INDEXSERVE: INDEXER AS A SERVICE FOR EFFICIENT SPARSE LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Sparse attention reduces long-context decoding costs, but the decoder-resident index still grows with context length, limiting batch sizes even after key–value (KV) cache offloading. We present INDEXSERVE, a shared indexer service that moves index storage and token selection out of decoders while preserving the model's selection rule. The service retains historical indexes and exchanges only current-step inputs and selected token IDs during decoding. A graph-compatible runtime and direct device-to-device transfers over a superpod fabric support per-layer remote selection, while PagedIndex and length-aware batching pool index memory and selection work across decoders. INDEXSERVE is independent of KV offloading but complements it to enable larger decode batches. Evaluated with KV offloading on DeepSeek-V3.2 and GLM-5.1, INDEXSERVE achieves 2.46–$8.32\times$ the peak generation throughput of ECHO, the strongest evaluated baseline, after accounting for service accelerators, at the cost of higher inter-token latency.

1 INTRODUCTION

Large language models (LLMs) increasingly handle workloads such as long-document understanding, large code repositories, and agentic tasks, with contexts reaching hundreds of thousands or millions of tokens (Bai et al., 2025; GLM-5-Team et al., 2026). Standard attention reads the key–value (KV) states of all previous tokens and computes attention over them, so both memory traffic and computation grow with context length. Sparse attention reduces this cost by letting each query attend to only a small subset of past tokens. Earlier methods typically introduce sparsity at inference time without changing model training (Tang et al., 2024; Lee et al., 2024; Liu et al., 2024). Recent models instead learn selection during training. DeepSeek Sparse Attention (DSA), adopted by recent large models from DeepSeek, GLM, *etc.* (DeepSeek-AI et al., 2025; GLM-5-Team et al., 2026; DeepSeek-AI et al., 2026), uses a lightning indexer to score all past tokens and select only the top-k tokens, with $k = 2048$. As the indexer is learned as part of the model, serving systems must preserve its selection rule.

The lower attention cost of sparse attention cannot fully translate this algorithmic saving into serving system's throughput gains. A DSA model still keeps the full KV cache and additionally maintains a *K index* (referred to as the *index* below), which stores one compact index key for each past token and is scanned to select the top-k tokens (DeepSeek-AI et al., 2025). Both states grow linearly with context length. As any past token may be selected at a later step, its KV and index must remain in the accelerator's high-bandwidth memory (HBM), which limits the number of requests that can run concurrently and thus caps the batch size the accelerator can reach, avoiding full use of its compute resources. This problem is especially severe during decoding: each request produces only one token per step, so efficient execution relies on batching many requests to amortize weight reads and kernel overhead (Kwon et al., 2023; Patel et al., 2024; Zhong et al., 2024; Agrawal et al., 2024). Long-context state therefore creates a *capacity wall* for the decoding stage in LLM serving: HBM limits the batch size before the sparse operators can reach high utilization.

To relieve this capacity pressure, serving systems such as SGLang and vLLM move KV out of HBM. The state-of-the-art approach moves the full KV cache from HBM to local host DRAM, referred to as *KV offloading* (Xie et al., 2026; vLLM Ascend, 2026). Each request keeps only a fixed-size *sparse-KV buffer* in HBM for selected entries, fetching misses from DRAM on demand (Figure 2).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

By removing the full KV cache from HBM, KV offloading allows more requests to run concurrently and increases the output throughput.

However, the other context-growing state in sparse attention, the index, must still remain fully in HBM because the indexer scans the complete history to select the top-k tokens. Our measurements show that this index state can still make sparse-attention serving concurrency-bound and limit throughput. With DeepSeek-V3.2 at context lengths above 120K tokens, the current state-of-the-art KV offloading implementation vLLM Ascend (2026) admits only two concurrent requests per data-parallel (DP) replica, despite a configured limit of 16 max requests (Sec. 3.1). We characterize three root causes that make the index remain the HBM capacity wall. (Section 3.2). First, the index grows with context, while the sparse-KV buffer has a fixed size. At long contexts, the index takes most of the per-request HBM left after offloading. Second, current engines split HBM between the sparse-KV buffer and the index with a static partition set before requests arrive. No single split fits all context lengths, so part of HBM stays unused. Third, tensor parallelism (TP) cannot shard the index. All attention heads share one index key per token, so every TP rank stores a full copy and repeats the scan.

We present INDEXSERVE, a shared indexer service for sparse-attention serving. INDEXSERVE decouples the index state and scan/select compute from the decode stage and places them in a shared pool of accelerators (Figure 2b). This separation is practical because scan/select needs no model weights: it uses only the decoder-produced index inputs and the stored index, so the historical index can remain in the service itself without model parameters. The decode instances (referred to as *Decoders*) in Prefill-Decode disaggregation Zhong et al. (2024) keep the model weights and the remaining decode operators, while INDEXSERVE stores the index and performs token selection. INDEXSERVE follows the model's original scoring rule, numerical format, and top-k rule, so it introduces no approximation to token selection. INDEXSERVE is independent of KV offloading: KV offloading removes the KV cache from decoder HBM, while INDEXSERVE removes the index. We build and evaluate INDEXSERVE on top of KV offloading as it is the state-of-the-art for sparse-attention serving, and together the two remove all of the growing states from decoder HBM, unlocking the high-throughput potential of sparse attention.

INDEXSERVE is designed to tackle two challenges. First, remote indexer sits on the critical path of every sparse-attention layer, so it runs tens of times per decoding step. Host-launched calls and CPU copies would break the kernel graph execution (NVIDIA, 2026a) and add their overhead at every layer. INDEXSERVE therefore keeps remote selection inside graph replay, which runs a recorded sequence of device operations without launching each one from the host. It registers request and response buffers at fixed addresses before capture and moves data directly between device buffers over the fast scale-up network on *Superpod* Zuo et al. (2025); NVIDIA (2026b). Second, one service must serve many decoders whose requests have different and growing context lengths. Reserving memory for each request's maximum context wastes HBM, and batching short and long scans together delays the short ones. We introduce *PagedIndex*, which grows each request's index page by page and gives decoders stable logical handles, so the service can share memory and change batch members without rebuilding graphs. Moreover, we use *Length-aware batching* groups tasks with similar index lengths, sets batch sizes from offline profiles, and limits the time spent waiting to fill a batch.

We implement INDEXSERVE on Huawei CloudMatrix384 supernod (refered to as Ascend A3) (Zuo et al., 2025) and compare it with SGLang HiSparseSGLang (2026), vLLM sparse offloadingvLLM Ascend (2026), and ECHOLiu et al. (2026) on DeepSeek-V3.2 and GLM-5.1 at long contexts. At peak throughput, INDEXSERVE reaches 2.46–8.32$\times$ the generation throughput of ECHO, the strongest sparse serving baseline. At matched throughput, INDEXSERVE adds negligible inter-token latency (ITL), while at its maximum throughput, mean ITL remains at 149 ms.

Our contributions are threefold. First, we identify the decoder-resident index as a capacity bottleneck that limits sparse-attention batching after KV offloading (Section 3). Second, we design INDEXSERVE to move index storage and selection out of decoders without changing the model's selection rule (Section 4). Third, we evaluate INDEXSERVE on two models against three systems, quantifying resource-normalized peak throughput gains and their latency cost (Section 5).

2 Background and Related Work

Sparse attention and its indexer. Sparse attention lets each query attend to a small subset of key–value (KV) entries. In DeepSeek Sparse Attention (DSA), the lightning *indexer* scores each causally visible token s for query token t with H_I query heads and one shared index key per token, then keeps the top-k token identities (DeepSeek-AI et al., 2025):

$$I_{t,s} = \sum_{j=1}^{H_I} w_{t,j}^I \, \mathrm{ReLU}\big((\mathbf{q}_{t,j}^I)^\top \mathbf{k}_s^I\big), \qquad \mathcal{S}_t = \mathrm{TopKIndices}_{s \le t}(I_{t,s}, k). \tag{1}$$

The query vectors $\mathbf{q}_{t,j}^I$ and weights $w_{t,j}^I$ are learned projections of token t; $\mathbf{k}_s^I$ is the compact index key for token s. Attention then reads only the KV entries named by $\mathcal{S}_t$, and combines values with softmax weights from scaled query–key dot products. These attention vectors are separate from the indexer vectors. The index keys of all visible tokens form the *K index*, which we call the *index* from here on. Attention touches k tokens, but the index and its per-query scan in Equation 1 still grow with the visible context length L.

Algorithm-level optimizations change what is selected. One line of work lowers sparse-attention cost by rewriting selection itself. HISA first scores pooled block representatives, then restricts token-level selection to the kept blocks, which approximates DSA selections and needs no extra training (Xu et al., 2026). IndexCache reuses selected token identities across neighboring layers to skip indexer calls (Bai et al., 2026). GLM-5.2's IndexShare moves cross-layer reuse into training, so one indexer supplies selections to a group of four layers (Z.ai, 2026).

System-level optimizations keep selection fixed. A second line keeps the deployed model's selection rule and numerics, and changes only where state lives and how it moves. SGLang HiSparse, vLLM sparse offloading, and ECHO keep the full KV cache in host DRAM and keep a fixed-size sparse-KV buffer of $h \ge k$ tokens per request on the accelerator (Xie et al., 2026; vLLM Ascend, 2026; Liu et al., 2026); Figure 2a shows this layout for vLLM sparse offloading. Cache hits reuse resident KV, and misses fetch every entry the model's selection needs. ECHO overlaps transfers with indexer computation through lossless prefetching. All three move the KV cache out of the decoder but keep the index in it.

Our scope. IndexServe moves the index of training-aware sparse models such as DSA from decoder HBM to other deployed accelerators, preserving the deployed model's scoring expression, numerical format, and top-k rule without adding selection approximations. It complements algorithm-level optimizations and can operate with or without KV offloading; our evaluation uses KV offloading.

3 Motivation: The K Index Caps Decode Batching

A DSA request stores two states that grow with context length: the KV cache and the index. KV offloading moves the KV cache to DRAM, so decoder HBM then holds a fixed-size sparse-KV buffer and a growing index per request. Section 3.1 shows that the decode batch stays small while the sparse stages could still use larger batches. Section 3.2 gives three reasons why the index sets this limit, and Section 3.3 states the opportunity. All results use the DeepSeek-V3.2 INT8 deployment of Section 5.1, with conditions given in Appendix C: the stage profile runs an INT8-quantized indexer at a 64K context, whereas the capacity bound assumes a BF16 index.

3.1 Index Capacity Limits Decode Batching

Sparse attention raises the batch needed to fill the hardware. Decoding is memory-bound at small batches, and batching spreads weight reads over more requests (Kwon et al., 2023; Patel et al., 2024; Zhong et al., 2024; Agrawal et al., 2024). Sparse attention makes batching matter more: dense long-context attention is bandwidth-bound and saturates early, whereas sparse attention reads only k tokens per query, so small batches leave it bound by launch and latency costs. Figure 1a measures this gap by profiling each stage alone over batches 4–128: the INT8-quantized indexer and sparse

attention at a 64K context, and dense attention at 32K and 64K. At batch 8, dense attention reaches 89–95% of its maximum throughput, versus only 46%, 31%, and 12% for the indexer, sparse attention, and MoE. Larger batches improve all three; the indexer levels off near batch 48, while sparse attention and MoE continue improving.

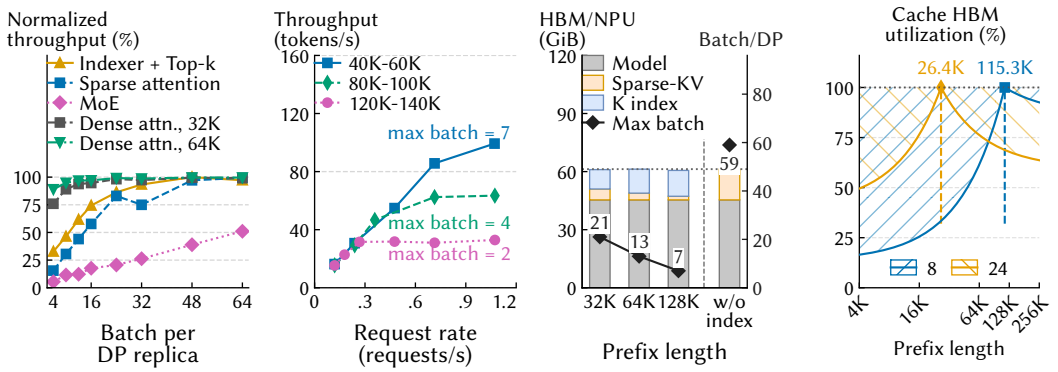

Figure 1: Batching limits in DeepSeek-V3.2 INT8. (a) Stage throughput normalized to each stage's maximum. (b) vLLM sparse offloading on Workloads A–C; labels show maximum active batch per DP replica. (c) Analytical HBM capacity with a BF16 index and per-length optimal sparse-KV/index split; "w/o index" removes the decoder-resident index. (d) Analytical static-partition HBM utilization at maximum concurrency for `max-num-seqs` of 8 and 24; hatching marks unused HBM and dashed lines mark full-utilization prefixes.

The measured batch stops far below the configured limit. Figure 1b serves the three workloads of Section 5.1 with vLLM sparse offloading. Throughput flattens early: as context length grows, the maximum active batch per DP replica reaches only 7, 4, and 2, against a configured limit of 16. This panel counts admitted requests in end-to-end serving at 40K–140K contexts. Read with panel (a), the batch stays small while the sparse stages would still gain throughput from more requests.

Removing the index would raise the batch by 2.8–8.4×. Figure 1c bounds the admitted batch analytically for a BF16 index. After model weights, each NPU keeps about 15.9 GiB of HBM for the sparse-KV buffer and K index. Even with this split tuned for each prefix length, the bound falls from 21 to 13 to 7 requests between 32K and 128K, because each request's index grows while its sparse-KV buffer stays fixed. Without the index, the same budget admits 59 requests at any prefix length, 2.8–8.4× more. This is an idealized bound, not a serving run; it shows the gap this paper targets.

3.2 Why the Index Sets the Limit

The index grows with context; the sparse-KV buffer does not. After KV offloading, the index is the only per-request state in decoder HBM that still grows. In BF16, each past token of DeepSeek-V3.2 adds 1152 bytes of multi-head latent attention (MLA) KV and 256 bytes of index (DeepSeek-AI, 2025), whereas a 4096-token sparse-KV buffer costs a fixed 0.268 GiB per request across 61 layers. Table 1 shows the index passing this fixed buffer as contexts grow. At 128K, the BF16 index is 87.7% of the per-request HBM left after KV offloading. Even in INT8 with an FP16 scale per key, it is 3.61× the size of the sparse-KV buffer.

Prefix	K index	DRAM full KV	Sparse-KV buffer
32K	0.477 GiB	2.145 GiB (22.2%)	0.268 GiB (1.78×)
64K	0.953 GiB	4.289 GiB (22.2%)	0.268 GiB (3.56×)
128K	1.906 GiB	8.578 GiB (22.2%)	0.268 GiB (7.11×)

Table 1: Per-request payload for DeepSeek-V3.2 across 61 layers (BF16, 4K-token sparse-KV buffer). Parentheses give index/full-KV and index/sparse-buffer ratios.

A static partition leaves HBM unused. Current engines also cannot pick the split per length: the two pools share one HBM budget under a partition fixed before any request arrives. SGLang HiSparse sets index capacity as a configured multiple of sparse-KV capacity (SGLang, 2026); vLLM sparse offloading reserves sparse-KV buffers for the configured concurrency and gives the rest of HBM to the index. Figure 1d shows what follows. At short prefixes, part of the index pool sits idle; at long prefixes, the index pool fills first and reserved sparse-KV slots go unused. A static partition therefore wastes HBM on top of the index footprint itself; we do not measure the two effects separately.

TP cannot shard the index. In DeepSeek-V3.2, MLA KV and the index each store one vector or key shared across query heads, so neither can be sharded by KV head under tensor parallelism (TP). Decode context parallelism (DCP) shards MLA KV along the sequence dimension (vLLM, 2026), but the engines we evaluate do not shard the index. Every TP rank therefore stores the same index keys and repeats scan/select, which duplicates both HBM and work. This replication makes the capacity wall worse but does not cause it; Section 5.4 reports a gain at TP1, where no replication exists.

3.3 Insight and Opportunity

Three properties make moving the index out of decoder HBM practical. **Scan/select needs no model weights.** Unlike projections, scan/select needs only the current token's index query, new index key, and scoring coefficients, plus stored keys (Equation 1), allowing a weight-free service with no model parallelism. Keeping historical keys there makes each exchange independent of context length: about 8 KiB per layer for fixed H_I, d_I, and k (Appendix B). **Superpod fabrics carry per-layer exchanges.** Each indexer-bearing layer communicates at every decode step, requiring a path without host mediation. Ascend A3 provides direct cross-server NPU memory access over its scale-up fabric without PCIe or RNIC hops (Appendix A). Other scale-up fabrics expose comparable device-to-device access (Zuo et al., 2025; Xiao et al., 2026; NVIDIA, 2026b); software synchronization and scanning still add to the round trip. **Shared batching overcomes per-decoder limits.** Scanning a long index does little work per row and is launch- and latency-bound at small batches: throughput grows 3.0× from batch 4 to 64 (Figure 1a). Pooling compatible calls across decoders enables larger batches despite each decoder's low admission limit (Section 3.1), while pooling index memory lets decoder and service capacity be sized independently.

4 Design

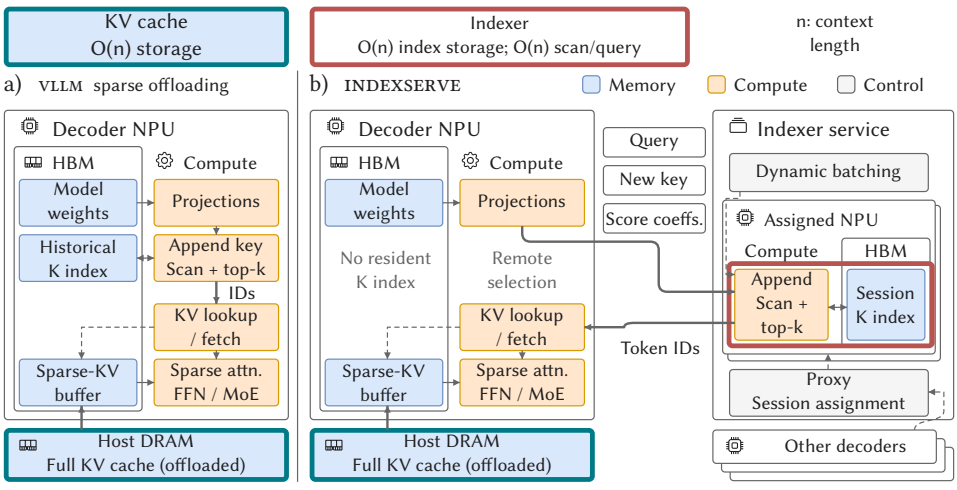

Figure 2: KV and index offloading. Both systems offload full KV to host DRAM. (a) vLLM retains the index and scan/select on the decoder; (b) IndexServe moves them to service NPUs. Teal/red outlines mark KV/index offloading; dashed arrows denote control dependencies.

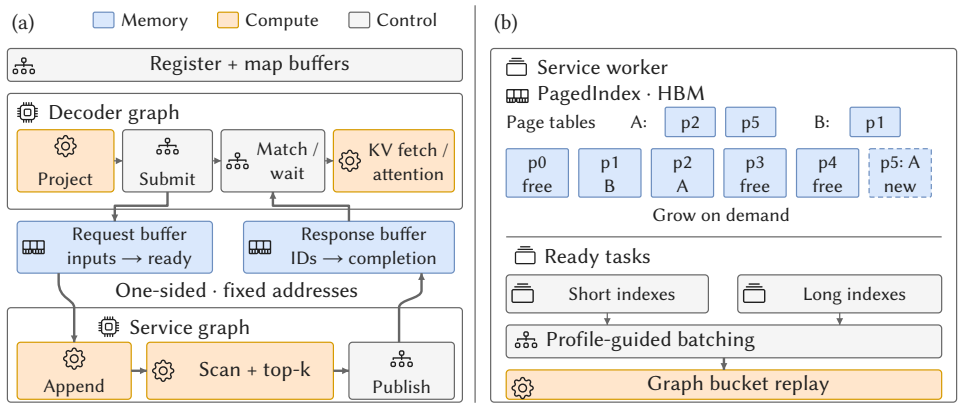

Figure 3: Indexer Service mechanisms. (a) Graph replay with fixed communication buffers and device-side synchronization. (b) PagedIndex with on-demand allocation and length-aware, profile-guided batching.

4.1 INDEXER AS A SERVICE

INDEXSERVE moves the index and the scan that reads it into a shared accelerator pool (Section 3.3, Figure 2b), which frees decoder HBM for larger batches. Decoders keep model weights, projections, sparse attention, FFN/MoE, and the KV cache. Our implementation uses KV offloading, so a decoder holds a sparse-KV buffer backed by the full KV cache in DRAM; without offloading, the decoder keeps the full KV cache in HBM and the service interface stays the same. At each indexer-bearing layer, the decoder sends the projected query, the new index key, and the scoring coefficients. The service appends the key, scans the visible history, and returns top-k token IDs. The decoder then fetches missing KV and runs attention.

Two properties of this boundary matter. First, the service computes the same scoring expression, numerical format, and top-k rule as the colocated engine, so the design adds no selection approximation. Second, in steady-state decode only current-step values and selected IDs cross the boundary, so the payload per call is fixed for given model dimensions and k; session setup sends the full prefix index once (Appendix F).

A proxy assigns each session to an accelerator by free index capacity and scan load, and its index stays there during decoding. Each task covers one session's current token at one layer, and compatible tasks from different decoders share a service batch, so decoder and service batch sizes can differ.

4.2 GRAPH REPLAY AND REMOTE COMMUNICATION

Remote selection sits on the critical path of every indexer-bearing layer, so host-issued launches and CPU staging would repeat their overhead tens of times per token. INDEXSERVE therefore keeps remote selection inside graph replay, which runs a recorded sequence of operations without launching each one from the host (Figure 3a). Before capture, the runtime registers request and response buffers and maps them for remote device access; their addresses stay fixed across replays while their contents change with each call.

After projection, device operations send the inputs over the superpod fabric and wait for the matching result, moving tensor data directly between device buffers. The service also replays captured graphs for its supported batch sizes. Because each captured graph reads row descriptors from fixed buffers, the scheduler changes which sessions occupy which rows by rewriting descriptors, not by rebuilding the graph. On both paths, data must be visible before the sender signals readiness or completion.

4.3 INDEX MEMORY AND SCHEDULING

A shared service holds indexes for sessions whose final lengths are unknown. Reserving each session's maximum context would waste the pooled memory, and handles tied to one engine's memory layout would make state shared by several engines harder to manage. *PagedIndex* therefore grows each

session's index as needed (Figure 3b). It divides service HBM into fixed-size pages and maps logical token positions to those pages. Appending a key fills the last page or allocates another, without moving earlier keys. Decoders use stable logical handles, and the service tracks page locations and frees pages after their last use.

Scheduling faces the opposite problem. Larger batches raise scan throughput, but short-index tasks then share batch completion with longer scans, which can delay their results, and every request waits while the batch is assembled. Each accelerator therefore groups compatible tasks with similar index lengths. Offline profiles set the batch size for each length group, balancing throughput and execution time. When batch-formation waiting reaches its limit, the scheduler dispatches the available tasks without filling the target batch. Short-index queues get more dispatch opportunities, while recurring long-index opportunities and oldest-first selection within each queue keep long tasks moving. Admission holds back new sessions when memory or scan capacity is exhausted. These limits bound the waiting that batch formation adds, not a task's total queueing time once offered load exceeds service capacity. Appendix F gives setup, graph, memory, and scheduling details.

5 EVALUATION

5.1 SETUP

We compare INDEXSERVE with vLLM sparse offloading (vLLM Ascend, 2026), SGLang HiSparse (Xie et al., 2026), and ECHO (Liu et al., 2026) on Ascend A3, using DeepSeek-V3.2 INT8 and GLM-5.1 INT4. All baselines run on Ascend backends, including our reimplementation of ECHO's early KV prefetching and indexer-kernel fusion in SGLang. The main comparison uses 16 decoder NPUs (TP8×DP2). Workloads A–C replay the same InfiniteBench (Zhang et al., 2024) requests with 40K–60K, 80K–100K, and 120K–140K contexts and a 128-token output cap. All systems keep full KV in DRAM, and we measure decoding only: each request starts generation once its precomputed KV cache and index are loaded. We report mean ITL and normalize INDEXSERVE throughput by $16/(16+n)$, where n is the number of service NPUs used in that run: 1, 2, and 3 for DeepSeek Workloads A–C, and 3, 5, and 7 for GLM. Idle NPUs on the dedicated service server are left out of this count. The scalability study in Section 5.4 is a separate experiment that counts whole servers. Appendix C gives setup and replay details.

5.2 THROUGHPUT

Each ratio below compares two systems at their own peak throughput, reached at different request rates and latencies, under the accounting of Section 5.1. Freeing index HBM lets more requests share decoder weight reads and kernel launches (Section 3.1). In Figure 4, DeepSeek's maximum batch stays at 37 as contexts grow, while baseline batches fall from 7 to 2. This batch gap matches its gain over ECHO growing from 3.82 to 8.32×. GLM's smaller weights leave the baselines more HBM, and its gain is 2.46–2.88×. The benefit is therefore largest when index memory limits the batch. ECHO's KV prefetching cuts transfer delay but keeps this memory limit. Appendix D tests other memory splits.

5.3 LATENCY

Remote selection adds transfers, page lookup, and service waiting to the critical path. At light load, the indexer takes 16.8 ms versus 9–10 ms in the baselines; faster attention pays back part of this cost (Figure 5b). These stage times are summed across the model's layers per decode step, not the cost of a single remote call. ECHO stays faster because it also exposes less KV-fetch time. Longer contexts need more scan work: at fixed batch 12, per-step scan/select summed over layers grows from 6.0 to 10.6 ms between 64K and 128K (Figure 5c).

The load sweep shows how throughput trades against latency. INDEXSERVE keeps mean ITL at 75–84 ms over 0.2–0.6 requests/s, but reaches 149 ms at peak throughput (Figure 5a). Larger batches and longer service waiting can each lengthen a decode step; the light-load breakdown in panel (b) does not say which of the two causes the increase at peak. Latency-sensitive serving should therefore run below saturation, using the batch targets and admission limits of Section 4.3 (Appendix C.2).

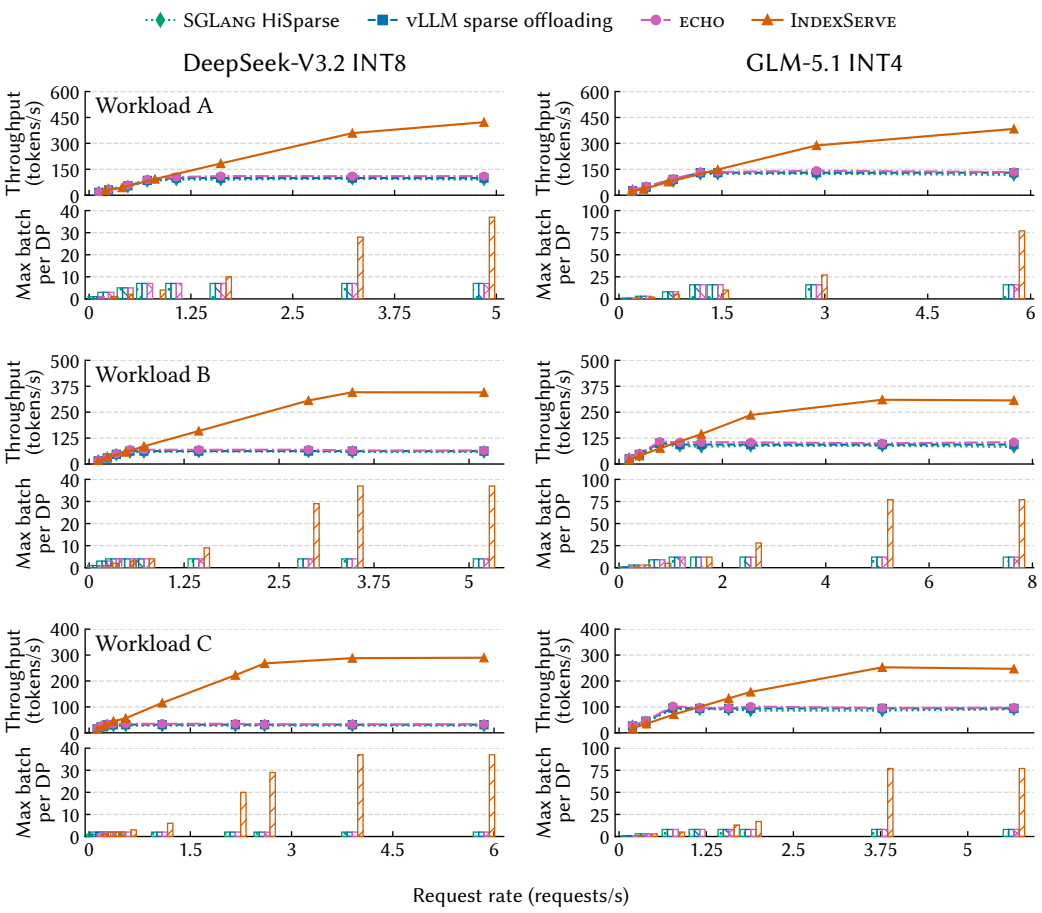

Figure 4: Generation throughput (upper lines) and maximum batch per DP replica (lower bars) for Workloads A–C, TP8×DP2. IndexServe throughput uses the normalization in Section 5.1. Each pair shares a linear request-rate axis; bars have small system-specific offsets around measured rates.

5.4 In-Depth Analysis

Ablation. We add the mechanisms of Section 4 one at a time, in the order we built IndexServe, and measure the peak throughput of each configuration (Figure 5f). Moving the index to the service alone is slower than the baseline, 45.2 versus 62.4 tokens/s. It admits a much larger batch, but host-issued calls and per-layer transfers make each forward pass far slower. Graph replay and memory-semantic communication (Section 4.2) cut that per-layer cost, which lowers latency and raises throughput to 109.9 and 207.5 tokens/s. PagedIndex then removes the service-side static index reservation that had capped the batch; it reaches 299.2 tokens/s and needs fewer service NPUs. Dynamic batching lowers each request's indexer latency (Section 4.3) and reaches 346.1 tokens/s, 5.54× the baseline.

Decoder parallelism. For GLM Workload C, the 5.16× gain at TP1 cannot come from removing TP index replication, since there is none there (Figure 5d). The service still frees decoder HBM and pools selection across decoders. The sweep rules out replication as the only source of the gain but does not separate the contributions of freed memory, cross-decoder batching, and avoided replication. Appendix E gives the other layouts and their service-memory limits.

Scalability. At 15 total servers, IndexServe gains 2.96× over vLLM while giving one server to the service. Sharing therefore pays for its resource cost. However, throughput falls 4.7% from 13 to 15 servers with the service pool fixed (Figure 5e). This decline matches saturation of the fixed service pool as more decoders feed it. Enlarging the pool or limiting admitted requests are the design options for this regime; the sweep locates the limit and does not measure their effect (Appendix C.3).

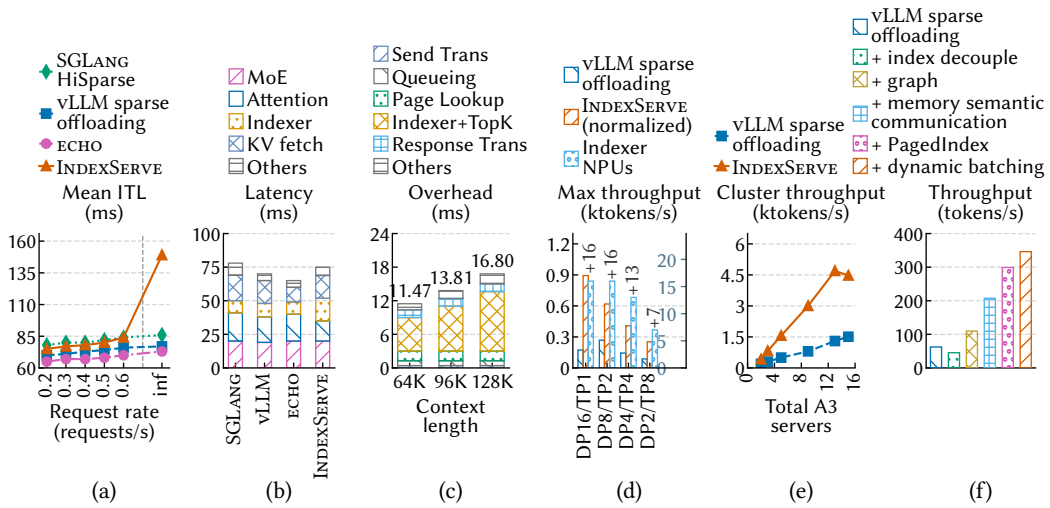

Figure 5: Latency and throughput analysis. (a)–(c) and (f): DeepSeek-V3.2 INT8; (d)–(e): GLM-5.1 INT4, Workload C. (a) Mean ITL versus request rate; `inf` marks each system's peak-throughput point. (b) Exposed ITL breakdown at 0.2 requests/s. (c) Indexer Service overhead versus context length, local batch 12. Times in (b,c) are summed across layers per decode step. (d) Peak throughput across DP×TP layouts on 16 decoder NPUs; IndexServe is resource-normalized, with service NPU counts on the right axis. (e) Unnormalized cluster throughput versus total A3 servers, TP8×DP2, with one IndexServe server dedicated to the service. (f) Cumulative peak-throughput ablation, TP8×DP2, Workload B; mechanisms accumulate left to right.

Scope of the measurements. Our evidence covers two models on one Ascend A3 scale-up fabric, with decoding measured from preloaded KV cache and index. Prefill, index loading, and end-to-end request latency fall outside it. The reported gains are resource-normalized peak throughput, not equal-latency or tail-latency results. The ablation adds the mechanisms in one order on one workload, so each step gives that mechanism's gain in that order only. IndexServe keeps the deployed scoring rule, numerical format, and top-k rule by design (Section 4.1); we do not report a measured comparison of selected token IDs or model quality.

6 Conclusion

After KV offloading, the index still limits long-context sparse-attention decode batching. We presented IndexServe, which moves index storage and scan/select to a shared service while preserving the model's selection rule. Across two models, it improves resource-normalized peak throughput at the cost of higher inter-token latency.

AI USE STATEMENT

Generative AI assisted with manuscript organization, iterative prose revision, reviewer-style feedback on drafts, literature verification, interpretation of experiment records, plotting code, and diagrams. Citation metadata was checked against primary publication pages and official documentation. The authors are responsible for verifying the technical claims, citations, experiment records, and plotted values before submission.

REPRODUCIBILITY STATEMENT

The accompanying artifact contains one directory per data figure, with CSV/JSON inputs, plotting code, provenance notes, and PDF/SVG exports. These scripts regenerate the data figures from their recorded inputs. The schematic diagrams are maintained as independent SVG sources and are not derived from measurement data. Appendix C describes the workloads and replay procedure, timing boundaries, resource accounting, and parallel configurations behind the reported results.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, 2024. URL https://www.usenix.org/conference/osdi24/presentation/agrawal.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3639–3664, 2025. doi: 10.18653/v1/2025.acl-long.183. URL https://aclanthology.org/2025.acl-long.183/.

Yushi Bai, Qian Dong, Ting Jiang, Xin Lv, Zhengxiao Du, Aohan Zeng, Jie Tang, and Juanzi Li. IndexCache: Accelerating sparse attention via cross-layer index reuse. *arXiv preprint arXiv:2603.12201*, 2026. URL https://arxiv.org/abs/2603.12201.

DeepSeek-AI. DeepSeek-V3.2 model configuration. Official model repository, 2025. URL https://huggingface.co/deepseek-ai/DeepSeek-V3.2/blob/main/config.json. Accessed September 16, 2026.

DeepSeek-AI et al. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025. URL https://arxiv.org/abs/2512.02556.

DeepSeek-AI et al. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026. URL https://arxiv.org/abs/2606.19348.

GLM-5-Team et al. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026. URL https://arxiv.org/abs/2602.15763.

Huawei. Atlas 900 A3 SuperPoD: Affinity principles. MindCluster 7.3.0 documentation, 2026a. URL https://www.hiascend.com/document/detail/en/mindcluster/730/clustersched/schedulingug/dl_affinity_0007.html. English documentation; accessed September 23, 2026.

Huawei. Ascend product models. MindStudio 7.0.0 documentation, 2026b. URL https://www.hiascend.com/document/detail/en/mindstudio/700/Referenceinformation/productdescription/hardwaredesc_0001.html. English documentation; accessed September 23, 2026.

Huawei. Ascend C API: Basic data transfer. CANN 8.5.0 documentation, n.d. URL https://www.hiascend.com/document/detail/en/canncommercial/850/API/ascendcopapi/atlasascendc_api_07_0103.html. Accessed September 21, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. URL https://arxiv.org/abs/2309.06180.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, 2024. URL https://www.usenix.org/conference/osdi24/presentation/lee.

Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. RetrievalAttention: Accelerating long-context LLM inference via vector retrieval. *arXiv preprint arXiv:2409.10516*, 2024. URL https://arxiv.org/abs/2409.10516.

Guangda Liu, Wenhao Chen, Chengwei Li, Zhenyu Ning, Jing Lin, Yiwu Yao, Quan Chen, Shixuan Sun, Jieru Zhao, and Minyi Guo. ECHO: Efficient KV cache offloading with lossless prefetching for serving native sparse attention LLMs. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 17–37. USENIX Association, 2026. URL https://www.usenix.org/conference/osdi26/presentation/liu-guangda.

NVIDIA. CUDA Programming Guide: CUDA Graphs. NVIDIA documentation, 2026a. URL https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html. Accessed September 26, 2026.

NVIDIA. Overview: NVIDIA IMEX service for NVLink networks. NVIDIA documentation, 2026b. URL https://docs.nvidia.com/multi-node-nvlink-systems/imex-guide/overview.html. Accessed September 16, 2026.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture*, pp. 118–132, 2024. doi: 10.1109/ISCA59077.2024.00019. URL https://doi.org/10.1109/ISCA59077.2024.00019.

SGLang. HiSparse: Hierarchical sparse attention. SGLang documentation, 2026. URL https://github.com/sgl-project/sglang/blob/main/docs/docs/advanced_features/hisparse_guide.mdx. Accessed September 16, 2026; documentation is distinct from the original SGLang paper.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. QUEST: Query-aware sparsity for efficient long-context LLM inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 47901–47911, 2024. URL https://proceedings.mlr.press/v235/tang24l.html.

vLLM. Context parallel deployment. vLLM documentation, 2026. URL https://docs.vllm.ai/en/latest/serving/context_parallel_deployment/. Accessed September 22, 2026.

vLLM Ascend. Layerwise and sparse KV cache offloading design. vLLM Ascend documentation, 2026. URL https://docs.vllm.ai/projects/ascend/en/latest/developer_guide/Design_Documents/layerwise_and_sparse_kv_cache_offloading.html. Accessed September 16, 2026.

Ao Xiao et al. xDeepServe: Model-as-a-service on Huawei CloudMatrix384. *arXiv preprint arXiv:2508.02520*, 2026. URL https://arxiv.org/abs/2508.02520.

Zhiqiang Xie, Zhangheng Huang, Tingwei Huang, Ziyi Xu, Ruiyang Ma, and Christos Kozyrakis. HiSparse: Scaling sparse-attention decoding with hierarchical KV cache management. *arXiv preprint arXiv:2608.07009*, 2026. URL https://arxiv.org/abs/2608.07009.

Yufei Xu, Fanxu Meng, Fan Jiang, Yuxuan Wang, Ruijie Zhou, Zhaohui Wang, Jiexi Wu, Zhixin Pan, Xiaojuan Tang, Wenjie Pei, Tongxuan Liu, Di Yin, Xing Sun, and Muhan Zhang. HISA: Efficient hierarchical indexing for fine-grained sparse attention. In *Conference on Language Modeling*, 2026. URL https://arxiv.org/abs/2603.28458v4.

Z.ai. GLM-5.2: Built for long-horizon tasks. Official model release, 2026. URL https://huggingface.co/blog/zai-org/glm-52-blog. Published June 17, 2026.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Khai Hao, Xu Han, Zhen Leng Thai, Shuo Wang, Zhiyuan Liu, and Maosong Sun. ∞Bench: Extending long context evaluation beyond 100k tokens. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15262–15277, 2024. doi: 10.18653/v1/2024.acl-long.814. URL https://aclanthology.org/2024.acl-long.814/.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, 2024. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Pengfei Zuo et al. Serving large language models on Huawei CloudMatrix384. *arXiv preprint arXiv:2506.12708*, 2025. URL https://arxiv.org/abs/2506.12708.

A ASCEND A3 SUPERPOD, A3 SERVER, AND 910C NPU

This appendix gives the platform-specific implementation of the graph and memory-semantic communication design in Section 4.2. The design needs mapped remote memory, device-initiated transfers, and explicit completion ordering. The mechanisms below meet those requirements on Ascend A3; their names and topology are not part of the service interface.

A.1 HARDWARE HIERARCHY AND MEMORY PLACEMENT

This paper uses three hardware terms at three levels. The *Ascend A3 superpod* is the complete scale-up system. It spans 12 compute racks and four communication racks and contains 48 *A3 servers*, 768 Ascend 910C NPUs, and 192 Kunpeng CPU sockets. An *A3 server* is the unit that the public CloudMatrix paper calls a compute node. It contains 16 *910C NPUs*, four CPU sockets, and seven first-level Unified Bus (UB) switches. We count servers when describing deployment scale and NPUs when describing parallelism or resource allocation. Huawei's English product documentation identifies the Atlas 900 A3 SuperPoD as the A3 cluster basic unit (Huawei, 2026b). Its English scheduling manual gives the corresponding multi-server hierarchy (Huawei, 2026a).

Figure 6 shows these boundaries. Second-level UB switches connect the 48 A3 servers without oversubscribing the scale-up fabric (Zuo et al., 2025). Here, *cross-server* means crossing A3 servers within one superpod; it does not mean crossing the separate scale-out network between superpods.

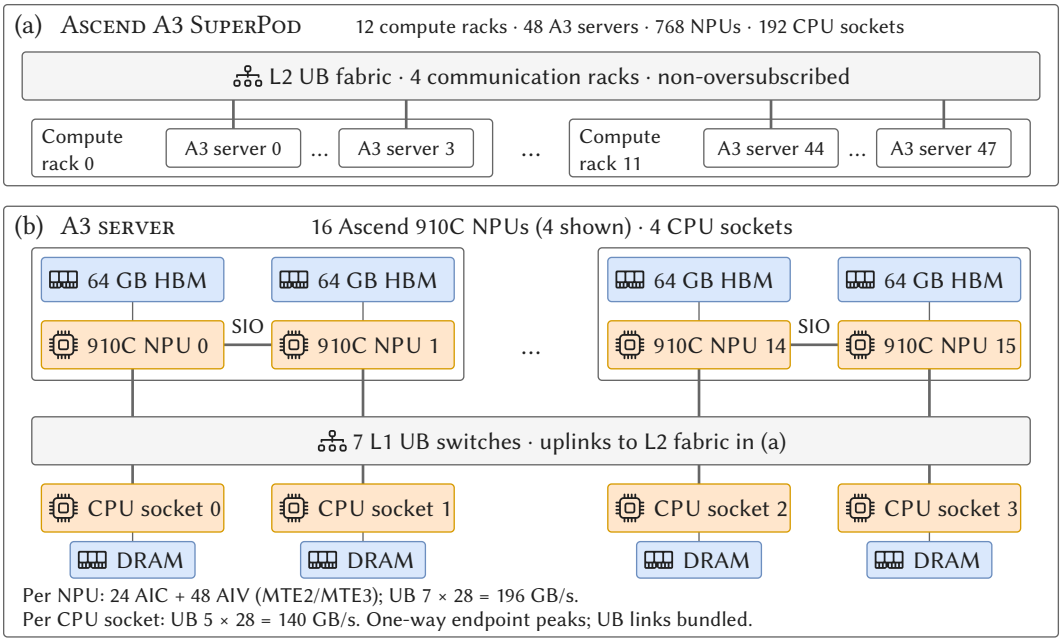

Figure 6: Ascend A3 hardware hierarchy and UB interconnect. (a) The A3 superpod contains 48 A3 servers across 12 compute racks, connected through the L2 UB fabric in four communication racks. (b) Each A3 server has seven L1 UB switches, 16 910C NPUs (four shown), and four CPU sockets. Each NPU connects to HBM, each CPU connects to DRAM, and SIO connects paired NPUs. UB links are drawn as bundles; individual switch-plane wiring is omitted. Bandwidth annotations are one-way endpoint peaks for the reference configuration.

Each 910C NPU has 64 GB of HBM, 24 AI Cube (AIC) cores, 48 AI Vector (AIV) cores, and up to 1.6 TB/s of local memory bandwidth. The four CPU sockets of an A3 server form four NUMA domains. All decoder–service traffic in our evaluation shares the NPU ingress and egress path of this fabric. Paired NPUs are connected by the SIO interconnect (Zuo et al., 2025). UB attaches both NPU memory and CPU DRAM to the scale-up fabric. Placing the indexer on another server therefore adds no separate low-bandwidth PCIe-to-RNIC path. The decoder can keep a sparse-KV buffer in HBM, back it with a DRAM-resident full KV cache, and exchange requests with a service that holds the past index in service HBM.

A.2 GLOBAL ADDRESSING AND DEVICE-DRIVEN ACCESS

UB provides a global physical address space for *fabric-attached* memory. An NPU's virtual addresses reach this space through its memory mappings and hardware address translation. The runtime establishes page-table mappings; the memory management unit translates the virtual address used by a device kernel into a fabric-attached physical address. This allows byte-addressable access to mapped remote HBM or CPU DRAM across the superpod. It does not make every host allocation remotely accessible: private host memory is not the same as memory configured for fabric access. A global physical address space also does not mean that different processes use the same virtual addresses.

INDEXSERVE uses this fabric for small, memory-semantic transfers between the decoder and the remote indexer service. At initialization, the runtime allocates communication regions, registers them with the memory backend, and exchanges the information needed to set up accessible mappings. Each participant keeps its own mapped address and the offsets of request, response, and control slots. It keeps those mappings valid until all graph references and outstanding transfers have drained. Device kernels running on compute cores then access mapped remote memory through memory-transfer operations (Huawei, n.d.). These kernels gather current-step inputs, transfer them to the service, and publish completion without a separate host-issued copy for each field. The same mechanism returns selected token IDs to the decoder. This device-driven path is especially useful for the small exchanges repeated at every model layer.

Global reachability does not give a coherent shared heap. The runtime orders payload writes before it publishes completion, checks request identity before consuming a response, and delays buffer reuse until consumption. These rules keep the graph's addresses stable while the contents change on each replay. Kernel-driven transfers also use device resources, so running more of them in parallel can interfere with model execution.

A.3 ASCEND MAPPING AND CONCURRENT TRANSFER KERNELS

The Ascend implementation uses MemFabric's shared-memory mappings to expose the communication regions to device kernels. A remote location comes from the mapping's base, the peer's runtime-provided symmetric-region stride, and its slot offset. The runtime supplies the stride because its virtual layout can reserve more space per participant than the memory that participant contributes. The kernel uses this mapping metadata to resolve a peer address, and keeps the resulting mappings across graph replays.

Ascend's compute and transfer resources make this mapped path programmable. Each 910C NPU contains the AIC and AIV cores shown in Figure 6. The AIV execution path has on-chip buffering and two memory transfer engines, MTE2 and MTE3, for moving data between global memory and the local buffer. Ascend C `DataCopy` operations express these transfers (Huawei, n.d.). With a valid remote mapping, global memory can point to a peer's fabric-attached memory. The kernel can therefore gather decoder fields and write remote slots through the memory path without staging the numerical payload in a CPU buffer.

The request kernel gives independent fields to several AIV cores so that their transfer engines can run at the same time. The prototype uses nine cores to copy nine segments to fixed, 32-byte-aligned offsets. The cores synchronize after their payload transfers; one chosen core publishes the request marker and owns the response protocol. The core-local buffer stays part of the transfer path. Memory-semantic communication therefore avoids a CPU bounce buffer and can remove an intermediate packed HBM buffer; it does not mean that data skips all buffers or moves for free.

Synchronization must match the execution resources that take part. The prototype uses an AIV-only barrier for its communication cores after their payload transfers. Control transfers are padded to the backend's supported granularity, which is 32 bytes here. The chosen publishing core then writes the ready or completion marker after the matching payload is visible. Appendix B states these ordering rules without Ascend-specific barriers and alignment, and says which copies are removed.

A.4 BANDWIDTH AND LATENCY FOR INDEXER DISAGGREGATION

Endpoint link rates and sustained memory-transfer bandwidth are different quantities. In the reference configuration, each NPU has seven 28 GB/s UB links, totaling 196 GB/s in one direction; each CPU socket has five such links, totaling 140 GB/s. These are aggregate endpoint line rates, not the bandwidth of every NPU–DRAM pair.

Table 2: Memory-access performance from the reference platform characterization. NPU bandwidth is sustained bandwidth when an NPU reads the indicated memory. Latency is the reported access latency for that path, not the decoder–service round-trip time. These measurements are distinct from endpoint line rates and from INDEXSERVE's end-to-end results.

Memory accessed over UB	NPU bandwidth (GB/s)	Latency (μs)
Other-package HBM, same server	165	0.7
HBM, different server	160	2.1
CPU DRAM, same server	112	0.8
CPU DRAM, different server	110	2.2

Low access latency matters because each layer waits for its selection result before sparse attention can proceed. Table 2 reports 0.7–2.1 μs for remote HBM access, with cross-server bandwidth close to the in-server path. This supports small exchanges over the scale-up fabric, although software synchronization and indexer execution add to the full round trip. INDEXSERVE keeps scans over the stored index keys inside the service and sends only current-step inputs and selection results during decode.

A.5 STRIPED DRAM BACKING ACROSS CPU SOCKETS

Larger decoder batches also raise the KV traffic that CPU DRAM must supply. Host–accelerator bandwidth is a known limit for KV offloading: ECHO, for example, studies recall over a 64 GB/s PCIe Gen5 connection between the host and each GPU (Liu et al., 2026). The superpod exposes multiple CPU sockets through UB, but placing a KV pool behind one socket can still concentrate its reads on that socket's egress. This matters most for the baseline's shared full-KV pool in DRAM: tensor-parallel ranks in the same data-parallel group share one DRAM-resident full KV cache (vLLM Ascend, 2026). Sharing stores one copy while several ranks can still fetch KV at the same time.

INDEXSERVE stripes the pool's physical backing across CPU sockets so KV reads can use their combined egress bandwidth. A stable logical address space maps successive stripes to different sockets; decoder block handles and captured graph pointers do not change. The stripes must be fine enough to spread a layer's working set, not just to balance total allocation across all layers. This mapping keeps one CPU socket's supply bandwidth from becoming the bottleneck at large batches. It does not remove contention at NPU ingress or elsewhere in the shared fabric.

B DSA COMPUTATION AND COMMUNICATION AT THE SERVICE BOUNDARY

This appendix gives the payload and metadata behind the fixed-buffer interface in Section 4.2. Appendix A describes the Ascend memory mapping and transfer kernels. The protocol below covers an ordinary decode call.

B.1 THE CUT BETWEEN PROJECTIONS AND SCAN/SELECT

DSA separates model-dependent projections from a scan over the stored index keys (Equation 1). At layer ℓ and token step t, the decoder produces H_I index queries $\mathbf{q}_t^I \in \mathbb{R}^{H_I \times d_I}$, one new index key $\mathbf{k}_t^I \in \mathbb{R}^{d_I}$, and scoring coefficients $\mathbf{w}_t^I \in \mathbb{R}^{H_I}$ (DeepSeek-AI et al., 2025). The decoder keeps these projections, including the model's positional processing and the chosen numerical format. The service owns the past index and the operators that consume these projected values.

For each call, the service first appends the new key and its scale to the correct sequence and layer. It then computes the model's scores over all causally visible keys, applies the same top-k rule, and

returns the selected token identities. The scan must see this update so that the current token can be selected. The full score vector stays on the service. Model weights, the decoder hidden state, and the stored index keys do not cross this boundary on each decode call. The decoder uses the returned identities to find sparse-KV-buffer hits and fetch missing KV before running the original sparse attention.

B.2 PER-REQUEST, PER-LAYER PAYLOAD

DeepSeek-V3.2 specifies $H_I = 64$, $d_I = 128$, and $k = 2048$ (DeepSeek-AI, 2025). Table 3 counts the numerical payload for the prototype's INT8-quantized indexer format: INT8 queries and index keys, FP16 scales and scoring coefficients, and INT32 selected identities. A query has one scale per head, while each new index key has one scale. This count holds for that format only; transport alignment, graph-bucket padding, and control fields add to the actual transfer.

Table 3: Numerical payload for one request at one indexer-bearing layer in the DeepSeek-V3.2 INT8-quantized indexer format. Descriptor, completion, and optional miss-plan bytes are additional; the entries do not describe a padded transport-buffer allocation.

Direction	Field	Shape and format	Bytes
Decoder → service	Index queries	64×128 INT8	8,192
	Query scales	64 FP16	128
	Scoring coefficients	64 FP16	128
	New index key	128 INT8	128
	New-key scale	1 FP16	2
	Input numerical payload		8,578
Service → decoder	Selected token IDs	2048 INT32	8,192

More generally, for element widths b_q, b_w, b_I, and b_{id} in bytes, the request and response sizes are

$$\begin{aligned} S_{\mathrm{req}} &= H_I d_I b_q + H_I b_w + d_I b_I + S_{\mathrm{scales}} + S_{\mathrm{descriptor}}, \\ S_{\mathrm{resp}} &= k b_{\mathrm{id}} + S_{\mathrm{result\ metadata}}. \end{aligned} \tag{2}$$

The INT8-indexer numerical exchange totals 16,770 bytes, approximately 16.38 KiB per layer. As a separate format example, BF16 queries alone occupy 16 KiB. With BF16 new keys and FP16 coefficients, the unquantized input payload is 16,768 bytes; adding INT32 top-2048 IDs gives 24.375 KiB. Each example keeps its own numerical configuration; changing quantization does not preserve the selections of another configuration.

A descriptor travels with the numerical payload. It identifies the model adapter, sequence and generation, layer, token step, invocation, valid length, index handle, and response slot. These fields say which resident state to update and which decoder may consume the result. Their groups appear below; integer widths and packed size depend on the backend ABI. A compact miss plan also gives destination slots and backing addresses and carries a residency version; its size depends on the number of misses.

For fixed H_I, d_I, and k, this per-call numerical exchange does not grow with context length L, although resident index size and scan work do. Setup must load all L index keys for each indexer-bearing layer. At 128K tokens, this is 16.25 MiB per layer in the INT8 key/scale format, or 32 MiB in BF16. These initial transfers are counted separately from decode calls. The selected KV takes a different path, directly from the DRAM-resident full KV cache to the sparse-KV buffer. For DeepSeek-V3.2, at $k = 2048$ and 1152 bytes per BF16 MLA KV entry, an all-miss layer needs 2.25 MiB per request on that path. Moving the indexer does not remove this traffic. Other model adapters use Equation 2 with their own head counts, dimensions, and numerical formats.

B.3 STABLE GRAPH BUFFERS AND RUNTIME ADDRESS RESOLUTION

Both sides capture computation over stable buffers and change their contents at runtime. The decoder graph keeps the addresses of its input descriptors, request slots, response slots, and completion records. A captured service bucket likewise keeps its input/output workspaces, page-directory storage,

and row descriptors. Replay changes the active rows, logical index handles, lengths, and slot offsets stored in those buffers. It does not swap a newly allocated tensor for a pointer built into the captured graph. A batch shape change picks another captured bucket, and unused rows are masked.

This interface has two separate mappings. The communication backend maps a registered region into an address that a device can reach across the fabric. Inside the service, *PagedIndex* maps each logical index position to its HBM page. The two have different lifetimes: the communication region stays pinned across replays, while the index's page-table entries grow as the sequence appends tokens. Adding an index page updates a directory entry and does not move the fixed graph buffers.

For page capacity P, logical token position i resolves to page-table entry $\lfloor i/P \rfloor$ and in-page position $i \bmod P$ for the requested sequence and layer. The row descriptor gives the logical handle and valid length; the service resolves the matching physical page at execution time. The page-table update and new key must be visible before that row's scan begins. The directory needs enough preallocated metadata capacity for the captured configuration. Its pointer cannot be replaced silently when the index outgrows that capacity. Reserving this metadata does not reserve the sequence's maximum-length index payload.

This indirection therefore lets batch membership change and indexes grow under graph replay. The application-facing handle names persistent index state; the transport slot names one exchange; the invocation identity separates successive uses of that slot. None of these should be guessed from a raw virtual address. The decoder and service may map the same registered region at different virtual addresses.

B.4 Communication metadata and publication ordering

The protocol separates routing information, per-invocation state, and completion state. Routing information binds a session to its service owner and registered communication slots. Invocation metadata gives the sequence generation, layer, token step, model/format, valid index length, and logical index handle. Completion metadata echoes the invocation identity and records status so a decoder can tell its current result from an earlier use of the same slot. On the request path, the decoder writes the current numerical inputs and descriptor before it publishes a ready marker. The service may consume them only after it sees that marker under the backend's visibility rules. It resolves the index handle, installs any needed page mapping, and appends the new key before scan/select. The current token therefore appears exactly once in the causally visible history for that call.

On the response path, the service writes selected IDs and response identity before it publishes completion. The decoder's device-side wait checks that the completion belongs to its current call before it releases the dependent KV lookup, fetch, and attention operations. Payload visibility must come before marker visibility on both paths, and seeing a marker must come before consuming the payload. The implementation uses device transfer-completion and synchronization primitives for these orderings. A shared address or a nonzero control word alone does not give them.

The slot belongs to its call until the response has been consumed. With several slots, a producer can prepare another exchange only in a free slot; it cannot overwrite a live request or an unread response. Identity checks and generation changes keep a late completion from being taken for new work. These are protocol rules, not a claim that the ordinary graph-replay experiments have covered every failure and cancellation case.

B.5 Backend operations and message preparation

The backend exposes registration, mapped address exchange, request publication, completion observation, and deregistration after quiescence. A registration record binds a buffer to its owner, access permissions, size, and lifetime. Model code sees logical handles and stable buffers; the backend turns them into reachable addresses. The A3 implementation uses the Unified Bus memory path described in Appendix A. NVIDIA IMEX offers one possible device-memory exchange mechanism on NVLink systems (NVIDIA, 2026b); it is a portability example, not a finished port or an equal coherence model.

The implementation gathers fields into a compact registered layout and fuses data preparation where the operator interface allows it. A direct-pack kernel reads the input tensors and writes their assigned

offsets in the remote slot, which removes a separate local packed-request buffer. This keeps the fixed destination layout and lets independent fields transfer at the same time. Block tables are versioned and updated when mappings change, instead of being copied in full every layer. Small-message core count is a tuning knob: too few compute cores serialize packing; too many raise interference or polling cost. We do not give one best core count without a shape-specific sweep.

We use *zero-copy* to mean avoiding CPU payload staging and needless intermediate buffer copies. The request and response still move across the fabric; kernels may use on-chip buffers, and the service may copy or gather rows into the contiguous local layout its scan operator needs. Batch assembly, alignment, and output distribution are therefore part of the communication and service costs. Registering a buffer makes direct access possible; it does not mean every operator can consume that layout without preparation.

B.6 TP OWNERSHIP AND FANOUT

An owner submits one request only after the inputs match across TP ranks. If the model shards query projections, an assembly step runs before the service call and is charged to that call. Returned IDs go to the ranks that use the same selection. Data and completion markers use separate ownership rules so consumers cannot overwrite each other's unread result. We attribute no speedup to removing copies that the baseline never had.

C EVALUATION DETAILS

This appendix gives the setup, replay procedure, latency breakdown, and the limits on how to read the results in Section 5.

C.1 SETUP AND REPLAY

Hardware and models. All servers belong to the same Ascend A3 superpod (Appendix A). Each A3 server has 16 Ascend 910C NPUs. The main comparison uses one decoder server and a separate service server. The scalability experiment uses up to 14 decoder servers with one service server. DeepSeek-V3.2 INT8 occupies 647 GiB across 61 layers; GLM-5.1 INT4 occupies 391 GiB across 78 layers. Both models use BF16 MLA KV entries; all systems keep the full KV cache in DRAM. Like ECHO (Liu et al., 2026), we separate prefill from decode. Each request starts generation after its precomputed KV cache and index have been loaded. INDEXSERVE keeps the index in service HBM.

Software versions and Ascend ports. We run SGLang v0.5.17 and vLLM v0.26.0 with their Ascend backends, using vLLM Ascend v0.26.0rc1. Our HiSparse baseline uses an Ascend port under internal development within the SGLang community. We reimplement ECHO's early KV prefetching and its fusion into the indexer kernels in SGLang's Ascend backend. All baselines support graph execution.

Baseline decode path. During decoding, each baseline offloads the full KV cache to a DRAM pool and keeps the full index in HBM. At each step, it fills an HBM-resident sparse-KV buffer from the indexer's top-k selection, reusing cached entries and fetching only misses.

Baseline settings. The main throughput comparison uses `max-num-seqs=16` for vLLM (vLLM Ascend, 2026) and `host_to_device_ratio=5` for SGLang HiSparse and ECHO (SGLang, 2026). The latter sets the ratio of index-token capacity to sparse-KV-token capacity. Appendix D varies the static capacity split. The separate decoder-parallelism sweep uses vLLM `max-num-seqs=24` (Appendix E).

Configuration for the model analysis in Section 3. That analysis uses the DeepSeek-V3.2 INT8 model on vLLM Ascend v0.26.0rc1-a3 with TP8 $\times$ DP2, EP16, ACLGraph, vLLM sparse offloading, $k = 2048$, and a 4096-token sparse-KV buffer per request per layer. For Figure 1a, we replay each stage for over 20 seconds on two TP8 replicas and time its graph with device events. Throughput is normalized to each stage's maximum throughput. Larger batches did not increase throughput. Hardware checks show that these peaks are limited by compute, bandwidth, or data movement. The

INT8-quantized indexer and sparse attention use a 64K context; dense attention uses 32K and 64K contexts with 16 heads per NPU; MoE includes routing and communication. Figure 1b samples the same three context-length groups as Workloads A–C and uses vLLM sparse offloading with `max-num-seqs=16`. It replays Poisson arrivals with the same 128-token output cap. Figure 1c excludes runtime overhead and page rounding; model weights occupy 45.4 GiB of each NPU's 61.3 GiB HBM, and the remaining 15.9 GiB is split between the sparse-KV buffer and index. Its index footprint uses 256 bytes per key, that is, a BF16 index, whereas the stage profile above runs the INT8-quantized indexer. The capacity bound is an idealized example and the stage profile is a microbenchmark, so the two show different aspects of the same deployment rather than one configuration.

Replay procedure. For each model, we fix the prompt set, context-length distribution, decoding parameters, and 128-token output cap. All four systems replay the same request arrival sequence. The throughput and latency experiments sweep the request rate from light load through saturation. The scalability experiments drive every decoder server with enough concurrent requests to form large service batches and reach its maximum-throughput operating point. Reported batch sizes count requests per DP replica; the TP8 $\times$ DP2 deployment has twice that batch in aggregate.

Resource accounting. Throughput counts completed output tokens per second. For a 16-NPU decoder and n service NPUs, we multiply INDEXSERVE throughput by $16/(16+n)$. Unused NPUs on the service server are excluded from this per-NPU accounting. The scalability comparison instead counts whole servers, each with 16 NPUs. At a total of m servers, INDEXSERVE uses $m-1$ decoder servers and one service server; vLLM uses all m for decoding. Mean ITL is the time between consecutive output tokens.

C.2 LATENCY BREAKDOWN

Figure 5(a)–(c) reports DeepSeek-V3.2 INT8 latency. At 0.2 requests/s, mean ITL is 75 ms for INDEXSERVE, 78 ms for SGLang HiSparse, 70 ms for vLLM sparse offloading, and 65 ms for ECHO. INDEXSERVE stays within 75–84 ms through 0.6 requests/s, where it matches HiSparse at 84 ms. At maximum throughput, its ITL rises to 149 ms, which is the latency cost of running at peak throughput.

Panel (b) splits exposed ITL into MoE, attention, indexer, KV fetch, and other work, counting each interval once. Each component is the time per batched decode step summed across the model's layers, not the cost of a single layer or remote call. The INDEXSERVE indexer takes 16.8 ms, versus 9–10 ms for the baselines. Its attention takes 15 ms; that saving and the other stage differences keep total ITL at 75 ms. Compared with ECHO, INDEXSERVE spends 7.8 ms more on the indexer and 6 ms more on exposed KV fetch, partly offset by 5 ms less on attention. ECHO's prefetching reduces exposed KV-fetch time to 11 ms, versus 17 ms for INDEXSERVE. This is why the capacity gain does not also give the lowest ITL.

Panel (c) fixes local batch size at 12 and varies context length. Table 4 gives its components, again in milliseconds per batched decode step summed over the 61-layer sweep. Scan/select grows with context length. The plotted breakdown holds transfer, queueing, and page lookup fixed at 4.384 ms in total at each point. These values do not show that communication or waiting stays fixed as offered load grows.

Table 4: Indexer Service time in milliseconds at local batch size 12. The shared transfer/queue/lookup column sums send transfer, queueing, page lookup, and response transfer. The 128K total matches the 16.8 ms indexer component in Figure 5(b).

Context	Indexer + top-k	Transfer/queue/lookup	Other	Total
64K	5.982	4.384	1.100	11.466
96K	8.026	4.384	1.400	13.810
128K	10.616	4.384	1.800	16.800

C.3 LOAD CONTROLS AND SCALING LIMITS

The latency sweep helps pick an operating point below peak throughput. The 75–84 ms ITL range at 0.2–0.6 requests/s and the 149 ms peak-throughput point come from different offered loads. Each baseline's peak also happens at its own load, so peak ITL is not a comparison at equal throughput. The breakdown we have is for light load; it does not say how much of the peak increase comes from queueing, larger batches, or other work.

The design has three controls: the admitted load, the service batch target, and the aggregation wait. Lower batch targets and shorter waits can cut batching delay, but may give up throughput. Admission must also keep demand within memory and scan capacity. Adding service capacity is another option, with a clear resource cost. The current experiments do not test each control separately or give a hard latency guarantee.

In the scaling sweep, INDEXSERVE uses one service server throughout. Throughput rises to 4,690.4 tokens/s at 13 total servers, then falls to 4,470.3 at 15; vLLM reaches 1,510.3 at 15. The decline matches saturation of the fixed service pool as more decoders feed it. Adding service capacity or limiting admitted requests are possible answers, but this sweep does not measure their effects. Service sizing must cover both index memory and scan load.

D SENSITIVITY TO STATIC CAPACITY PARTITIONS

Sparse-offloading runtimes split cache HBM between an index and a sparse-KV staging buffer. The split is fixed before a request arrives, even though the two pools grow differently with context length and concurrency. This appendix first shows how vLLM sparse offloading and SGLang HiSparse expose that split. It then gives the full vLLM sensitivity sweep we used to pick the main-paper configuration.

vLLM sparse offloading: per-request sparse-KV slots. vLLM configures a maximum number of concurrent sequences through `max-num-seqs`. Let this value be C. The runtime reserves a sparse-KV buffer for each of the C possible sequences and uses the remaining cache HBM for index entries. In our configuration, each admitted request can stage 4,096 sparse-KV tokens per layer. If M is the cache-HBM budget, m_s is the per-request sparse-KV-buffer footprint, and $m_i(L)$ is the index footprint of a request with context length L, a simplified capacity bound is

$$B_C(L) = \min\left(C, \frac{M - Cm_s}{m_i(L)}\right), \qquad M \geq Cm_s. \tag{3}$$

This expression leaves out allocator metadata, page rounding, graph workspaces, and other runtime allocations. A small C can cap the batch while leaving index capacity unused. A large C reserves more sparse-KV slots but leaves less HBM for long-context indexes. We evaluate $C \in \{8, 16, 24, 32\}$.

SGLang HiSparse: index-to-sparse-KV token ratio. SGLang HiSparse instead sets index-token capacity as a multiple of sparse-KV token capacity (SGLang, 2026). Let R be this `host_to_device_ratio`, T_s be the aggregate sparse-KV-token capacity, and $T_i = RT_s$ be the index-token capacity. If c_s and c_i are the corresponding bytes per token, the cache-HBM constraint gives

$$T_s = \frac{M}{c_s + Rc_i}, \qquad B_R(L) = \min\left(\frac{T_s}{h}, \frac{RT_s}{L}\right), \tag{4}$$

where h is the sparse-KV reservation per admitted request. A larger R gives more token capacity to the index and less to sparse-KV staging. So vLLM fixes the number of per-request staging slots, whereas SGLang fixes a ratio between two aggregate token capacities. Both are static, and their best settings depend on the context-length and concurrency distribution.

Methodology. Figures 7–14 compare only vLLM sparse offloading and INDEXSERVE. Both models use TP8×DP2 on 16 decoder NPUs. Each figure follows the Workload A/B/C definitions in Section 5.1. The setting in each title applies only to vLLM; INDEXSERVE has no corresponding static partition. We normalize each INDEXSERVE point by $16/(16 + n)$, where n is the service-NPU

count for that model, workload, and setting, and label it *IndexServe (normalized)*. DeepSeek-V3.2 under Workload C with `max-num-seqs=32` is capacity-unavailable.

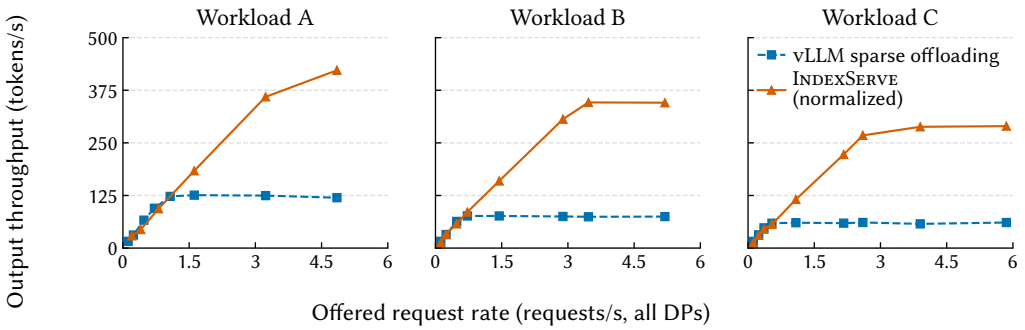

Figure 7: DeepSeek-V3.2 INT8, TP8×DP2, with vLLM `max-num-seqs=8`. Panels show Workloads A, B, and C. IndexServe is normalized by total allocated NPUs; vLLM uses the 16 decoder NPUs.

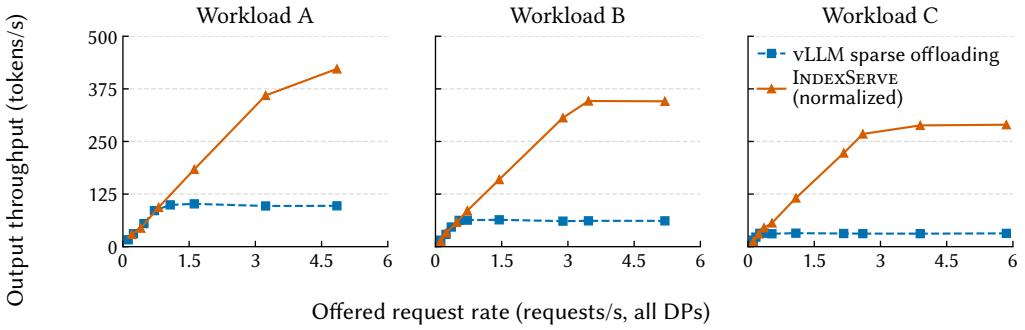

Figure 8: DeepSeek-V3.2 INT8, TP8×DP2, with vLLM `max-num-seqs=16`. Panels show Workloads A, B, and C; IndexServe throughput is resource-normalized.

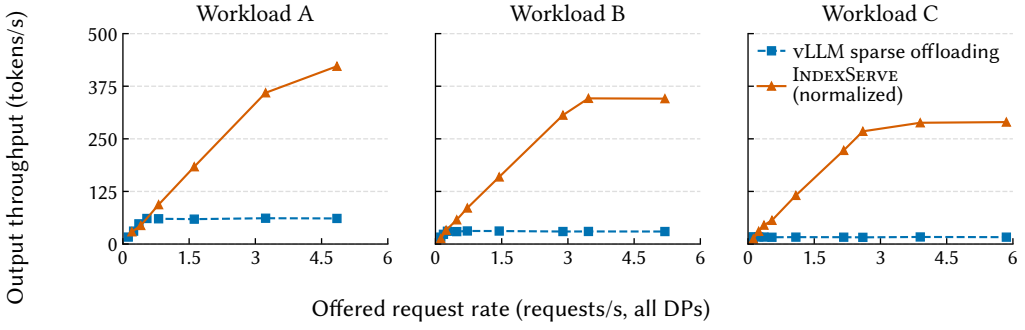

Figure 9: DeepSeek-V3.2 INT8, TP8×DP2, with vLLM `max-num-seqs=24`. Panels show Workloads A, B, and C; IndexServe throughput is resource-normalized.

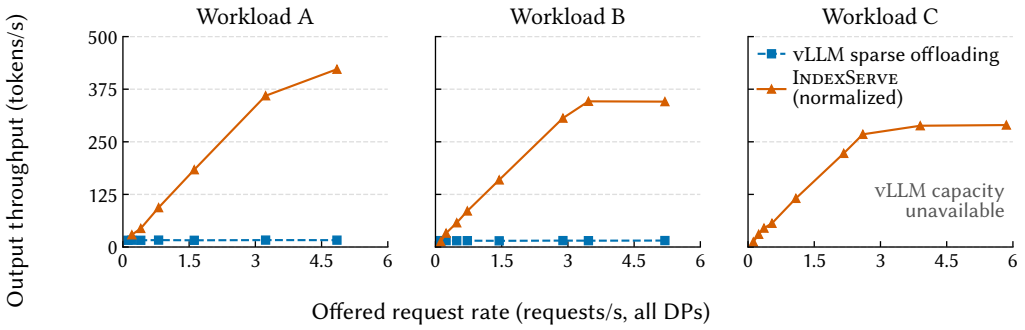

Figure 10: DeepSeek-V3.2 INT8, TP8×DP2, with vLLM `max-num-seqs=32`. Panels show Workloads A, B, and C. vLLM cannot admit Workload C; IndexServe throughput is resource-normalized.

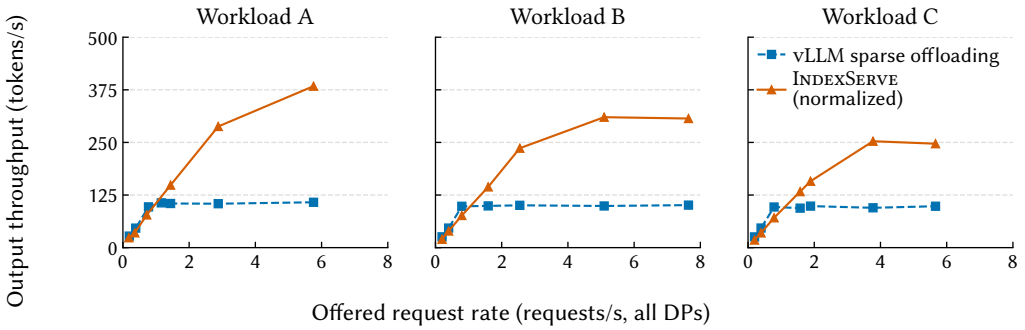

Figure 11: GLM-5.1 INT4, TP8×DP2, with vLLM `max-num-seqs=8`. Panels show Workloads A, B, and C; IndexServe throughput is resource-normalized.

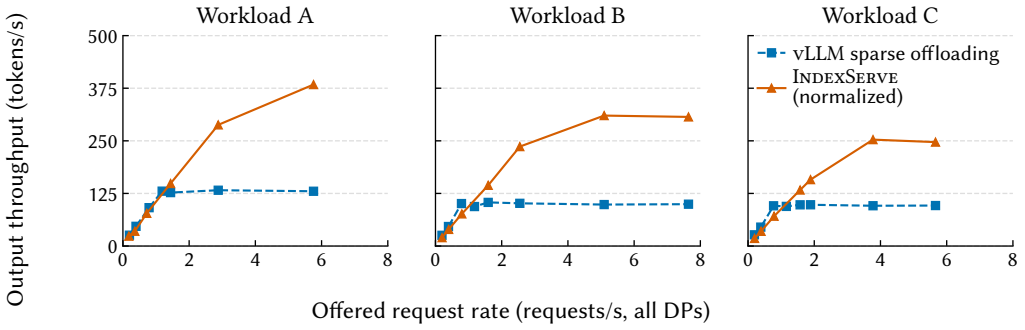

Figure 12: GLM-5.1 INT4, TP8×DP2, with vLLM `max-num-seqs=16`. Panels show Workloads A, B, and C; IndexServe throughput is resource-normalized.

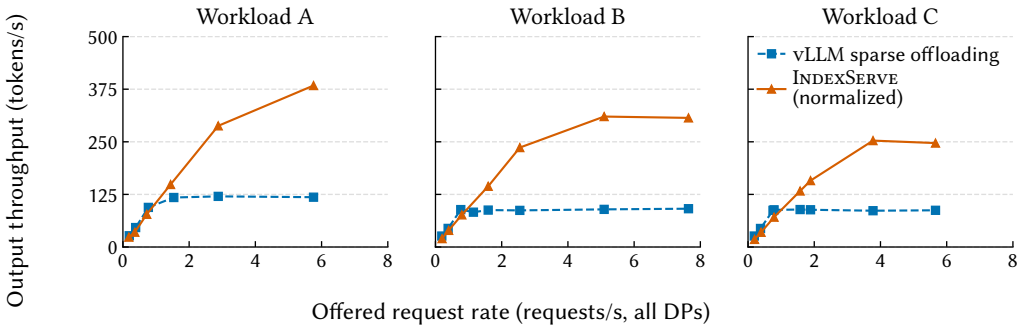

Figure 13: GLM-5.1 INT4, TP8×DP2, with vLLM `max-num-seqs=24`. Panels show Workloads A, B, and C; IndexServe throughput is resource-normalized.

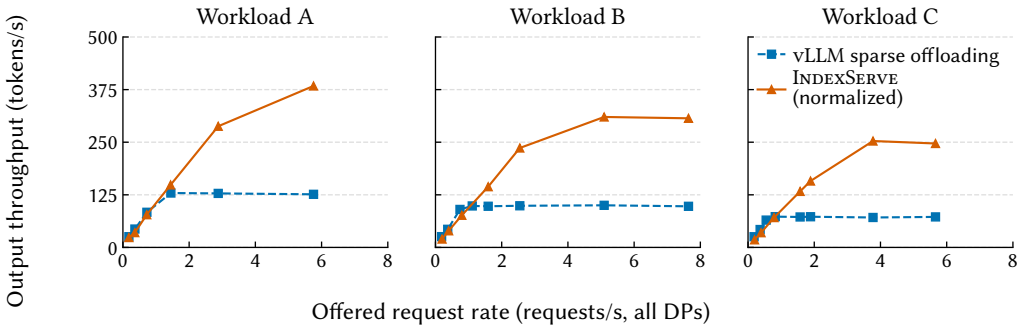

Figure 14: GLM-5.1 INT4, TP8×DP2, with vLLM `max-num-seqs=32`. Panels show Workloads A, B, and C; IndexServe throughput is resource-normalized.

E Sensitivity to Decoder Parallelism

This appendix reports request-rate sweeps for the DP×TP layouts in Section 5. We use GLM-5.1 INT4 and hold the decoder budget at 16 NPUs. The four layouts are DP16×TP1, DP8×TP2, DP4×TP4, and DP2×TP8. A larger TP therefore leaves fewer independent DP replicas while the decoder budget stays fixed. This sweep also tests whether IndexServe's gain comes only from removing rank-local index replication under TP. The colocated baseline's replication factor falls with TP and is gone at TP1. A resource-normalized gain at TP1 therefore cannot come from removing TP replication. The full curves across the four layouts show that the gain stays as the replication factor changes. The sweep does not measure the separate contributions of decoupled index capacity, cross-request service batching, and avoided replication. All figures use the `max-num-seqs=24` source data and follow the Workload A/B/C definitions in Section 5.1. The setting applies to vLLM; IndexServe has no corresponding static partition.

The horizontal axis is the total offered request rate across all DP replicas. The curves therefore show both how many independent replicas a layout has and how each wider TP group runs a request. We plot only vLLM sparse offloading and IndexServe. Each IndexServe point is labeled *IndexServe (normalized)* and scaled by $16/(16+n)$ using the service-NPU count n recorded for that layout and workload.

Maximum throughput on Workload C. Table 5 gives the points in Figure 5(d). The four layouts use 16, 16, 13, and 7 service NPUs. Even at TP1, where the baseline has no TP index replication, IndexServe gains 5.16× after counting those service resources. The full curves below come from separately recorded runs. For Workload C, their DP16 and DP8 configurations use 15 service NPUs each, giving normalized peaks of 922.6 and 638.3 tokens/s. Some baseline peaks also differ from

the main panel. Table 5 therefore gives the main-panel values directly, not maxima taken from these curves.

Table 5: Maximum throughput in tokens/s for GLM-5.1 INT4, Workload C. Both systems use 16 decoder NPUs. INDEXSERVE throughput includes the service-NPU cost through the $16/(16+n)$ normalization.

DP×TP	Service NPUs	vLLM	INDEXSERVE (normalized)	Speedup
16×1	16	173.3	893.8	5.16×
8×2	16	268.6	618.4	2.30×
4×4	13	144.2	408.2	2.83×
2×8	7	87.1	252.8	2.90×

Workloads B and C under DP16, and Workload C under DP8, use up the index HBM of the one A3 server given to the Indexer Service. These endpoints are capacity walls, not evidence that indexer compute has saturated. Adding Indexer Service NPUs would add index capacity and could admit larger batches; we do not measure that configuration.

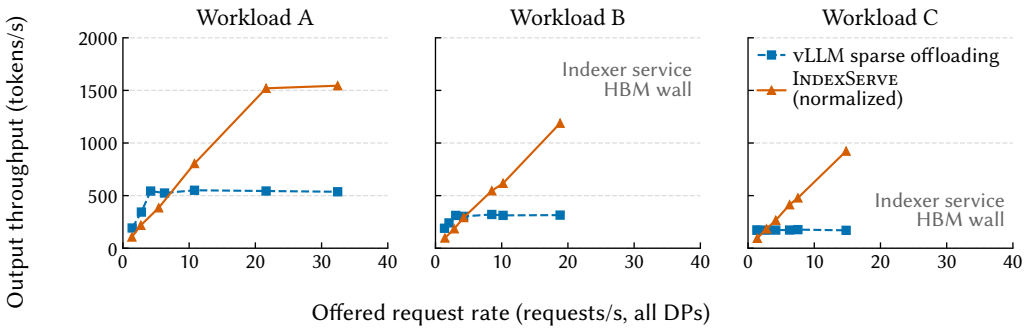

Figure 15: GLM-5.1 INT4 request-rate sweeps with TP1×DP16 on 16 decoder NPUs and vLLM `max-num-seqs=24`. Panels show Workloads A, B, and C; Workloads B and C reach the HBM capacity of one A3 server allocated to the Indexer Service. Additional service NPUs can increase throughput. INDEXSERVE throughput is resource-normalized.

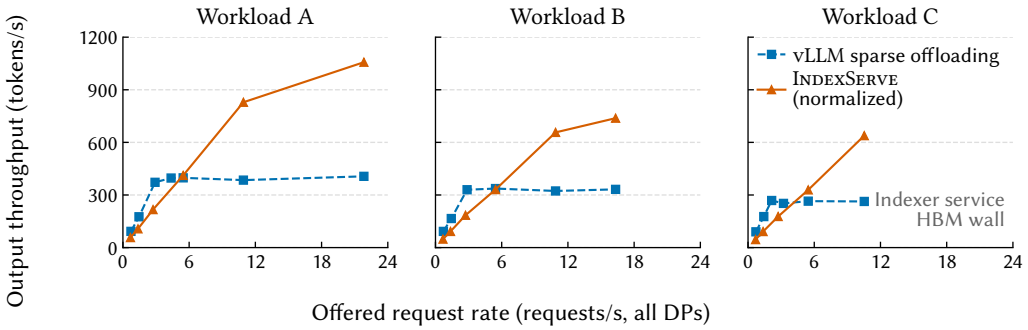

Figure 16: GLM-5.1 INT4 request-rate sweeps with TP2×DP8 on 16 decoder NPUs and vLLM `max-num-seqs=24`. Panels show Workloads A, B, and C; Workload C reaches the HBM capacity of one A3 server allocated to the Indexer Service. Additional service NPUs can increase throughput. INDEXSERVE throughput is resource-normalized.

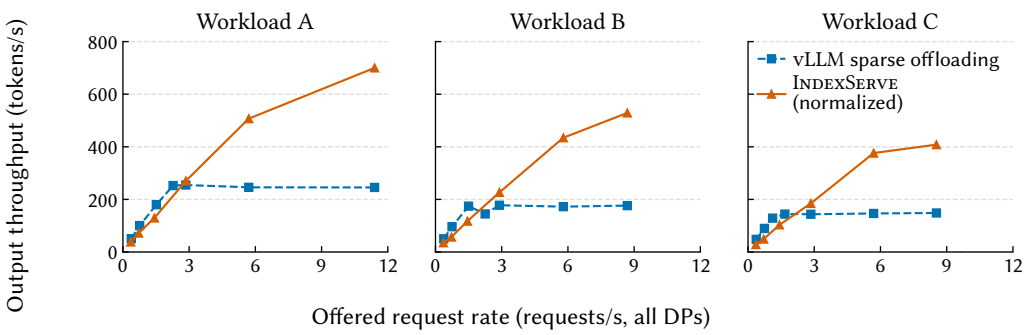

Figure 17: GLM-5.1 INT4 request-rate sweeps with TP4×DP4 on 16 decoder NPUs and vLLM `max-num-seqs=24`. Panels show Workloads A, B, and C; IndexServe throughput is resource-normalized.

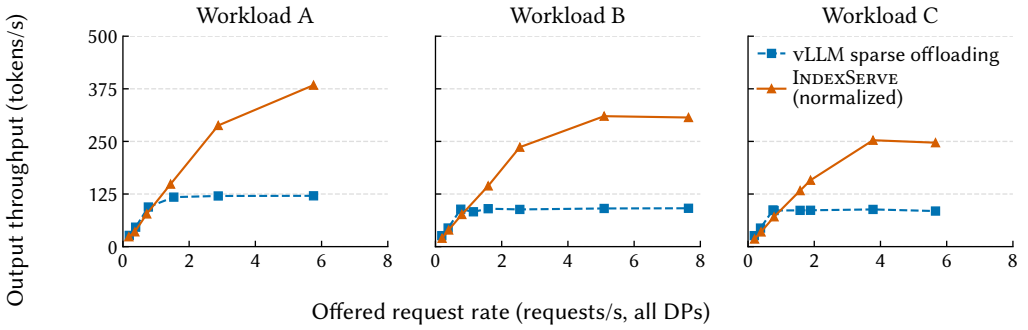

Figure 18: GLM-5.1 INT4 request-rate sweeps with TP8×DP2 on 16 decoder NPUs and vLLM `max-num-seqs=24`. Panels show Workloads A, B, and C; IndexServe throughput is resource-normalized.

F Design Details

This appendix expands the service setup, graph execution, memory management, and scheduling in Section 4. Appendix B gives the payload formats and communication protocol; Appendix A describes the Ascend implementation.

F.1 Session setup and task execution

The proxy assigns a new session to one service accelerator using both free index capacity and expected scan load. Admission counts index pages, page tables, graph workspaces, and communication buffers. Enough space for the index alone does not mean the device can serve it. The service loads the initial index and publishes its logical handle only after that index and the decoder's KV backing are ready. This first load sends the full prefix index; the fixed-size numerical exchange in Section 4.1 applies to later decode calls.

The session keeps the same owner during ordinary decoding. The proxy sets up this route but does not have to relay each call's tensor data through CPU memory. A service task covers one session, token step, and indexer-bearing layer. Tasks may share a batch only when they have the same owner and compatible scoring operations and numerical formats. The service gathers their inputs, appends each key to its session's index, and scans that session's valid history. The new key must be visible before the scan so that the current token can be selected. The service returns each row's selected IDs to its decoder.

Sharing a batch does not merge session state. Each session keeps its own index, valid length, and output. The decoder finds sparse-KV-buffer hits and fetches every missing entry before attention. If a decoder step needs several rows, it must wait for all of them, even when the service puts those rows in different batches. Finishing one service batch early does not by itself let that decoder step resume.

F.2 Stable buffers for graph replay

The decoder captures submission, completion observation, and result use with fixed request and response buffer addresses. Each replay updates the current inputs, index handle, and invocation identity in those buffers. A device-side wait releases KV lookup only after the matching result arrives. A change in decoder batch configuration picks the matching captured graph.

The service captures a separate graph for each supported execution bucket. An execution bucket fixes the batch shape and the addresses of inputs, outputs, and workspaces. Before replay, the scheduler fills row descriptors with the chosen tasks, their index handles, and valid lengths. Unused rows are masked. The graph appends keys, scans valid histories, and picks top-k IDs. Changing row membership or index length changes runtime data, not the graph's buffer bindings.

The page-table directory also has a stable address and enough metadata capacity for the admitted configuration. Adding an index page updates an entry before the consuming graph runs. It does not replace the directory or move existing keys. Growth within this configuration therefore needs no graph recapture. The runtime must stay within the reserved directory capacity; stable addresses do not allow unbounded growth.

The fixed communication buffers use remote memory mappings, while growing indexes use local page tables. Appendix B explains these two mappings and gives the registration lifetime, invocation checks, visibility ordering, slot reuse, and TP ownership rules.

F.3 Paged allocation and capacity

PagedIndex maintains a page table for each session at each indexer-bearing layer. Let P be the number of keys per page and s_I the bytes per key, including any scale. Token position t uses logical page $\lfloor t/P \rfloor$ and offset $t \bmod P$. Physical pages need not be adjacent. Scan/select resolves them through the page table, while the decoder keeps the same logical handle.

For a prefix of length L_r, the initial load allocates $\lceil L_r/P \rceil$ pages per layer. New keys fill the last page. When it is full, the allocator takes a free page and publishes its mapping before the append runs.

Earlier keys stay in place. After a session ends, its pages return to the pool only when all outstanding reads and writes have finished.

For n_I equally sized indexer-bearing layers, the allocated index payload is

$$M_{\text{index}} = \sum_r n_I \lceil L_r/P \rceil \, Ps_I. \tag{5}$$

With N nonempty sessions, unused space in the final pages is less than Nn_IPs_I. This bound covers payload pages only; page tables, alignment, graph buffers, and workspaces need more memory. Fixed-size pages remove the need to find one contiguous region for a whole index. They still leave partly filled final pages and may need separate pools for incompatible formats. Sessions share free pages as they grow. Admission must also count scan load, since a device can have enough memory but too little execution capacity for another session.

F.4 Length groups and batch sizes

Each owner groups ready tasks by the valid history length they will scan. It prefers equal lengths and uses nearby lengths when exact matches are rare. A growing session can change groups without moving its stored index. Grouping limits the extra time a short scan waits at a batch's shared completion barrier.

Offline profiles record execution time $T_c(b)$ for each length group c and supported batch size b. The profiled path includes input preparation, key append, scan/select, and result preparation. Its rate is $b/T_c(b)$. The scheduler picks a target that meets the required rate while keeping batch execution time low. Short indexes often need more rows to spread fixed costs across tasks, whereas long indexes can saturate the scan engine at smaller batches. Adding rows after throughput levels off mainly delays completion.

The target must cover both waiting and execution. Minimizing $T_c(b)/b$ alone minimizes average work per row, not the time a task waits for its result. The scheduler limits aggregation waiting and the predicted execution time of a batch. If too few compatible tasks are ready, or the oldest task hits its aggregation limit, it uses a smaller supported bucket or masks unused rows. These limits do not guarantee a fixed response time under overload.

F.5 Dispatch order and progress

A dispatch opportunity lets one eligible batch run. Short-index groups get more recurring opportunities, which cuts the number of long batches a short task waits behind. Every active long-index group keeps recurring opportunities, and tasks within each group go oldest first. Another ready group can use an empty group's opportunity.

This policy does not interrupt a running scan. Batch execution budgets limit how long one dispatch holds the device. With bounded execution times, finitely many older admitted tasks, and recurring opportunities, long tasks keep making progress. This is not a fixed latency guarantee or a general multi-tenant isolation policy. Length ranges, batch targets, and dispatch shares depend on the profiled workload; the design assumes no universal ratio between short and long groups. New sessions must wait when index memory or scan capacity is exhausted.