. This saturation reflects GPU compute capacity limits and rising synchronization overheads. The Tiny model benefits more from larger batches than the Base model, since the Base model's larger weights consume more per-sample GPU capacity (Figure 7c).
▷ *Throughput Stability.* Figure 7b also indicates increasing uncertainty in throughput at larger batch sizes, visible as wider spread between max and min measurements. System synchronization, network communication, and accumulated wait times contribute to this variability, which becomes more pronounced when the batch size exceeds 4 (confirmed by per-request latency tracking in Figure 9a).

To isolate the pipeline's contribution, we compare against a monolithic model under saturated load (500 pre-queued requests). As shown in Figure 7d, the pipeline achieves -1.9% to $+15.2\%$ throughput improvement over the monolithic model. At $B{=}1$ both degenerate to sequential execution (54 vs. 53 req/s); gains plateau beyond $B{=}8$ due to multiprocessing overhead and the CPU thread pool size (8 threads).

Takeaway — Batch inference yields up to $2.21\times$ throughput improvement, but returns diminish sharply beyond $B{=}4$ on our hardware. The optimal batch size depends on both GPU capacity and model complexity, motivating the adaptive scheduler evaluated in Section IV-D.

D. Latency Analysis

We assess latency performance under increasing user load:
▷ *Latency Breakdown.* Figure 8a compares E2E and component latencies. SMOOTH (×) scales significantly better than the baseline (▲): the baseline exceeds acceptable latency at 6 users (590.45 ms), whereas SMOOTH supports up to 15 users (221.18 ms). The dominant component is process latency (■), which rises with batch size. Wait latency (▲) also grows,

particularly at the scalability boundary (e.g., 39.22% increase from 14 to 15 users), as larger batches require longer batch accumulation times.

We further isolate the shared backbone's contribution by comparing SMOOTH with shared backbone (Figure 8b) against SMOOTH with separate per-task models (SMOOTH's pipeline but without shared backbone) (Figure 8c). The shared backbone substantially reduces wait latency as user count increases (e.g., at n=6: 48.6% $\rightarrow$ 36.1% of total latency), because the unified stream provides more frequent batching opportunities.

▷ *Response Variance and Batch Size.* Figure 9a plots per-request E2E latency across 3000 requests. As concurrent users increase, the scheduler selects larger batch sizes, producing progressively higher latency plateaus ($t_{1500\rightarrow1800} < t_{1800\rightarrow2100}$). Latency variance grows sharply with batch size (64.18 $\rightarrow$ 1089.39), because earlier-arriving requests in a large batch must wait for the batch to fill before execution begins. Beyond B=8, variance becomes unacceptable ($\geq$ 1058.97), validating the scheduler's strategy of selecting the smallest batch size that meets throughput demand.

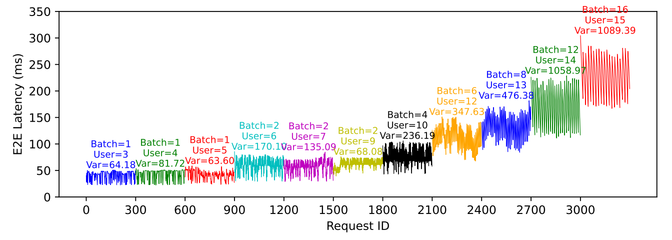
To quantify adaptive scheduling benefits, we compare against a greedy approach that always selects the maximum feasible batch size. Our scheduler achieves 9%–82% latency reduction at a controlled sacrifice of excess throughput capacity (1–141 req/s). Because the selected configurations still strictly satisfy the required throughput demand, overall scalability is uncompromised. This demonstrates greedy throughput maximization yields sharply diminishing returns in E2E responsiveness.

▷ *Interplay between Utilization and Latency.* Reading Figures 7a, 7c and 9a together shows that utilization and latency are two views of the same batching decision. Lifting the baseline's ~40% utilization cap requires dense batches, which only can be achieved when SMOOTH aggregates across tasks. Yet throughput gains flatten beyond B=4 while per-request latency variance grows from 64 to over 1000 as batch size rises. The result is a narrow operating band where the GPU is productively loaded and E2E latency remains bounded: outside this band, larger batches buy utilization at super-linear tail-latency cost, while smaller batches restore responsiveness but leave the GPU idle. The adaptive scheduler formalizes this trade-off by selecting the smallest batch size whose utilization meets the incoming throughput demand.

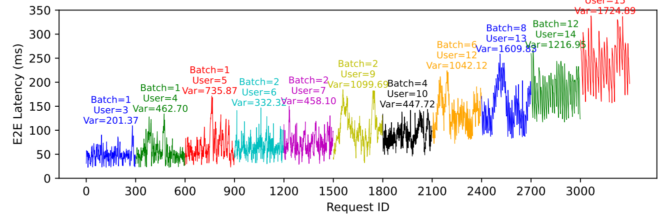
Takeaway — Beyond the throughput gains from cross-task batching, the shared backbone independently reduces wait latency by unifying the request stream. Larger batches improve throughput but increase latency variance. The adaptive scheduler resolves this tension by selecting the smallest batch size that meets the throughput demand, achieving up to 82% latency reduction over greedy sizing.

E. Request Arrival Patterns

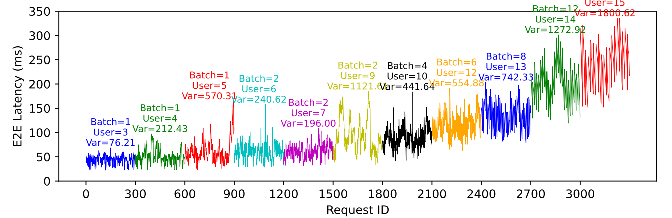
The preceding experiments use a fixed-rate (uniform) arrival distribution. To test robustness under realistic workloads, we additionally evaluate Poisson and log-normal arrival patterns,



(a) Fixed-rate distribution



(b) Poisson distribution



(c) Log-normal distribution

Fig. 9: Per-request E2E latency under different arrival patterns. Each point is one request in arrival order; color encodes the operating regime (concurrent users) as load grows along the x-axis, and plateau heights correspond to the batch size selected by the adaptive scheduler rather than to slower processing. (a) Fixed-rate; (b) Poisson; (c) log-normal.

both parameterized with the same mean rate λ as the fixed-rate case. For the Poisson process, inter-arrival times are exponentially distributed with mean $1/\lambda$. For the log-normal distribution, inter-arrival times follow $Y = e^X$ where $X \sim N(\mu, \sigma^2)$, with μ computed as Equation (5) and σ=0.7 chosen empirically.

$$\mu = \ln\left(\frac{1}{\lambda}\right) - \frac{\sigma^2}{2} \tag{5}$$

Results are presented in Figure 9. Both distributions exhibit similar macroscopic trends to the fixed-rate case, but with significantly higher latency variance. Under the Poisson distribution, variance increases from 201.37 to 1724.89; under the log-normal distribution, from 76.21 to 1800.62. The log-normal pattern additionally produces sudden latency spikes (e.g., requests 600–900 in Figure 9c) caused by occasional long inter-arrival pauses that starve the batch accumulator, inflating wait times for already-queued requests.

Takeaway — Burstier arrival patterns amplify the batch-accumulation caused latency variance, hinting that some throughput headroom must be reserved if a QoS needs to

be ensured. A moderate batch size (e.g., B=4) provides a robust operating point that absorbs arrival variability while maintaining acceptable E2E latency.

Generalization of the Findings. The reported numbers stem from one concrete configuration (three classification tasks, a frozen Swinv2-base backbone, an RTX 3090 Ti, and three arrival distributions), but the qualitative trends generalize along several axes. The sparsity-to-density abstraction applies to other vision tasks that share a visual backbone (detection, segmentation, retrieval), although heavier heads shift the optimal batch size downward. Cross-modality workloads exaggerate the benefits, since they can integrate more separate sparse streams. The observed effects originate from kernel-dispatch patterns and memory redundancy rather than from absolute compute capacity, so the trends are expected to persist on embedded-tier GPUs such as the Nvidia Jetson Orin. Heavier-tailed arrivals beyond the fixed, Poisson, and log-normal cases we study would amplify batch-accumulation variance and may require elastic admission control, but the scheduler's decision rule remains valid because it depends on observed queue dynamics rather than an assumed arrival model. Finally, the design is backbone-agnostic: any ViT or CNN trunk with lightweight heads is compatible, with the boundary condition that the backbone remains the dominant cost.

V. Related Work

A. Multitask Deep Learning

MTL trains a single network on multiple tasks by sharing a backbone across lightweight task-specific heads [11], [15], [16]. Existing MTL research, however, treats the shared backbone purely as a means of parameter efficiency instead of computing efficiency. SMOOTH exploits this overlooked property, *structurally merging K* sparse per-task request streams into one dense flow that can be batched efficiently on the GPU.

Mixture-of-Experts (MoE) [17]–[20] handles diverse domain knowledge, but at a different granularity. MoE applies *intra-model* sparse routing: a gating network selects which expert sub-networks process each input. This decides *which parameters* are active per sample, but leaves the *inter-request* sparsity problem untouched. MoE does not consolidate requests from different tasks into shared batches, nor does it bridge the gap between thin per-task arrival rates and GPU-efficient batch sizes.

B. Scalability for Edge Offloading

Edge offloading moves computation from mobile devices to nearby servers [2], yet few works serve *multiple concurrent tasks* at scale. Jaguar [21] and SEAR [22] optimize latency and caching for multi-user AR but remain single-task systems with no cross-task GPU optimization. CoEdge [4] improves throughput by aggregating same-task requests into batches from multiple edge nodes. However, because each model is deployed separately for a specific task, every instance still faces a sparse request stream. OffloaDNN [7] partitions DNN layers across the edge-cloud continuum, distributing computation *spatially* rather than improving request-level density.

SMOOTH differs structurally. CoEdge batches requests *within* tasks; OffloaDNN distributes layers *across* nodes; SMOOTH clusters requests *across* tasks by unifying the model. This converts the problem from spatial distribution to *temporal densification*: instead of spreading sparse work over more hardware, SMOOTH consolidates heterogeneous streams into dense batches on a single edge server.

C. Model Serving Systems

Production serving frameworks such as NVIDIA Triton, TorchServe, Clipper [5], and Lazy Batching [6] provide dynamic batching, SLA-aware queuing, and model management. Their dynamic batching aggregates requests *within* a single model (i.e., same-task requests), which is the only batching opportunity visible when each task is represented by an independent model instance. They excel at serving *individual* models but treat each task as an independent instance with separate memory and execution pipelines, offering no way to share computation *across* tasks, and therefore cannot address the cross-task sparsity that SMOOTH targets.

SMOOTH is complementary: rather than optimizing *how* a model is served, it restructures *what* is served. A shared backbone replaces K redundant models, and system mechanisms (CPU-GPU pipelining, $\mathcal{O}(1)$ adaptive scheduling) are co-designed around this structure. The shared backbone can still be deployed inside Triton or TorchServe to benefit from their runtime features. The distinction is that SMOOTH eliminates the structural redundancy of multi-model deployments that serving frameworks, by design, do not address.

VI. Conclusion and Future Work

This paper presents SMOOTH, a scalable multitask offloading system that introduces the *Sparsity-to-Density Abstraction* to resolve the fundamental conflict between sparse, heterogeneous edge workloads and the GPU's need for dense, homogeneous compute blocks. By decomposing monolithic models into a shared backbone with lightweight task-specific heads, SMOOTH structurally merges per-task request streams into a single dense stream amenable to efficient cross-task batching. A compute-lightweight ($\mathcal{O}(1)$) adaptive scheduler dynamically balances throughput and latency without competing for edge resources. Evaluation on a proof-of-concept system demonstrates up to 2.21× throughput improvement, 82% latency reduction, and 45% idle memory savings compared to conventional monolithic deployments. Nevertheless, several directions remain open.

a) Content-Based Edge-Cloud Collaborative Offloading: .The shared backbone improves throughput and scalability by merging sparse streams into dense batches, but it trades per-task accuracy compared to fully fine-tuned task-specific models (Section IV-A). A natural extension is to combine SMOOTH with cloud offloading: routine or low-precision classifications stay on the edge model to keep latency low, while

complex or ambiguous inputs are selectively offloaded to high-capacity cloud models when higher accuracy is required. This content-aware offloading strategy would require a lightweight decision mechanism that assesses the difficulty of the request at the edge before routing, creating a tiered inference pipeline that balances accuracy, latency, and resource cost across the computing continuum. Nevertheless, content-centric offloading also introduces new challenges and opportunities, such as how to adapt ICN-based distributed caching and computation reuse techniques [23] to this hybrid edge-cloud setting.

b) Multi-Edge Collaborative Orchestration: SMOOTH currently runs on a single edge server with a fixed set of task types, yet real edge deployments usually span many nodes and the task mix at each node reflects the habits of its local users. Following the familiar 80/20 pattern, a node tends to see a handful of dominant tasks most of the time, while the remaining long-tail tasks arrive too rarely for the local model to stay sharp. A natural next step is to let neighboring nodes help each other: when a long-tail request arrives, it can be forwarded to a nearby node whose specialization already covers that task, with routing decisions informed by lightweight on-path network assessment [24]. This extends the *Sparsity-to-Density* idea from intra-node batching to inter-node task routing, so that dense workloads emerge across the edge as a whole. Realizing it raises open questions about routing policy, load balancing, and network overhead, where communication-efficient techniques such as sparse gradient compression [25] hint at how to keep inter-node traffic affordable, while emerging 6G connectivity [26] offers further room for co-design.

c) Operational Considerations: Adopting SMOOTH in production edge systems carries several practical drawbacks that operators should weigh against its throughput and memory benefits. The shared backbone concentrates failures (a crash affects every task on the node) and couples backbone and heads (upgrades require coordinated redeployment). Multi-tenant isolation is coarser than in per-model serving, and cold-start cost is dominated by a single large backbone rather than multiple task-specific models. Because the backbone is frozen, per-task accuracy is capped by its pretrained features and may need per-node adaptation under domain shift. The adaptive scheduler also adds a modest observability overhead that must be budgeted on constrained devices. Evaluating energy trade-offs and validating SMOOTH end-to-end on embedded-tier GPUs (e.g., Nvidia Jetson Orin) is a critical direction for future development.

Acknowledgment

This research was supported in part by NordForsk through Nordic University Cooperation on Edge Intelligence (168043), in part by the Dutch Growth Fund project 6G Future Network Services, and in part by Research Council of Finland XRISE project and InterEarth RESET project.

References

[1] M. Satyanarayanan, "The emergence of edge computing," *computer*, vol. 50, no. 1, pp. 30–39, 2017.

[2] D. Xu *et al.*, "Edge intelligence: Empowering intelligence to the edge of network," *Proceedings of the IEEE*, vol. 109, no. 11, pp. 1778–1837, 2021.

[3] G. Ananthanarayanan *et al.*, "Real-time video analytics: The killer app for edge computing," *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[4] Z. Jiang *et al.*, "Coedge: A cooperative edge system for distributed real-time deep learning tasks," in *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*, 2023, pp. 53–66.

[5] D. Crankshaw *et al.*, "Clipper: A {Low-Latency} online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[6] Y. Choi *et al.*, "Lazy batching: An sla-aware batching system for cloud machine learning inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 493–506.

[7] C. Puligheddu *et al.*, "Offloadnn: Shaping dnns for scalable offloading of computer vision tasks at the edge," in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2024, pp. 1–11.

[8] L. Bossard *et al.*, "Food-101 – mining discriminative components with random forests," in *European Conference on Computer Vision*, 2014.

[9] M.-E. Nilsback *et al.*, "Automated flower classification over a large number of classes," in *2008 Sixth Indian conference on computer vision, graphics & image processing*. IEEE, 2008, pp. 722–729.

[10] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," 2009.

[11] Z. Liu *et al.*, "Swin transformer v2: Scaling up capacity and resolution," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 12 009–12 019.

[12] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[13] RabbitMQ, "Rabbitmq," 2024.

[14] T. Wolf *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[15] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[16] R. Liang *et al.*, "A multi-head ensemble multi-task learning approach for dynamical computation offloading," in *GLOBECOM 2023-2023 IEEE Global Communications Conference*. IEEE, 2023, pp. 6079–6084.

[17] N. Shazeer *et al.*, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*, 2017.

[18] S. Qin *et al.*, "Optimal expert selection for distributed mixture-of-experts at the wireless edge," *arXiv preprint arXiv:2503.13421*, 2025.

[19] Q. Song *et al.*, "Mixture-of-experts for distributed edge computing with channel-aware gating function," *arXiv preprint arXiv:2504.00819*, 2025.

[20] C. Riquelme *et al.*, "Scaling vision with sparse mixture of experts," *Advances in Neural Information Processing Systems*, vol. 34, pp. 8583–8595, 2021.

[21] W. Zhang *et al.*, "Jaguar: Low latency mobile augmented reality with flexible tracking," in *Proceedings of the 26th ACM international conference on Multimedia*, 2018, pp. 355–363.

[22] ——, "Sear: Scaling experiences in multi-user augmented reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 5, pp. 1982–1992, 2022.

[23] W. Geng *et al.*, "Sok: Distributed computing in icn," in *Proceedings of the 10th ACM Conference on Information-Centric Networking (ICN)*, 2023, pp. 88–100.

[24] ——, "Poster: Kut: Towards lightweight on-path network assessment for edge orchestration," in *Proceedings of the 21st International Conference on emerging Networking EXperiments and Technologies*, 2025, pp. 9–11.

[25] Y. Wang *et al.*, "Pactrain: Pruning and adaptive sparse gradient compression for efficient collective communication in distributed deep learning," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2025, pp. 1–7.

[26] A. Zavodovski *et al.*, "Transforming the internet with 6g: Towards architectural extensibility," in *2025 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. IEEE, 2025, pp. 1–6.