ExoMem: OS-Governed Memory for Local Large-LLM Inference on Mobile Edge Devices

Jun You*
Department of Computer Science
City University of Hong Kong
Hong Kong SAR, China

Kun Wang*
Department of Computer Science
City University of Hong Kong
Hong Kong SAR, China

Jiesong Chen
Department of Computer Science
City University of Hong Kong
Hong Kong SAR, China

Zhidan Liu
INTR Thrust, Systems Hub
HKUST (Guangzhou)
China

Zhenjiang Li
Department of Computer Science
City University of Hong Kong
Hong Kong SAR, China

Abstract

High-quality large language model inference is increasingly used on mobile edge, from satellites to smart homes, where limited connectivity or privacy requirements motivate local model inference. In many such settings, token generation is inherently compute-limited and added latency may be acceptable, but output quality, memory feasibility and crash-free execution are essential. However, on today's mobile edge with unified-memory architectures, running a large model near the device limit can trigger non-deterministic out-of-memory failures. This is because deep learning frameworks retain freed buffers inside caching pools, thus continuing to occupy shared DRAM, which the OS cannot reclaim. During inference, fragmentation and conservative reuse can produce growing memory usage, eventually exceeding the memory availability. In this paper, we present ExoMem, an OS-governed middleware that restores OS authority over physical pages without sacrificing framework compatibility. ExoMem keeps model topology and operators in the framework, while anchoring weights in OS-managed mappings, and introduces a dormant state, whose virtual addresses remain stable for correctness but whose physical pages are immediately reclaimable when needed, leveraging standard OS page caching as an implicit multi-tier hierarchy. Implemented as a plug-and-play extension, ExoMem enables stable local inference for substantially larger LLMs on mobile edge, including 72B-class models on 64GB NVIDIA Jetson devices, while preserving headroom for concurrent workloads.

CCS Concepts

• **Human-centered computing** → **Mobile computing**; • **Computing methodologies** → **Machine learning**.

Keywords

EdgeAI systems, On-device LLMs, Memory management

ACM Reference Format:
Jun You, Kun Wang, Jiesong Chen, Zhidan Liu, and Zhenjiang Li. 2026. ExoMem: OS-Governed Memory for Local Large-LLM Inference on Mobile Edge Devices. In *The 32nd Annual International Conference on Mobile Computing and Networking (MobiCom '26), October 26–30, 2026, Austin, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3795866.3844471

1 Introduction

Large language models (LLMs) are rapidly becoming the engine behind sensing, reasoning, and decision support in mobile edge systems, such as robots, drones, vehicles, and Jetson-class devices that often interpret rich signals and act locally [10–12, 20, 30, 43, 46, 65, 67]. Most existing efforts make LLMs *lighter* or *faster*, targeting smaller memory footprints and better latency or throughput. In this paper, we study an orthogonal but increasingly important dimension on mobile edge devices — *in many deployments, slow token generation can be acceptable, but output quality is non-negotiable and inference must be performed locally, e.g.*, running a high-quality 72B-parameter model (such as Qwen2.5-72B-FP16) on a Jetson AGX with 64 GB memory is commonly viewed as impractical. This paper asks: can we make large-LLM inference feasible on mobile edge devices locally?

Satellite edge computing makes this need especially clear [14, 17, 52, 61]. Downlink opportunities are capacity-limited

*Co-first authors.

MobiCom '26, Austin, TX, USA

ACM ISBN 979-8-4007-2505-0/26/10
https://doi.org/10.1145/3795866.3844471

and intermittent. As a result, satellites often need to make onboard decisions, such as anomaly triage, scene prioritization, and mission summarization. In these settings, smaller models can be insufficient, where missing an anomaly, misranking high-value scenes, or producing low-fidelity summaries directly affects mission decisions and wastes scarce downlink resource [44, 62]. At the same time, many tasks are not interactive at the token level, *e.g.*, selecting what to downlink, generating periodic mission summaries, or producing explanations for detected anomalies, so they can tolerate added latency as long as the output quality is high and the system remains robust under tight resource budgets [2, 6, 25, 58, 60]. Beyond space, similar latency-tolerant but quality-critical workloads are emerging on consumer devices, where on-device assistants continuously ingest sensitive content and periodically produce private summaries, action items, and searchable memory during idle periods [32, 54–56, 59].

These scenarios motivate running large LLMs within a limited memory, without exceeding the budget even transiently. By doing so, an LLM can run even when the full model does not fit in the available memory budget, by bringing in only the weights needed at each step. Achieving this brings a trinity of benefits: (1) The device can run a larger LLM and obtain higher quality reasoning within memory constraints; (2) Strict containment allows the device to reserve reliable headroom for other concurrent tasks, including smaller responsive models and real time control workloads; (3) Avoiding sudden memory peaks prevents out-of-memory (OOM) failures that can destabilize mission critical devices.

A natural question is whether we can directly reuse existing approaches that execute a model *layer by layer* while swapping weights between memory and external storage. Variants of this idea have been successful for traditional models on mobile edges and for LLMs on cloud servers [3, 4, 22, 28, 47, 48]. We find that layer-wise swapping is still a necessary mechanism, but existing solutions cannot be applied to mobile edge because of two properties that co-exist in practice: the very large layer size of LLMs and a unified memory architecture (UMA) of the mobile edge device, where CPU and GPU share the same physical DRAM pool.

Our analysis identifies the key reason that LLM execution on mobile edge devices through the layer swapping can lead to growing memory usage and unexpected peak usage, causing memory constraints invalid. The root cause is a mismatch between framework-level memory lifecycle management and operating system (OS)-level memory reality under UMA. On UMA, although CPU and GPU share the same physical DRAM, allocation ownership must still be isolated across processors to prevent errors from uncoordinated asynchronous access. This isolation forces deep learning frameworks to retain freed buffers inside private caching pools rather than returning them to the OS [1, 5]. On discrete CPU–GPU systems such isolation is harmless, as the retained CPU-side pages do not consume GPU memory. On UMA, however, the retained pages still occupy the shared DRAM that the GPU, OS, and co-located workloads all depend on, causing physical memory to accumulate across layers and leading to unpredictable OOM failures. This reflects a fundamental system-level constraint of UMA rather than a framework-specific engineering issue. Addressing this requires restoring OS visibility and control over physical pages while preserving framework correctness. To this end, we solve technical challenges:

(1) Opaque-pool reconciliation. How can we make physical memory truly reclaimable under OS policy when the deep learning framework manages memory through opaque pools? If framework reservations are left unchecked, effective memory use will increase and exceed the budget. But if the OS reclaims pages behind the framework's pointers without coordination, it can corrupt state and crash execution. The challenge is to restore OS authority over physical pages while keeping the framework in charge of logical tensors and operator execution, which is non-trivial to achieve.

(2) Model-aware paging. Even if the first challenge is solved, we need to further extend this memory management to LLM execution. Inference shifts the management granularity from tensors to structured layer instances, and LLM decoding repeatedly reuses weights across tokens with strong temporal locality. A naive on-demand swapping can cause repeated reloads and long storage stalls, even when there is strong temporal locality across tokens.

One possible solution is to build custom inference engines with tightly controlled memory management in low-level C++/CUDA [16, 26, 29, 37, 53, 66]. These engines can be efficient, but they impose substantial costs in reduced flexibility, limited model/operator coverage, and recurring engineering effort as model architectures and optimizations evolve.

Our key idea is to retain the universality and fast evolution of general frameworks while restoring deterministic, deployment-ready memory behavior by decoupling physical memory lifecycle from the framework allocator and grounding it in OS mechanisms. In this paper, we propose ExoMem, a middleware that separates each LLM into a lightweight shell (topology and operator logic) retained by the framework and a heavyweight payload (weights) anchored in OS-managed mappings. ExoMem introduces a dormant state between active and destroyed: when weights are logically releasable, their virtual addresses remain stable for correctness, but their physical pages become immediately reclaimable by the OS under memory pressure and reusable at a low cost when still resident. By leveraging standard OS page caching and replacement, ExoMem effectively provides an implicit multi-tier hierarchy suited to unified memory.

We implement ExoMem as a lightweight middleware for mobile edge devices using Python, C++ and CUDA, packaged as a native PyTorch extension via pybind11 (open source link: https://github.com/ExoMem-repository/Code). It is a plug-and-play middleware that requires no modifications to the framework or model code. We evaluate ExoMem on NVIDIA Jetson devices (including Jetson Orin NX 16 GB and AGX 64 GB with JetPack 6.0 and an NVMe SSD) using representative LLMs, including Qwen2.5-7/14/72B, a vision model Qwen2.5-VL-7B, and a sparse MoE model (Qwen1.5-MoE-A2.7B), and compare against both framework-based baselines (*e.g.*, HuggingFace Accelerate-style constrained inference, naive layer-wise loading) and specialized engines (*e.g.*, llama.cpp and server-inspired offloading systems). Extensive experiments demonstrate that ExoMem achieves memory efficiency comparable to highly-optimized custom inference engines while reducing memory consumption by 78.9–87.6% compared to other approaches. It also achieves 2.1–51.7× higher throughput efficiency than all baselines. In summary, this paper makes the following contributions:

- Identify the framework-OS mismatch as a key reason that limits the local execution for large-LLM inference on mobile edge devices.
- Design an OS-governed middleware that makes physical memory reclaimable and peak-safe without breaking framework compatibility.
- Develop a prototype system and conduct extensive experiments with various LLMs to evaluate the performance of ExoMem under various settings.

2 Background and Motivation

2.1 LLM Inference on Mobile Edge

Demand for high-quality local inference. Many edge workloads increasingly need the kind of long-context, cross-modal reasoning that only larger LLMs reliably provide, for example, triaging and summarizing satellite observations, or interpreting autonomous driving logs and telemetry [6, 25, 61]. In these applications, the cost of a mistake is often paid by downstream operations, *e.g.*, a missed anomaly, a misprioritized scene, or an incomplete summary can waste operator time, bandwidth, or compute cycles. When connectivity is limited or privacy policies restrict off-device processing, the device cannot try in the cloud and fall back locally. This makes local inference with relatively uncompromised model quality an important capability.

Severe memory constraints. The central bottleneck on mobile edge is not only that DRAM is small in absolute terms, but also that only a fraction of it might be available to the LLM at runtime. Table 1 breaks the shared memory budget into pieces that are easy to overlook. First, a fixed portion is consumed before the LLM model does any useful work, *e.g.*, the OS, drivers, and the CUDA runtime together reserve several gigabytes that the application cannot reclaim. Second, realistic deployments rarely dedicate the device to LLM inference alone. Domain services such as satellite image processing or autonomous driving modules often run continuously and consume multiple additional gigabytes.

Platform	Total	OS	RT	App	Avail	Max Model
Orin Nano	8	1.1	1.5	—	5.4	2B
Orin NX	16	1.4	1.5	—	13.1	6B
+ Sat. tasks	16	1.4	1.5	6.2	6.9	3B
+ Aut. tasks	16	1.4	1.5	4.1	9.0	4B
AGX Orin	64	1.8	1.5	—	60.7	30B

Table 1: Memory budget breakdown (GB) and largest FP16 model size that can fit fully in DRAM on NVIDIA Jetson platforms. "OS" includes Linux and drivers; "RT" is the CUDA runtime; "App" is co-resident domain workloads; "Avail" is remaining headroom for the LLM.

Once these mandatory components are accounted for, the remaining headroom can be limited. On a 16 GB Orin NX, the available (Avail) budget drops from about 13.1 GB to about 6.9 GB with representative satellite tasks [41], and to about 9.0 GB with an autonomous driving stack [63]. To understand this limit, in Table 1, we use the maximum LLM model that can run in the remaining memory as a reference. In FP16, 1B parameters require about 2 GB just to store weights [64], before counting the KV cache and temporary buffers. Under a 6.9 GB headroom, even keeping a 3B model fully resident is already near the limit, and that is far below the size range where strong reasoning typically emerges.

This gap persists even on the largest Jetson-class devices. A 64 GB AGX Orin can leave roughly 60.7 GB for the application in the best case, which corresponds to around a 30B model in FP16 if weights are held entirely in memory. In contrast, a 72B model needs around 144 GB just for FP16 weights. This means that, for high-quality models, fitting the model in DRAM is not a viable design on mobile edge.

Layer-wise execution becomes mandatory. Given the above, running large models locally requires treating weights as an out-of-core working set. Layer-wise swapping is the natural solution [3, 4, 28, 47, 48], *i.e.*, load the next layer's weights from external storage, execute its operators, and then release those weights before moving on. This trades throughput for feasibility, which is acceptable for many edge workflows that are not interactive at the per-token level. In practice, token generation speed in these settings is primarily limited by device compute and storage bandwidth. The tier-one requirement is not *fast*, but *high-quality* and *memory-feasible*: a large LLM must stay within a fixed budget throughout inference without fragile peaks or gradual drift that triggers an OOM. Next, we show why a straightforward swapping fails to meet this requirement on mobile edge.

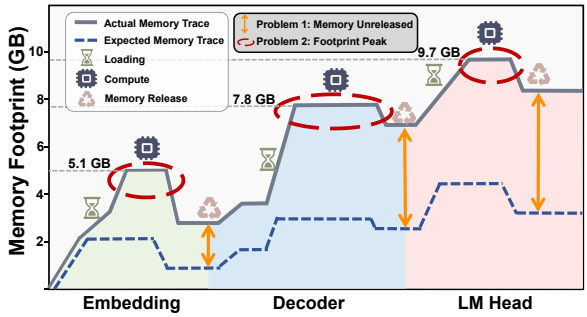

Figure 1: Physical memory footprint during layer-wise LLM execution on a Jetson Orin NX (Qwen2.5-7B, FP16). The actual trace diverges from the expected bounded sawtooth, showing (1) a rising baseline after release and (2) per-layer transient peaks during computation.

2.2 Problems of Swapping on UMA

In principle, layer-wise swapping should produce a bounded, repeatable memory footprint. Each layer follows the same three steps: load weights, run computation, then release weights. If the device behaves ideally, the physical memory trace should look like a sawtooth, *i.e.*, memory rises during loading, stays roughly flat during computation, and drops back after release, with only a slow upward trend from unavoidable states such as the KV cache and CUDA runtime.

Figure 1 shows that this expectation breaks in practice. We run a minimal Qwen2.5-7B-FP16 setup on a 16 GB Jetson Orin NX [51], and execute three representative components (embedding, one decoder layer, and the LM head) using a native swapping-style loop. After each component finishes, we explicitly delete the corresponding tensors and invoke `torch.cuda.empty_cache()` and `gc.collect()` to encourage reclamation. In this experiment, we track system-level physical memory usage [38], not only framework-reported *allocated* bytes, because the latter can hide memory retained inside framework's internal pools. The dashed line shows the expected bounded behavior, while the solid line shows the actual physical memory footprint. Two deviations stand out clearly, *i.e.*, a staircase-like baseline increase after each release, and a large transient peak during each compute.

Problem 1: Released layers do not translate into reclaimed physical memory. After each layer finishes, the footprint should drop close to the pre-layer baseline. Instead, the solid line in Figure 1 does not return to the previous level. The baseline grows upward across the embedding, decoder, and LM head phases, forming an irreversible staircase. This behavior is a form of **lazy reclamation** that is harmless to discrete systems but damaging on unified memory.

The reason is that general-purpose deep learning frameworks are optimized for private caching allocators. When a tensor is freed, the allocator commonly keeps the underlying memory in an internal pool for future reuse, rather than returning it to the OS. From the framework's perspective, this is "free" memory. From the OS perspective, the physical pages are still in use, so they continue to consume the shared memory. The problem is amplified by the structure of layer-wise execution: successive layers often request different tensor sizes, so previously cached layers may not be reusable due to fragmentation or size-class mismatches. The allocator then requests additional pages from the device, even though substantial memory is sitting idle inside the pool. Over a long autoregressive decoding run, this causes physical memory usage to accumulate and eventually crosses the budget, producing unpredictable OOM failures.

Problem 2: Implicit copies create large transient peak usage. The second problem is the pronounced peak that appears when computation begins for each layer, highlighted in Figure 1 (for example, 5.1 GB for embedding, 7.8 GB for the decoder phase, and 9.7 GB for the LM head). These peaks are caused by implicit copying in framework's data path.

During swapping, weights are typically brought in from storage through OS-managed mechanisms such as file-backed mappings and the page cache. When the framework prepares these weights for GPU kernels, it often performs a `memcpy` into its own managed allocation (for contiguity, alignment, or allocator policy reasons). On UMA, both the source region and the destination region still reside in the same physical DRAM pool. As a result, two large replicas can exist simultaneously during the copy and kernel launch window. Even if the duplicate is later freed, the peak happens at exactly the time when the system is most vulnerable. In tight budgets, these transient peaks can trigger an OOM issue even when the post-release baseline might have fit.

The framework-OS semantic gap. Both problems come from the same underlying issue: the framework's logical view of tensor lifetimes does not match the OS's physical view of page ownership on UMA devices. Mobile-oriented deep learning frameworks such as MNN and ExecuTorch [23, 36] manage memory in the same way and experience the same problem. This framework-OS gap thus reflects a fundamental system-level constraint. With smaller models (like DNNs), this mismatch is often masked because the working set remains comfortably within the allocator's pool and the device never approaches resource limit. With large LLMs, where layers are huge and swapping repeats for many tokens, the mismatch becomes a system-level failure mode, *i.e.*, memory that is logically released remains physically trapped, and implicit copies create budget-breaking peaks.

This thus motivates the design principle of our system, that is, keep the framework in control of model structures and operators, but restore OS authority over physical pages so that *release* truly means *reclaimable*. By doing this, the swapping operation does not require a second in-DRAM replica on the unified memory of mobile edge devices.

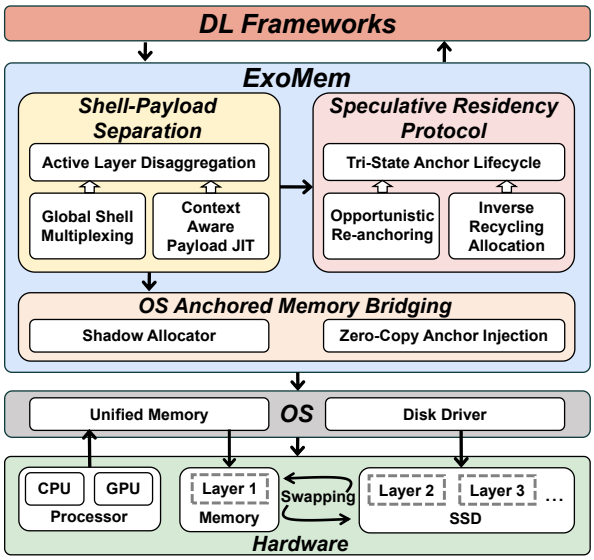

Figure 2: Architecture of the ExoMem design.

3 System Design

Guided by the principle above, we present ExoMem, a lightweight memory-management middleware for UMA edge devices. As shown in Figure 2, ExoMem sits between the deep-learning framework and the OS. It intercepts framework memory operations, replaces them with deterministic OS primitives, and creates a direct, zero-copy path that maps layer weights from disk into unified memory and then into framework tensors, without changing the model code.

3.1 OS Anchored Memory Bridging

To eliminate the non-deterministic lazy reclamation inherent in framework-managed memory on mobile edge, the *OS anchored memory bridging* design makes the OS the source of truth for the *physical* lifetime of large tensor buffers, while leaving the framework in charge of *logical* tensor semantics. The key is to decouple each tensor into

- A lightweight logical object T (metadata only, kept inside the framework), and
- A heavyweight, OS-owned physical backing store, called an *anchor* α (data only, kept in OS-governed memory).

ExoMem allocates, pins/unpins, and ultimately releases α using deterministic OS primitives, yet presents T to the framework as a native tensor. As illustrated in Figure 3, the workflow of this design is as follows:

① ExoMem intercepts a framework allocation request.
② ExoMem bypasses the framework's private allocator and allocates the backing pages via the OS/driver, creating an anchor α that the OS can directly account for and reclaim.
③ ExoMem then injects the anchor-backed pointer into the tensor metadata T, so the framework perceives T as a normal tensor and executes without extra copies.

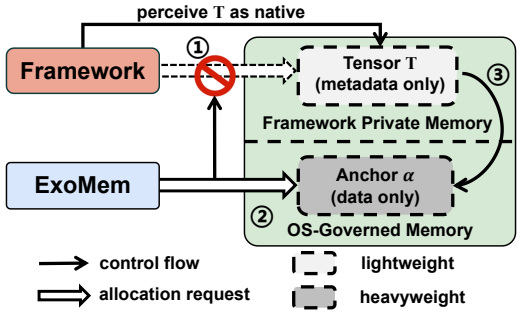

Figure 3: Execution workflow of OS anchored memory bridging, including ① intercept allocation, ② allocate OS-owned anchor α, and ③ inject α into tensor T, so that the framework perceives it as native.

When T is released, ExoMem is notified immediately and can deterministically transition α to the desired next state (*e.g.*, unpin + release to the OS, or hand off to a higher-level caching policy in §3.3). We realize this bridge with two components: a *shadow allocator* and *zero-copy anchor injection.*

3.1.1 Shadow Allocator. Deep learning frameworks commonly hide large allocations behind a private memory pool, which weakens OS visibility and makes reclamation timing framework-dependent. ExoMem addresses this by placing a *shadow allocator* at the middleware boundary, which has the following two responsibilities:

(1) Allocator bypass and OS-visible allocation. For each framework request, the shadow allocator diverts the allocation away from the framework pool and instead invokes OS/driver primitives to obtain and map the required pages. This makes the allocation explicitly visible to the OS accounting and eligible for immediate OS-controlled reclamation.

(2) Physical anchor establishment. The shadow allocator materializes the returned backing store as an anchor α (size, mapping, pin state, and ownership), and maintains the metadata needed to later unpin, remap, recycle, or release it deterministically, independent of the framework's internal caching and reference-counting heuristics.

3.1.2 Zero-Copy Anchor Injection. After ExoMem allocates an anchor α, the remaining challenge is *transparency*: if the framework treats α as external memory, it may silently copy data into its own buffers, causing both latency and duplicated memory footprint. We therefore inject α directly into the framework's tensor abstraction so that operators can run on the OS-managed pages *in place.* This mechanism operates in four phases as follows:

Phase 1: Anchor locking. The shadow allocator first marks anchor α as page-locked (pinned) via driver interfaces[1] [39].

[1]We pin pages while they are being accessed by GPU/compute kernels, and unpin immediately afterwards. This minimizes system-wide memory pressure and returns full reclaim authority to OS outside the active window.

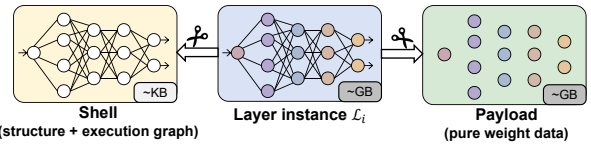

Figure 4: Disaggregation of a layer instance into a lightweight *shell* (framework-governed) and a heavyweight *payload* (OS-governed).

This serves a dual purpose for prohibiting OS swapping to guarantee physical stability and forcing the GPU memory management unit to map CPU-side virtual addresses directly to GPU-accessible device pointers.

Phase 2: Pointer injection. The deep learning framework instantiates only the tensor's metadata object without allocating native storage. We intercept this initialization and directly write the generated device pointer into the metadata's underlying storage field. This seamlessly binds the logical semantic object to the physical OS anchor.

Phase 3: Framework execution. During computation, the framework's operator dispatcher verifies the device pointer's validity purely from the metadata. Oblivious to the data's true OS-managed origin, the GPU directly accesses the physical anchor in main memory over the system bus, executing zero-copy computation transparently and efficiently.

Phase 4: Lifecycle stewardship. To enforce deterministic release, we attach a custom *safe-deleter* to the tensor's metadata. Bypassing the framework's passive garbage collection, this injected deleter instantly notifies the OS kernel to release the physical anchor the moment the tensor's reference count drops to zero, achieving immediate physical reclamation.

3.2 Shell-Payload Separation

While the previous module establishes deterministic memory control at the atomic tensor level, the execution scheduling of LLMs inherently revolves around highly encapsulated semantic units: *layers*. Traditional deep learning frameworks manage these layers as tightly coupled, monolithic objects.

A layer instance $\mathcal{L}_i$ contains not only weight data but also complex operator topologies and strict execution sequences. If managed purely at the basic tensor level, the framework would be forced to repeatedly instantiate heavy computation graphs containing hundreds of operator objects for every single layer, incurring severe initialization latency. To eliminate this overhead, we propose a structural abstraction that elevates the management granularity from low-level atomic data (tensors) to high-level instance objects (layers). As depicted in Figure 4, we disaggregate the monolithic layer into a *lightweight shell* and a *heavyweight payload*.

3.2.1 Active Layer Disaggregation. To achieve a complete decoupling of layer-level computation logic from its state

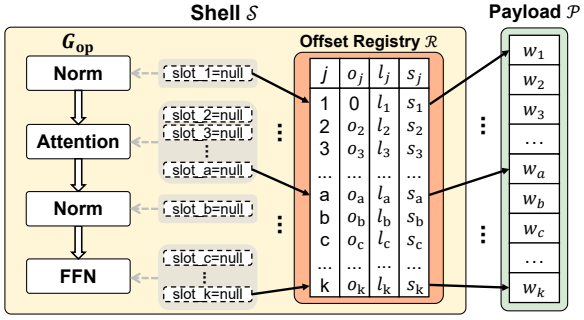

Figure 5: A stateless shell containing the operator execution graph with null slots, while a flat binary payload is indexed via an offset registry.

data, a layer object $\mathcal{L}_i$ is decomposed into two system abstractions: $\mathcal{L}_i = \mathcal{S}_i \oplus \mathcal{P}_i$, where $\mathcal{S}_i$ is a stateless *shell* and $\mathcal{P}_i$ is the *payload* for layer i, as illustrated in Figure 5.

Shell $\mathcal{S}$. Acting as a stateless execution engine, the shell contains not only the metadata of the layer structure, but also the complete operator execution graph G_{op} (representing the topological connections of operators like Attention, FeedForward, and Norm) [5]. Unlike native layers, the shell provisions k parameter slots $\{\texttt{slot}_j\}_{j=1}^{k}$ initialized as null pointers, holding no actual data. In addition, the shell maintains a lightweight offset registry $\mathcal{R} = \{(o_j, l_j, s_j)\}_{j=1}^{k}$ that records the exact start offset o_j, byte length l_j, and tensor shape s_j for each weight tensor residing in the payload. Formally, a shell $\mathcal{S}$ is defined as:

$$\mathcal{S} = \left(G_{\text{op}},\ \{\texttt{slot}_j = \texttt{null}\}_{j=1}^{k},\ \mathcal{R} \right). \tag{1}$$

Since all parameter slots are empty, the shell is extremely lightweight (typically a few kilobytes) and resides permanently in memory as a universal computation template.

Payload $\mathcal{P}$. The payload consists of pure weight data stripped of all topological semantics. Essentially, each payload is a concrete instantiation of the OS-managed physical anchor α (introduced in §3.1). Existing frameworks typically store weights in serialized formats, forcing CPU-intensive parsing, deserialization, and scattered memory allocations during loading [5]. To resolve this, we abandon hash-based structures. Instead, we concatenate all k discrete weight tensors of a layer into a single, compact, 1D flat binary block, strictly adhering to the topological execution order:

$$\mathcal{P} = [\ w_1 \parallel w_2 \parallel \cdots \parallel w_k\ ], \tag{2}$$

where w_j denotes the flattened byte sequence of the j-th weight tensor. Governed by the aforementioned registry $\mathcal{R}$, this dense packing simplifies complex weight lookups into $O(1)$ pointer arithmetic operations, completely eliminating the CPU parsing overhead and serialization latency.

3.2.2 Homogeneity-based Global Shell Multiplexing. Modern LLM architectures exhibit layer-wise homogeneity, *i.e.*,

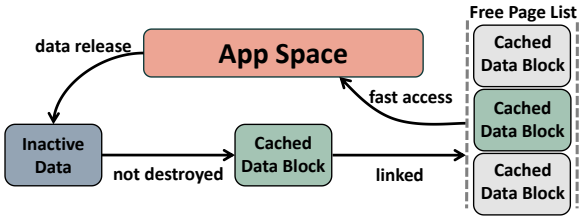

Figure 6: Standard OS page replacement workflow. Released memory is retained in the free page list as an inactive cache, allowing subsequent accesses to bypass slow disk I/O via fast remapping.

the N decoder layers share an identical operator topology and execution sequence, differing solely in the payload data injected into them. Exploiting this observation, we construct a single global template shell $\mathcal{S}^*$ on the GPU during system initialization. All N layers multiplex this singleton shell by

$$\mathcal{L}_i = \mathcal{S}^* \oplus \mathcal{P}_i, \quad \forall\, i \in [0,\, N). \tag{3}$$

Logically, this shell acts as a *computational proxy* for all isomorphic layers. This multiplexing yields two systemic advantages. First, it eliminates the CPU and memory overhead associated with building $O(N)$ redundant computation graphs. Second, since all layers repeatedly reuse the same shell $\mathcal{S}^*$, indicating they execute the exact same host code and kernel launch sequences, the CPU instruction cache hit rate will increase, reducing the dispatcher launch overhead.

3.2.3 Context Aware Payload JIT Binding. To dynamically inject static data into the execution logic at runtime, we design a context-aware payload JIT binding mechanism. As illustrated in Figure 5, when the inference advances to layer i, the dispatcher queries the global offset registry $\mathcal{R}$ within $\mathcal{S}^*$. The system retrieves the $O(1)$ physical coordinates $\{(o_j, l_j, s_j)\}_{j=1}^{k}$ of all k member parameters and mounts the payload $\mathcal{P}_i$ into the shell through the zero-copy injection:

$$\mathcal{S}^*.\texttt{slot}[j] \leftarrow \textsc{Inject}\big(\mathcal{P}_i[o_j : o_j + l_j],\ s_j\big),\ j = 1, \ldots, k. \tag{4}$$

Subsequently, the framework executes the standard forward pass, producing the hidden activation $\mathbf{h}_i$ from the previous layer's output $\mathbf{h}_{i-1}$:

$$\mathbf{h}_i \leftarrow \mathcal{S}^*.\textsc{Forward}(\mathbf{h}_{i-1}). \tag{5}$$

Upon completion of the computation, ExoMem immediately disconnects the connection between the shell and the payload of layer i by resetting the placeholders:

$$\mathcal{S}^*.\texttt{slot}[j] \leftarrow \texttt{null}. \tag{6}$$

Such immediate unbinding frees the mount points for the subsequent payload P_{i+1}, and it also ensures the instantaneous release and reuse of physical memory resources.

This mechanism ultimately transforms layer-wise inference into a highly efficient streaming pipeline, wherein successive payloads $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{N-1}$ flow seamlessly through the template $\mathcal{S}^*$, guaranteeing a minimal memory footprint and zero redundant graph instantiation.

3.3 Speculative Residency Protocol

Unlike traditional models (*e.g.*, DNNs) that process layers in a single pass, LLM inference is inherently *autoregressive*, demanding repeated access to all layers across successive token generation steps. Conventional layer-wise swapping ignores this deterministic temporal locality, naively loading and destroying the same layer's weights in every cycle. This oblivious strategy inevitably triggers unnecessary I/O.

To exploit the repeated access pattern of LLMs, and inspired by standard OS page replacement policies as shown in Figure 6, which keep frequently-accessed data in memory [34, 40], we design the *speculative residency protocol*. By establishing an elastic, OS-backed buffer mechanism, it transforms physical DRAM into a zero-overhead cache, opportunistically reusing layer payloads to minimize disk (SSD) I/O on memory-constrained mobile edges.

3.3.1 Tri-State Anchor Lifecycle. Traditional application-level memory lifecycle management is often binary: Active (pinned, mapped, and actively used by the framework) or Destroyed (freed and wiped). To retain weight data in physical memory without inflating the framework's perceived memory budget, we introduce a transitional Dormant state, redefining the payload $\mathcal{P}$'s lifecycle as:

$$\textsc{State}(\mathcal{P}) \in \{\textsc{Active},\ \textsc{Dormant},\ \textsc{Destroyed}\}. \tag{7}$$

When the JIT injector detaches a layer's payload, rather than actively destroying the anchor, the system transitions it to the Dormant state, exhibiting a crucial duality:

- **Logical release** (*OS perspective*): We invoke standard deallocation primitives, marking the physical pages as clean and reclaimable. To the OS, this memory is entirely freed and available for immediate reassignment to other processes under memory pressure.
- **Physical residency** (*data perspective*): Although logically freed, the OS kernel defers actual data erasure until the pages are physically reallocated to a new process. Consequently, the original weight data remains perfectly intact on the physical medium.

This duality effectively converts dormant payloads into an implicit, OS-managed L3 cache, preserving the opportunity for zero-I/O reuse without hoarding memory or blocking high-priority system allocations.

3.3.2 Inverse Recycling Allocation. The residency window of a Dormant anchor depends entirely on how quickly the OS reallocates and overwrites its physical pages. To maximize cache locality, standard OS allocators employ a "hot-reuse" policy (*e.g.*, LIFO page allocation), preferentially reassigning

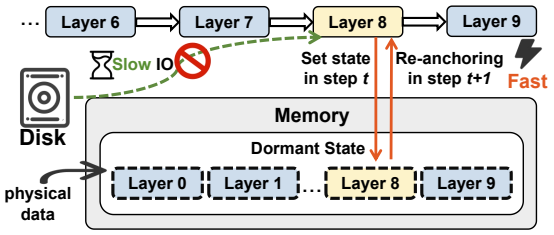

Figure 7: Opportunistic re-anchoring across autoregressive generation steps. By preserving layer payloads (*e.g.*, Layer 8) in an OS-managed dormant state during step t, the system can re-anchor the physical memory in step $t+1$, which bypasses the slow disk I/O bottleneck.

the most recently freed memory. Unfortunately, this default behavior instantly overwrites our newly dormant payloads.

To counter this, the shadow allocator employs an *inverse recycling* strategy. When requesting fresh memory for the subsequent layer, it deliberately guides the OS kernel to bypass the physical address space that just transitioned into the DORMANT state. By artificially steering allocations toward cold regions, defined as idle memory regions not marked as DORMANT in ExoMem, this strategy maximizes the physical residency window of dormant data, deferring their eviction until system-wide memory pressure dictates it.

3.3.3 Opportunistic Re-anchoring. When the inference loops back to a previously executed layer, the system performs a speculative state check before initiating any disk I/O. As illustrated in Figure 7, if a target payload (*e.g.*, Layer 8) was transitioned to the DORMANT state in step t and remains intact in physical memory (*i.e.*, not yet overwritten by the OS), the system executes a rapid re-anchoring in step $t+1$. It re-establishes the page locks and restores the virtual-to-physical mappings, completely bypassing the slow disk I/O stack. If the pages have already been overwritten by the OS, the system seamlessly falls back to standard disk loading[2].

Furthermore, when system memory pressure inevitably forces the actual eviction of dormant pages, the strictly sequential and cyclical nature of LLM inference allows us to deploy an efficient replacement algorithm [9]. Unlike general-purpose caching where future access patterns are unpredictable, autoregressive decoding traverses layers deterministically. When currently executing layer i, the subsequent access sequence is $i+1, \ldots, N-1, 0, \ldots, i-1, i$. Thus, the layer $(i-1) \bmod N$ is guaranteed to be the furthest in the future. We formulate the $O(1)$ optimal eviction decision as:

$$\text{EVICT} = \arg \max_{j \in \text{DORMANT}} (j - i) \bmod N, \tag{8}$$

[2]While asynchronous look-ahead prefetching is a viable orthogonal optimization, its practical latency-hiding benefit is fundamentally limited here, where the brief execution time of a single layer cannot sufficiently mask the severe disk I/O bottleneck of loading massive weights.

which transforms the problem into a lightweight and practical mechanism by mathematically maximizing cache hits under the memory constraints. In general, ExoMem benefits most when the available memory budget can accommodate a substantial portion of dormant payloads (yielding high re-anchoring hit rates) and when storage bandwidth is limited.

4 Evaluation

We develop ExoMem using Python 3.10, C++ 17, CUDA 12.6, and PyTorch 2.5.0. The entire middleware is implemented as a native C++/CUDA extension exposed to Python via `pybind11`. It is packaged as a plug-and-play module, requiring no modifications to model code and framework internals. Porting ExoMem to a new framework requires additional development in three aspects, including (1) allocator hooks to redirect large backing buffers to OS-managed anchors, (2) tensor-storage adaptation to bind flat weight payloads to native tensor slots, and (3) runtime/driver binding to connect OS-managed anchors to accelerator execution.

4.1 Experimental Setups

4.1.1 Models and Configurations. We evaluate ExoMem on three representative families of large models to demonstrate its generality across architectures and inference pipelines. Unless otherwise noted, all models run in FP16.

- **Dense language models.** We use Qwen2.5-7B, Qwen2.5-14B and Llama3.1-8B as primary models for evaluation. For stress-test under extreme memory pressure, we additionally include Qwen2.5-32B and 72B in micro-benchmarks.
- **Vision-language model (VLM).** We use Qwen2.5-VL-7B to validate efficiency for multimodal inference.
- **Mixture-of-Experts (MoE).** We use Qwen1.5-MoE-A2.7B to test compatibility with sparse execution, where token-dependent expert routing induces irregular and input-dependent memory behavior.

4.1.2 Baselines. We compare ExoMem against both general-purpose frameworks and specialized inference engines:

- **PyTorch API (PTA).** The official Hugging Face Accelerate mechanism for memory-constrained inference [5].
- **PyTorch Naive (PTN).** A custom layer-wise pipeline that explicitly loads and frees weights via PyTorch primitives (`del` and `torch.cuda.empty_cache()`), showing the limitations of default caching allocators under tight budgets.
- **FlexLLMGen (FLG).** A SOTA offloading engine that enforces an explicit GPU-CPU-disk hierarchy [47].
- **llama.cpp (LC).** A representative hand-optimized inference engine implemented in C/C++ with explicit memory management [16]. We report three configurations to cover its operating spectrum: "LCC" (CPU-only), "LCG" (GPU-only), and "LCH" (mixed execution).

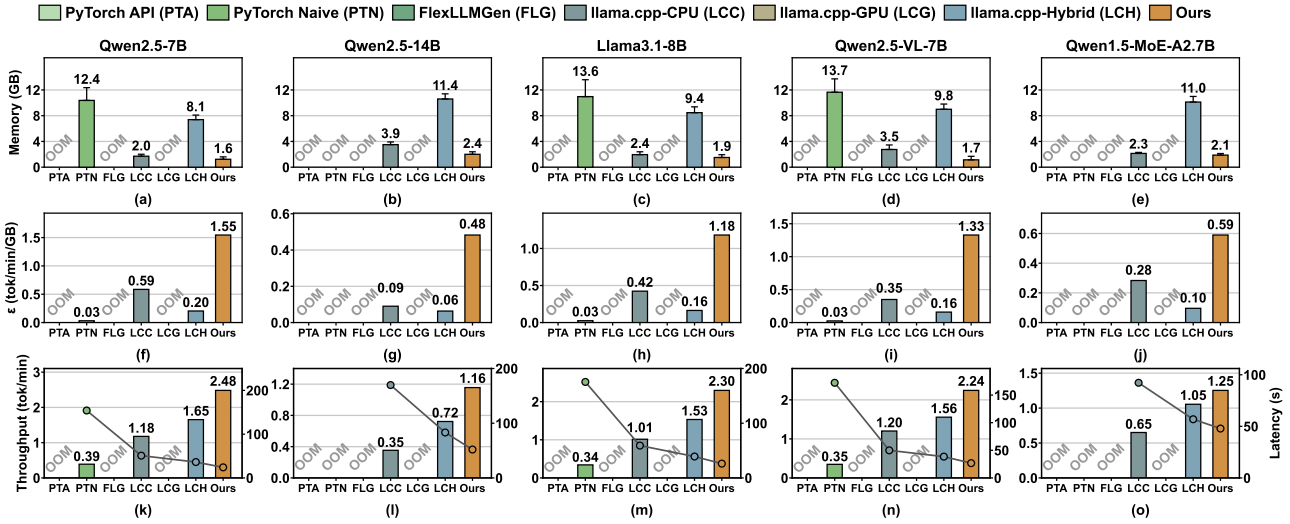

Figure 8: Overall performance across three types of models: dense language, VLM and MoE.

In addition to model weights, inference incurs dynamic memory overheads, including KV caches and intermediate activations, which persist across decoding steps and grow with the context length. To ensure a fair comparison, we configure all methods with the same maximum context length (prompt length plus maximum output length), and each method can be evaluated under the same worst-case activation footprint.

For ExoMem, we further reduce fragmentation and allocation overhead by isolating these runtime artifacts into dedicated memory pools. Specifically, we pre-allocate a contiguous activation pool sized for the maximum context length, from which KV caches and other dynamic tensors draw space. This supports efficient append-style growth without repeated reallocations, while keeping weight management (the focus of ExoMem) orthogonal to activation handling. By default, ExoMem swaps one layer at a time to ensure the memory reserved for weights does not exceed the size of LLM's maximum single layer, thus minimizing memory requirements.

4.1.3 Testbeds. Our primary testbed is NVIDIA Jetson Orin NX, a mobile edge platform with an Ampere GPU and 16 GB unified memory, running Ubuntu 20.04 with JetPack 6.0. To evaluate scalability across resource tiers (§4.4.1), we additionally use NVIDIA Jetson AGX Orin with 64 GB memory. Both devices store model weights on an NVMe SSD [27].

4.2 Overall Performance

In this section, we conduct experiments and evaluate each method by comparing the following two main metrics:

(1) **Memory Footprint.** We report *peak* and *average* physical memory usage. Peak memory is the critical constraint, dictating the minimum DRAM required to prevent OOM.
(2) **Throughput Efficiency.** To fairly assess deployment viability across varying model scales on resource-constrained devices, similar to recent research [26, 31, 68], we examine the throughput efficiency $\epsilon = \frac{\text{tokens/min}}{M_{\text{peak}}\ (\text{GB})}$, which measures the inference speed against memory consumption. We also report raw throughput and per-token latency to show the absolute inference performance.

4.2.1 Performance on Dense Language Model. This experiment compares ExoMem against each baseline method.

Setup. We first evaluate Qwen2.5-7B, Qwen2.5-14B and Llama3.1-8B on Jetson Orin NX with a 6K-token prompt. For the LC-Hybrid (LCH) baseline, we offload 50% and 33% of layers to the GPU for the 7B and 14B models, respectively. A result is marked "OOM" if the method fails to complete the run under the same context-length setting.

Results. Figures 8 (a)–(c), (f)–(h), and (k)–(m) report the results of the three dense language models.

Memory Footprint. As shown in Figure 8 (a) and (b), ExoMem is the only framework-compatible approach among PTA, PTN, and FLG that successfully runs both Qwen2.5-7B and Qwen2.5-14B. On Qwen2.5-7B, ExoMem achieves a peak memory of 1.6 GB, improving over LCC (2.0 GB) and dramatically reducing the peak relative to LCH (8.1 GB) and PTN (12.4 GB). PTA and FLG fail with OOM even for 7B. Notably, llama.cpp's CPU-only configuration (LCC) achieves a relatively small footprint (2.0 GB) because CPU-only execution bypasses the GPU runtime entirely, avoiding the framework-OS mismatch at the cost of significantly lower throughput (see below). Once GPU execution is enabled, the mismatch causes a sharp increase in memory usage of LCG and LCH. In contrast, ExoMem preserves GPU-accelerated execution while maintaining a low memory footprint. As the model scales to 14B, PTA/PTN/FLG and LCG all are OOM, whereas ExoMem remains memory efficient, peaking at only 2.4 GB (vs. 3.9 GB for LCC and 11.4 GB for LCH). The result

generalizes to Llama3.1-8B in Figure 8(c), where ExoMem peaks at 1.9 GB and outperforms all baselines. These results indicate that ExoMem enables deterministic, low-overhead weight reclamation, which is essential for scaling to larger models under tight unified-memory limits.

Throughput Efficiency. As shown in Figures 8 (f)–(h), ExoMem consistently delivers the highest throughput efficiency across all successful settings. For Qwen2.5-7B, ExoMem reaches $\epsilon = 1.55$, exceeding LCC (0.59) by 2.6×, LCH (0.20) by 7.8×, and PTN (0.03) by over 50×. This advantage is primarily driven by ExoMem's ability to prevent allocator-induced memory increase that forces conservative execution modes and exploits opportunistic re-anchoring to reuse cached weights (67.9% hit rate in this workload), thereby avoiding slow I/O on the critical path. For Qwen2.5-14B, ExoMem further widens the gap, achieving $\epsilon = 0.48$, which is 5.3× higher than LCC (0.09) and 8.0× higher than LCH (0.06). For Llama3.1-8B, ExoMem achieves a throughput efficiency of 1.18 tok/min/GB, higher than 0.42 for LCC and 0.16 for LCH.

To further understand the absolute inference performance of each approach, we also show the raw token-generation throughput and per-token latency values. Figures 8 (k)–(m) show that ExoMem consistently achieves the highest raw throughput (left-axis) and lowest per-token latency (right-axis) among all baselines across the three dense language models, *e.g.*, 1.50–1.61×, 2.10–3.3× and 6.36–6.76× faster than LCH, LCC and PTN, respectively. Overall, these results demonstrate that by strictly controlling the physical memory lifecycle, ExoMem sustains GPU-accelerated decoding while demanding a small amount of memory, whereas prior methods either OOM or sacrifice efficiency due to excessive memory overheads and I/O stalls. Furthermore, ExoMem primarily targets local, full-precision inference workloads where generation quality and memory feasibility take precedence over interactive response latency.

4.2.2 Performance on VLM and MoE Model. To evaluate architectural generality, we further evaluate ExoMem using a vision-language model (VLM) and a mixture-of-experts (MoE) model. As a transparent middleware, ExoMem requires no model-specific code changes; the same abstractions and runtime mechanisms apply unchanged across both models.
Setup. We deploy Qwen2.5-VL-7B (3K-token prompt, 450×450 image) and Qwen1.5-MoE-A2.7B (6K-token prompt) on the Jetson Orin NX. For the LCH baseline, we offload 50% (VLM) and 33% (MoE) of the layers to the GPU.
Results. Figures 8 (d)–(e), (i)–(j), and (n)–(o) report the results of the VLM and MoE models.

Memory Footprint. As shown in Figures 8 (d) and (e), ExoMem minimizes memory consumption across both architectures. On the VLM, it peaks at only 1.7 GB (averaging 1.1 GB), halving LCC's 3.5 GB peak and significantly outperforming LCH's 9.8 GB and PTN's 13.7 GB. Because the OS-anchored bridge deterministically reclaims payloads, it safely absorbs transient activation peaks from the vision encoder without cross-stage memory accumulation. On the MoE model, ExoMem retains a 2.1 GB peak, outperforming LCC (2.3 GB) and LCH (11.0 GB). By packing each MoE layer as a single flat binary block that is instantly released after computation, the memory footprint remains bounded to a single layer's capacity, which is agnostic to the total number of routed experts. Because expert structures within each MoE layer are typically identical, ExoMem's computation shell can be reused across all experts without additional instantiation. The I/O overhead can be further reduced by exploiting MoE routing to swap in only activated expert payloads.

Variant	Qwen2.5-7B		Qwen2.5-14B	
	Latency	Memory	Latency	Memory
w/o Shell-Payload	+12.8%	×7.2	OOM	
w/o IO-Opt	+61.1%	–	+39.3%	–

Table 2: Ablation study on memory and latency overheads incurred by disabling core components of ExoMem. "–" indicates no change.

Throughput Efficiency. As shown in Figures 8 (i) and (j), ExoMem also delivers the best throughput efficiency on both models. For Qwen2.5-VL-7B, ExoMem reaches $\epsilon = 1.33$, outperforming LCC (0.35) by 3.8× and LCH (0.16) by 8.3× (PTN achieves only 0.03). This indicates that ExoMem's shell-payload abstraction remains effective under the heterogeneous and irregular access patterns introduced by the vision encoder, enabling efficient reuse and avoiding unnecessary data movement. For Qwen1.5-MoE-A2.7B, ExoMem achieves $\epsilon = 0.59$, exceeding LCC (0.28) by 2.1× and LCH (0.10) by 5.9×. Figures 8(n) and (o) further show the absolute throughput and latency performance of each approach. Overall, these results show that ExoMem generalizes beyond dense LLMs. Deterministic reclamation and low-copy data paths remain effective for VLM and MoE inference, delivering higher efficiency without any architecture-specific engineering.

4.3 Ablation Study

We evaluate two variants of ExoMem against the full design to quantify the contribution of each core module:

- **ExoMem w/o Shell-Payload**: It disables the shell-payload abstraction (§3.2). Layers are instantiated and managed as heavy, native framework objects.
- **ExoMem w/o IO-Opt**: It disables the speculative residency protocol (§3.3). Payloads bypass the dormant state and are aggressively destroyed post-computation.

Table 2 summarizes the ablation experimental results.

Memory Viability vs. Shell-Payload: Disabling shell-payload separation increases the 7B model's peak memory by 7.2×

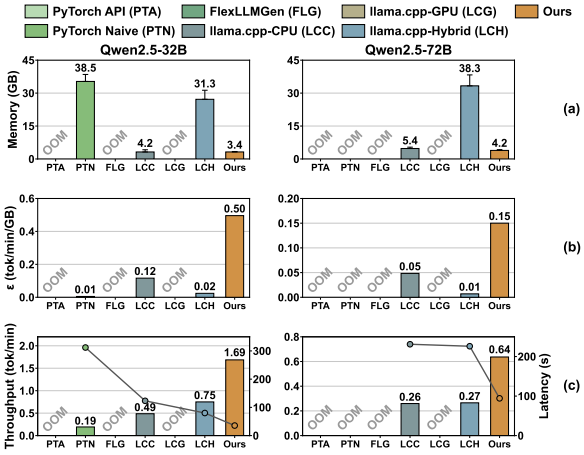

Figure 9: Performance of 32B and 72B models on the 64 GB Jetson AGX Orin.

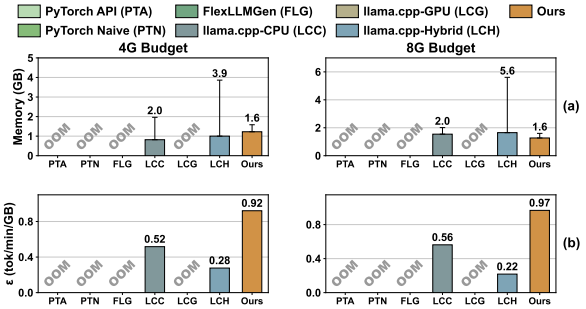

Figure 10: Performance on dynamic memory budgets.

	LRU	Clock	Random	Ours
4G Budget	0%	0%	4.2%	20.3%
8G Budget	0%	0%	19.5%	48.6%

Table 3: Re-anchoring hit rate under different page replacement policies.

and triggers a fatal OOM crash for the 14B model. This is a direct consequence of non-deterministic lazy reclamation, where logically freed pages remain trapped in the framework's opaque pool rather than returning to OS. Moreover, relying on native framework loading adds a 12.8% latency penalty due to redundant CPU-side weight parsing, deserialization, and implicit `memcpy` double-copying overheads.

Throughput vs. Speculative Residency: Disabling the speculative residency protocol brings massive latency penalties, *i.e.*, 61.1% for the 7B model and 39.3% for the 14B model, while keeping the peak memory footprint completely unchanged. Without dormant caching, the system degrades into a sequence of disk stalls, forcing a synchronous, full-disk reload at every autoregressive step. The 7B model suffers a more severe degradation because it loses a substantial 67.9% re-anchoring hit rate (compared to 14B's 34.7%).

4.4 Micro-Benchmarks

4.4.1 Impact of Hardware Scaling. To evaluate hardware scalability, we deploy the larger Qwen2.5-32B and -72B models on the 64 GB AGX Orin. For the LCH baseline, we offload 50% (32B) and 25% (72B) of layers to the GPU.

As shown in Figure 9, ExoMem scales efficiently for larger models. For the 32B model, it maintains a remarkably low peak memory of 3.4 GB, whereas PTN and LCH lead to 38.5 GB and 31.3 GB, respectively. Furthermore, ExoMem achieves an unmatched throughput efficiency of $\epsilon = 0.50$, outperforming LCC by 4.2× and LCH by 25×. On the other hand, even as the model scales to 72B, ExoMem's footprint barely inches up to 4.2 GB while sustaining the highest ϵ among all methods. ExoMem leaves ample memory headroom for other co-existing edge tasks. Figure 9(c) further compares the absolute throughput and latency of each method.

4.4.2 Impact of Dynamic Memory Budget. To simulate multi-task edge environments, we inject background processes on the Orin NX to constrain system memory to 4 GB and 8 GB during Qwen2.5-7B inference. Consequently, PTA, PTN, FLG, and LCG all suffer immediate OOM crashes. For LCH, we manually offload 1/7 and 1/4 of the layers under the 4 GB and 8 GB budgets, respectively.

Figure 10 shows that ExoMem's peak memory remains an invariant 1.6 GB across all budget constraints, as its footprint is strictly bounded by a single layer's payload. Conversely, LCH consumes 3.9 GB under the severe 4 GB limit, dangerously nearing system collapse. ExoMem also delivers high throughput efficiency, achieving $\epsilon = 0.92$ under 4 GB and $\epsilon = 0.97$ under 8 GB, both outperforming all baselines. As the memory budget relaxes, ExoMem's performance naturally improves because the speculative residency protocol seamlessly exploits the extra free pages to prolong dormant cache hits. Ultimately, ExoMem guarantees stable, predictable execution under severe external memory pressure, whereas competitors either fail or sacrifice safety margins.

4.4.3 Impact of Page Replacement Policies. In re-anchoring mechanism design (§3.3.3), we introduce a page replacement method for ExoMem. We compare it with other alternatives. Table 3 shows that traditional OS policies like LRU and Clock yield a 0% hit rate. Because LLM inference strictly cycles through layers (from 0 to $N-1$) at each decoding step, these policies suffer from complete cache thrashing, constantly evicting the exact layers needed for the next round. While Random replacement achieves marginal success by chance, our method explicitly leverages the deterministic execution sequence of the LLM. By evicting pages needed furthest in the future, it achieves hit rates of 20.3% and 48.6%, closely approaching the theoretical upper bound [9].

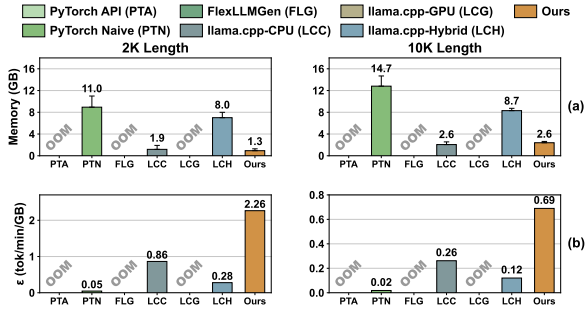

Figure 11: Performance on different context lengths.

4.4.4 Impact of Context Lengths. We evaluate Qwen2.5-7B using 2K and 10K token prompts on the Orin NX. As shown in Figure 11, ExoMem's peak memory scales modestly from 1.3 GB (2K) to 2.6 GB (10K). This growth strictly tracks the continuous expansion of the KV cache, yet the total memory consumption remains much lower than all baselines. ExoMem consistently outperforms in throughput efficiency, achieving $\epsilon = 2.26$ at 2K and $\epsilon = 0.69$ at 10K. The expected efficiency decay at 10K mathematically reflects the universally increased KV cache volume and heavier per-token computational burden inherent to long-context generation.

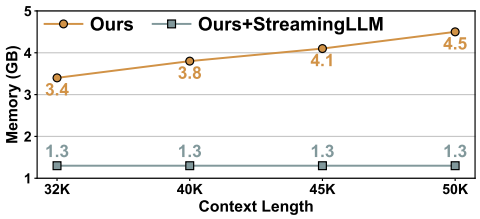

Figure 12: Peak memory footprint of ExoMem with and without StreamingLLM under long-context inference.

4.4.5 Compatibility with KV-Cache Optimization in Long-context Inference. ExoMem is complementary to KV-cache optimization to handle long-context inference. To verify this, we evaluate Qwen2.5-7B under long-context inference with context lengths ranging from 32K to 50K tokens, using ExoMem alone and ExoMem integrated with StreamingLLM [57]. As shown in Figure 12, the peak memory of ExoMem gradually increases from 3.4 GB to 4.5 GB as the context length grows, whereas integrating StreamingLLM maintains the peak memory at 1.3 GB throughout the experiment.

4.5 Case Study

In this subsection, we further evaluate ExoMem in a satellite edge intelligence case study for *safe co-location* of latency-sensitive perception and latency-tolerant LLM reasoning on a memory-constrained edge device. The goal is to verify that ExoMem's bounded footprint translates into (i) negligible interference to the primary real-time workload and (ii) predictable memory behavior that avoids unexpected OOM failures during concurrent execution.

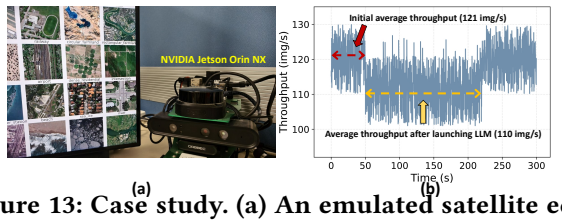

Figure 13: Case study. (a) An emulated satellite edge platform is deployed for SAR image classification. (b) The QoS (throughput) of ResNet, where the concurrent LLM is launched at t = 50 s and ends at t = 220 s.

Setup. We emulate an on-orbit telemetry workload on a Jetson Orin NX device as shown in Figure 13(a). The primary real-time perception task uses a ResNet-18 model [19] to classify continuous high-resolution remote satellite sensing data. We stream the orbital visual data from the RESISC45 dataset [13], which is a comprehensive benchmark containing 45 diverse scene classes, to the edge device. The vision pipeline processes standard $3 \times 224 \times 224$ inputs with a batch size of 32. Simultaneously, ExoMem launches Qwen2.5-7B as a background task with layer-wise swapping (to minimize the memory consumption) for continuous system log analysis. By parsing unstructured telemetry data and generating compressed diagnostic summaries locally, the LLM drastically reduces expensive downlink bandwidth overhead.

Metric	Peak (GB)	ResNet (img/s)	Throughput Efficiency
Exo	1.6	110	1.25
LCC	2.0	118	0.54
LCH	9.0	99	0.11

Table 4: Performance of ExoMem (Exo) compared with LCC and LCH baselines in case study, including peak memory, ResNet QoS, and LLM throughput efficiency.

Results. As shown in Figure 13(b), the throughput result demonstrates that the ResNet-18 pipeline achieves an initial average throughput of 121 images/s under strict memory isolation. After launching the log analysis by LLM at $t = 50$ s, the throughput drops to approximately 9.1%, maintaining a highly stable average of 110 images/s. From Figure 13(b), we can also see that worst-case throughput is not small, *e.g.*, no less than 102 images/s in this experiment. This result suggests that the background LLM avoids aggressive resource starvation, satisfying the QoS requirements for concurrent tasks, especially the real-time ones.

In Table 4, we further compare the performance of ExoMem (Exo) with other baselines (LCC and LCH) in this case study. The LCH baseline offloads LLM layers to the GPU, resulting in severe resource contention (9.0 GB peak) that degrades the throughput of the perception task to 99 images/s and reduces the throughput efficiency of the LLM to merely 0.11. While LCC avoids contention (2.0 GB, 118 images/s) by relying entirely on CPU-only inference, its LLM throughput

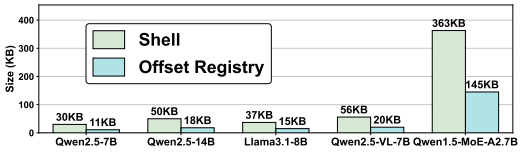

Figure 14: System overhead for different models.

efficiency is low, *i.e.*, at 0.54 in this experiment. In contrast, ExoMem preserves a smaller peak memory footprint (1.6 GB), maintains a relatively high QoS (110 images/s) of the perception task, and achieves the highest throughput efficiency (1.25) of the LLM inference.

4.6 System Overhead

During the inference time, ExoMem requires additional memory overhead to maintain the layer shell and offset registry for efficient JIT binding. In Figure 14, we measure this system overhead for each LLM used in the experiments. The result shows that ExoMem's overhead is small. For dense LLMs (Qwen2.5-7B/14B and Llama3.1-8B) and VLM (Qwen2.5-VL-7B), the shell and registry occupy only 30–56 KB and 11–20 KB, respectively. Even for the MoE model (Qwen1.5-MoE-A2.7B), where the more complex topology increases the metadata volume, the shell (363 KB) and registry (145 KB) remain well below 1 MB in total. These results indicate the low overhead of the ExoMem design, which is suitable for the mobile edge deployment.

5 Related Work

LLMs on mobile edge devices. Physical memory limits LLM deployment on mobile edge devices. Conventional approaches fit models into constrained memory via quantization [8, 24, 31, 45, 50], pruning [7, 15, 33, 45, 49], neural architecture search [21, 69], or distillation [18, 35, 42]. While effective at reducing footprints, these methods inherently trade reasoning capacity for resource efficiency, which is undesired for mission-critical applications demanding maximum accuracy. To preserve model integrity, alternative offloading systems swap or stream weights across GPU/CPU/storage tiers [3, 4, 28, 47, 48]. However, these approaches are designed for discrete memory hierarchies (*e.g.*, scarce GPU VRAM backed by abundant CPU RAM) and do not address the framework-OS memory-management mismatch on UMA. By contrast, ExoMem natively targets UMA on the mobile edge, where the CPU and GPU share the same DRAM pool. We identify that UMA stability failures stem from a framework-OS lifecycle mismatch, which we resolve through OS-anchored memory governance rather than lossy LLM model alterations. The principle of ExoMem can also be applied to NPU-based inference when the NPU runtime exposes the porting interfaces (§4) in the future.

Custom inference engines on mobile edges. Severe memory limitation has also motivated the development of specialized, bare-metal inference engines that bypass general-purpose frameworks. Systems like llama.cpp [16], TensorRT-LLM [37], vLLM [26], SGLang [66], and other edge-centric stacks [29, 53] employ hand-tuned memory allocators, paged KV caches, and custom block-swapping to minimize fragmentation. Despite their high efficiency, these engines demand huge engineering effort to support new architectures or operators, frequently lagging behind the rapidly evolving LLM ecosystem. ExoMem takes a fundamentally complementary approach. As a plug-and-play middleware, it retains full compatibility with general DL frameworks while achieving the deterministic physical-memory reclamation of dedicated engines, requiring no model- or operator-specific rewrites.

Cloud-environment memory management. In datacenter environments, offloading approaches [4, 28, 47] are optimized for discrete hierarchies (*e.g.*, dedicated GPU HBM, host DRAM, and NVMe arrays) to maximize throughput and multi-GPU scaling. Consequently, their memory semantics translate poorly to mobile edge devices with unified memory architectures, where swapping between logical CPU/GPU boundaries offers no physical memory relief. ExoMem overcomes this issue by abandoning discrete tiering in favor of OS-managed, reclaimable weight storage (via address-stable dormancy). By aligning framework-level execution directly with OS-level memory dynamics, ExoMem eliminates the non-deterministic memory consumption and OOM crashes that otherwise hinder mobile edge deployment.

6 Conclusion

In this paper, we show that unpredictable memory behavior under unified memory architectures is the primary bottleneck for edge inference of large LLMs. Modern frameworks trap freed memory in private pools, preventing OS-level reclamation. ExoMem eliminates this framework-OS mismatch by introducing OS-anchored memory governance. By storing weights in OS-managed mappings that seamlessly transition into a reclaimable dormant state, ExoMem achieves deterministic memory control without requiring any model-level modifications. It ensures the stable execution of large models on resource-constrained edge devices, effectively securing the reliable memory headroom necessary for concurrent workloads on the device.

Acknowledgments

We thank all the reviewers. This work is supported by the GRF grants (CityU 11205624 and CityU 11211326) and the CRF grant (C6086-25G) from Hong Kong RGC and in part by the National Natural Science Foundation of China under Grant 62572416. The corresponding author is Zhenjiang Li.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Maximilian Abstreiter, Sasu Tarkoma, and Roberto Morabito. 2025. Sometimes Painful but Certainly Promising: Feasibility and Trade-offs of Language Model Inference at the Edge. arXiv preprint arXiv:2503.09114.

[3] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. Llm in a flash: Efficient large language model inference with limited memory. In *Proc. of ACL*.

[4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proc. of IEEE SC*.

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proc. of ACM ASPLOS*.

[6] Apple Machine Learning Research. 2024. Introducing Apple's On-Device and Server Foundation Models. https://machinelearning.apple.com/research/introducing-apple-foundation-models. Accessed: 2026-08-02.

[7] Saleh Ashkboos, Maximilian L Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. 2024. Slicegpt: Compress large language models by deleting rows and columns. In *Proc. of ICLR*.

[8] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. 2024. Quarot: Outlier-free 4-bit inference in rotated llms. In *Proc. of NeurIPS*.

[9] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101. doi:10.1147/sj.52.0078

[10] Gordon Owusu Boateng, Hani Sami, Ahmed Alagha, Hanae Elmekki, Ahmad Hammoud, Rabeb Mizouni, Azzam Mourad, Hadi Otrok, Jamal Bentahar, Sami Muhaidat, et al. 2026. A survey on large language models for communication, network, and service management: Application insights, challenges, and future directions. *IEEE Communications Surveys & Tutorials* 28 (2026), 527–566. doi:10.1109/COMST.2025.3564333

[11] Jiani Cao, Lixiang Han, Kun Wang, Zhen Xiao, and Zhenjiang Li. 2026. Systematic Pruning and Acceleration for TinyML on Extremely Weak IoT Devices. *IEEE Transactions on Mobile Computing* (2026).

[12] Jiani Cao, Kun Wang, Yang Liu, and Zhenjiang Li. 2026. Neuropath: Practically adopting motor imagery decoding through eeg signals. In *Proc. of ACM SenSys*.

[13] Gong Cheng, Junwei Han, and Xiaoqiang Lu. 2017. Remote sensing image scene classification: Benchmark and state of the art. *Proc. IEEE* 105, 10 (2017), 1865–1883. doi:10.1109/JPROC.2017.2675998

[14] Bradley Denby and Brandon Lucia. 2020. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Proc. of ACM ASPLOS*.

[15] Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *Proc. of ICML*.

[16] Georgi Gerganov. 2023. *llama.cpp: LLM Inference in C/C++*. https://github.com/ggerganov/llama.cpp

[17] Gianluca Giuffrida, Lorenzo Diana, Francesco De Gioia, Gionata Benelli, Gabriele Meoni, Massimiliano Donati, and Luca Fanucci. 2020. CloudScout: A deep neural network for on-board cloud detection on hyperspectral images. *Remote Sensing* 12, 14 (2020), 2205. doi:10.3390/rs12142205

[18] Changyi He, Yifu Ding, Jinyang Guo, Ruihao Gong, Haotong Qin, and Xianglong Liu. 2025. DA-KD: difficulty-aware knowledge distillation for efficient large language models. In *Proc. of ICML*.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*.

[20] Kai Huang, Xiangyu Yin, Heng Huang, and Wei Gao. 2025. Modality plug-and-play: Runtime modality adaptation in LLM-driven autonomous mobile systems. In *Proc. of ACM MobiCom*.

[21] Zipeng Ji, Guanghui Zhu, Chunfeng Yuan, and Yihua Huang. 2025. Rz-nas: Enhancing llm-guided neural architecture search via reflective zero-cost strategy. In *Proc. of ICML*.

[22] Fucheng Jia, Zewen Wu, Shiqi Jiang, Huiqiang Jiang, Qianxi Zhang, Yuqing Yang, Yunxin Liu, Ju Ren, Deyu Zhang, and Ting Cao. 2025. Scaling up on-device llms via active-weight swapping between dram and flash. arXiv preprint arXiv:2504.08378.

[23] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lyu, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In *Proc. of MLSys*.

[24] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. 2023. Squeezellm: Dense-and-sparse quantization. arXiv preprint arXiv:2306.07629.

[25] Benjamin Kubwimana and Qijing Huang. 2025. EdgeReasoning: Characterizing Reasoning LLM Deployment on Edge GPUs. In *Proc. of IEEE IISWC*.

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proc. of ACM SOSP*.

[27] Stefanos Laskaridis, Kleomenis Katevas, Lorenzo Minto, and Hamed Haddadi. 2024. Melting point: Mobile evaluation of language transformers. In *Proc. of ACM MobiCom*.

[28] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *Proc. of ACM OSDI*.

[29] Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, and Yunxin Liu. 2024. Flexnn: Efficient and adaptive dnn inference on memory-constrained edge devices. In *Proc. of ACM MobiCom*.

[30] Xiang Li, Zhenyan Lu, Dongqi Cai, Xiao Ma, and Mengwei Xu. 2024. Large language models on mobile devices: Measurements, analysis, and insights. In *Proc. of ACM MobiSys Workshop*.

[31] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song

Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. In *Proc. of MLSys*.

[32] Kevin Lin, Charlie Snell, Yu Wang, et al. 2025. Sleep-time Compute: Beyond Inference Scaling at Test-time. arXiv preprint arXiv:2504.13171.

[33] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. In *Proc. of NeurIPS*.

[34] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of FAST*.

[35] Benjamin Minixhofer, Ivan Vulić, and Edoardo Maria Ponti. 2025. Universal cross-tokenizer distillation via approximate likelihood matching. arXiv preprint arXiv:2503.20083.

[36] Mergen Nachin, Digant Desai, Stephen Jia, Chen Lai, Mengwei Liu, Jacob Szwejbka, Raziel Alvarez, RJ Ascani, Dave Bort, Manuel Candales, Andrew Caples, Yanan Cao, Zhengxu Chen, Soumith Chintala, Gregory Comer, Tanvir Islam, Songhao Jia, Tarun Karuturi, Jack Khuu, Abhinay Kukkadapu, Tugsbayasgalan Manlaibaatar, Andrew Or, Kimish Patel, Siddartha Pothapragada, Lucy Qiu, Supriya Rao, Orion Reblitz-Richardson, Max Ren, Scott Roy, Anthony Shoumikhin, Scott Wolchok, Guang Yang, Angela Yi, Martin Yuan, Hansong Zhang, Jack Zhang, Jerry Zhang, Shunting Zhang, and Cagatay Bilgin. 2026. ExecuTorch: A Unified PyTorch Solution to Run ML Models On-Device. In *Proc. of MLSys*.

[37] NVIDIA. 2023. TensorRT-LLM: Optimized Inference for Large Language Models. https://github.com/NVIDIA/TensorRT-LLM.

[38] NVIDIA Corporation 2023. *Tegrastats Utility - Jetson Linux Developer Guide (Release 36.2)*. NVIDIA Corporation. https://docs.nvidia.com/jetson/archives/r36.2/DeveloperGuide/AT/JetsonLinuxDevelopmentTools/TegrastatsUtility.html

[39] NVIDIA Corporation. 2025. *CUDA C++ Programming Guide*. NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Version 12.x.

[40] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of ACM SIGMOD*.

[41] Leo Pauly, Wassim Rharbaoui, Carl Shneider, Arunkumar Rathinam, Vincent Gaudilliere, and Djamila Aouada. 2023. A survey on deep learning-based monocular spacecraft pose estimation: Current state, limitations and prospects. *Acta Astronautica* 212 (2023), 339–360. doi:10.1016/j.actaastro.2023.08.001

[42] Hao Peng, Xin Lv, Yushi Bai, Zijun Yao, Jiajie Zhang, Lei Hou, and Juanzi Li. 2025. Pre-training distillation for large language models: A design space exploration. In *Proc. of ACL*.

[43] Guanqiao Qu, Qiyuan Chen, Wei Wei, Zheng Lin, Xianhao Chen, and Kaibin Huang. 2025. Mobile edge intelligence for large language models: A contemporary survey. *IEEE Communications Surveys & Tutorials* 27, 6 (2025), 3820–3860. doi:10.1109/COMST.2025.3527641

[44] Victor Rodriguez-Fernandez, Alejandro Carrasco, Jason Cheng, Eli Scharf, Peng Mun Siew, and Richard Linares. 2024. Language models are spacecraft operators. arXiv preprint arXiv:2404.00413.

[45] Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. 2023. Omniquant: Omnidirectionally calibrated quantization for large language models. arXiv preprint arXiv:2308.13137.

[46] Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. Autoiot: Llm-driven automated natural language programming for aiot applications. In *Proc. of ACM MobiCom*.

[47] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *Proc. of ICML*.

[48] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proc. of ACM SOSP*.

[49] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. 2023. A simple and effective pruning approach for large language models. arXiv preprint arXiv:2306.11695.

[50] Fuwen Tan, Royson Lee, Łukasz Dudziak, Shell Xu Hu, Sourav Bhattacharya, Timothy Hospedales, Georgios Tzimiropoulos, and Brais Martinez. 2024. Mobilequant: Mobile-friendly quantization for on-device language models. In *Proc. of EMNLP*.

[51] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/

[52] David R Thompson, Alphan Altinok, Ben Bornstein, Steve A Chien, Joshua Doubleday, John Bellardo, and Kiri L Wagstaff. 2015. Onboard machine learning classification of images by a cubesat in Earth orbit. *AI Matters* 1, 4 (2015), 38–40. doi:10.1145/2757001.2757010

[53] Kun Wang, Jiani Cao, Zimu Zhou, and Zhenjiang Li. 2024. Swapnet: Efficient swapping for dnn inference on edge ai devices beyond the memory budget. *IEEE Transactions on Mobile Computing* 23, 9 (2024), 8935–8950. doi:10.1109/TMC.2024.3355764

[54] Zheng Wang, Zhongyang Li, Zeren Jiang, et al. 2024. Crafting Personalized Agents through Retrieval-Augmented Generation on Editable Memory Graphs. arXiv preprint arXiv:2409.19401.

[55] Zijie J. Wang and Duen Horng Chau. 2024. MeMemo: On-device Retrieval Augmentation for Private and Personalized Text Generation. In *Proc. of ACM SIGIR*.

[56] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *Proc. of ACM MobiCom*.

[57] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient streaming language models with attention sinks. In *Proc. of ICLR*.

[58] Bufang Yang, Lixing He, Neiwen Ling, Zhenyu Yan, Guoliang Xing, Xian Shuai, Xiaozhe Ren, and Xin Jiang. 2023. Edgefm: Leveraging foundation model for open-set learning on the edge. In *Proc. of ACM SenSys*.

[59] Ziqi Yin, Mingxin Zhang, and Daisuke Kawahara. 2024. Harmony: A Human-Aware, Responsive, Modular Assistant with a Locally Deployed Large Language Model. arXiv preprint arXiv:2410.14252.

[60] Sixing Yu, Juan Pablo Muñoz, and Ali Jannesari. 2024. Federated foundation models: Privacy-preserving and collaborative learning for large models. In *Proc. of LREC-COLING*.

[61] Ruichen Zhang, Hongyang Du, Yinqiu Liu, Dusit Niyato, Jiawen Kang, Zehui Xiong, Abbas Jamalipour, and Dong In Kim. 2024. Generative AI agents with large language model for satellite networks via a mixture of experts transmission. *IEEE Journal on Selected Areas in Communications* 42, 12 (2024), 3581–3596. doi:10.1109/JSAC.2024.3459037

[62] Wei Zhang, Miaoxin Cai, Tong Zhang, Yin Zhuang, and Xuerui Mao. 2024. EarthGPT: A universal multimodal large language model for multisensor image comprehension in remote sensing domain. *IEEE Transactions on Geoscience and Remote Sensing* 62 (2024), 1–20. doi:10.1109/TGRS.2024.3409624

[63] Hui Zhao, Min Meng, Xiuxian Li, Jia Xu, Li Li, and Stephane Galland. 2024. A survey of autonomous driving frameworks and simulators. *Advanced Engineering Informatics* 62 (2024), 102850. doi:10.1016/j.aei.2024.102850

[64] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A Survey of Large Language Models. arXiv preprint arXiv:2303.18223.

[65] Yishuo Zhao, Xurui Song, Xinger Huang, Shujie Tian, and Yuanfeng Zhou. 2025. Ev-gazelock: A user authentication system based on micro eye movement with event cameras. *ACM Transactions on Sensor*

Networks (2025).

[66] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. In *Proc. of NeurIPS*.

[67] Yue Zheng, Yuhao Chen, Bin Qian, Xiufang Shi, Yuanchao Shu, and Jiming Chen. 2025. A review on edge large language models: Design, execution, and applications. *Comput. Surveys* 57, 8 (2025), 1–35. doi:10.1145/3719664

[68] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proc. of ACL*.

[69] Xun Zhou, Xingyu Wu, Liang Feng, Zhichao Lu, and Kay Chen Tan. 2025. Design principle transfer in neural architecture search via large language models. In *Proc. of AAAI*.