SUN: SHARED USE OF NEXT-TOKEN PREDICTION FOR EFFICIENT MULTI-LLM DISAGGREGATED SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

In multi-model LLM serving, decode execution remains inefficient due to model-specific resource partitioning: since cross-model batching is not possible, memory-bound decoding often suffers from severe GPU underutilization, especially under skewed workloads. We propose Shared Use of Next-token Prediction (SUN), the first approach that enables cross-model sharing of decode execution in disaggregated multi-LLM serving. SUN decomposes the execution of a decoder-only Transformer into a prefill module and a decode module, and fine-tunes only the task-specific prefill module, enabling a frozen decode module to be shared across models. This design enables a model-agnostic decode routing policy that balances decode requests across shared workers to maximize utilization. Across diverse tasks and model families, SUN achieves accuracy comparable to full fine-tuning while maintaining system throughput with fewer decode workers. In particular, SUN improves throughput per decode GPU by up to 2.0× over conventional disaggregation while keeping time-per-output-token (TPOT) degradation within 10%. In eight-model multi-node serving, SUN retains 98.7% of baseline throughput with 25% fewer total GPUs at moderate load and increases throughput by 29.1% on the same hardware at higher load. SUN also outperforms popularity-aware allocation and GPU colocation.

1 INTRODUCTION

Recent advances in large language models (LLMs) have expanded their adoption beyond general-purpose chatbots. Increasingly, LLM-based applications rely on multiple domain-specific models or agent models that collaborate to solve complex real-world problems (Singhal et al., 2023; Cui et al., 2026; Wu et al., 2023; Rozière et al., 2024; Shen et al., 2024; Subramaniam et al., 2025; Wang et al., 2025; Su et al., 2025). Service providers therefore face the growing challenge of serving multiple specialized LLMs concurrently; for example, a multi-agent pipeline may invoke a router, domain specialists, and a summarizer.

Challenges in multi-LLM serving are two-fold: intra-model interference and inter-model isolation. Intra-model interference arises from colocating the prefill and decode phases of the same model on shared GPUs. Prefill processes the entire input prompt in parallel to initialize the KV cache and is typically compute-bound, while decode generates one token at a time using this cache and is often memory-bound. When colocated, these phases contend for resources: long prefill computations can stall latency-sensitive decode steps and increase time-per-output-token (TPOT), while frequent decode steps can delay prefill execution and increase time-to-first-token (TTFT) (Zhong et al., 2024; Patel et al., 2024; Hu et al., 2024).

Disaggregated serving mitigates intra-model interference by separating prefill and decode onto different devices (Zhong et al., 2024; Patel et al., 2024; Hu et al., 2024). However, it does not address inter-model isolation. In multi-LLM serving, request rates can be highly skewed across models (Duan et al., 2024; Yao et al., 2025): popular models accumulate queues while less frequently requested models leave their allocated GPUs underutilized (Fig. 1, left). This fragmentation is particularly costly for memory-bound decoding, where idle capacity behind one model cannot directly serve another model's requests.

To this end, we propose **Shared Use of Next-token Prediction (SUN)**, a disaggregated serving paradigm that keeps prefill execution task-specific while sharing decode execution across models.

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.

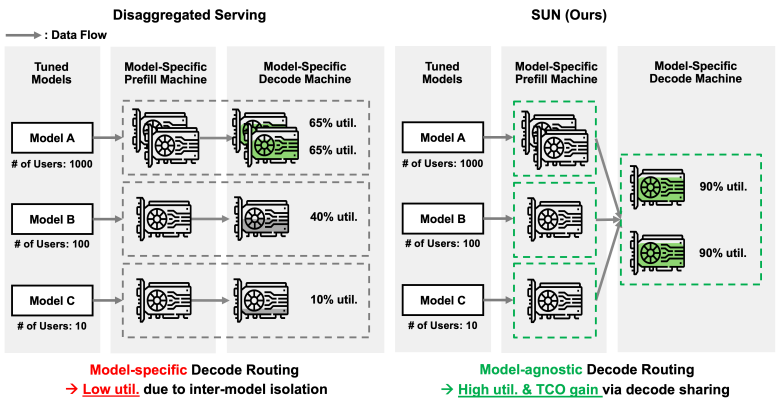


Figure 1: **Overview of SUN for disaggregated multi-LLM serving. (Left)** In conventional disaggregated serving, decode workers are isolated per model, causing GPU underutilization under skewed multi-model workloads. **(Right)** SUN shares a frozen decode module across task-specific models, improving GPU utilization and reducing total cost of ownership (TCO).

SUN targets task-specific models fine-tuned from a common backbone, with the same tokenizer, attention configuration, and KV-cache layout. It uses two mechanisms: prefill-only tuning, which trains each task-specific prefill module with a frozen shared decoder, and model-agnostic routing, which dispatches decode requests to a unified worker pool (Fig. 1, right). This enables cross-model batching while preserving task-specific behavior.

Compared with conventional disaggregated serving, SUN achieves up to $2.0\times$ higher throughput per decode GPU with 50% fewer decode GPUs and TPOT degradation within 10%. SUN also maintains accuracy close to full fine-tuning and avoids adapter-induced decode overhead. With eight models across four nodes, SUN retains 98.7% of baseline throughput on 25% fewer total GPUs at 8 RPS and delivers 29.1% more throughput on the same hardware at 16 RPS. It also improves throughput by 7.0% over popularity-aware allocation and 23.6% over GPU colocation, showing the additional benefit of cross-model decode batching.

Our contributions are:

- **Decode sharing for disaggregated multi-LLM serving.** SUN shares a frozen decoder across task-specific models, removing inter-model isolation through cross-model batching.
- **Prefill-only tuning for shared decoding.** SUN trains task-specific prefill modules to produce decoder-compatible KV caches while keeping shared decode weights frozen.
- **Model-agnostic routing for efficient shared decoding.** Our vLLM-based implementation pools decode workers to balance skewed demand and enable GPU consolidation.

2 Background and Motivation

2.1 LLM Inference Optimization

Continuous Batching Modern LLM serving frameworks such as Orca (Yu et al., 2022), vLLM (Kwon et al., 2023), and SGLang (Zheng et al., 2024) maximize GPU utilization under heterogeneous request lengths via continuous batching, which dynamically admits new requests and retires completed ones at each iteration, and via partial prefill, which splits long prompt processing to reduce prefill-induced stalls under mixed prefill/decode workloads.

Disaggregated Serving Disaggregated serving separates compute-bound prefill from memory-bound decode to reduce interference and improve latency (Zhong et al., 2024; Patel et al., 2024; Hu et al., 2024). Supported by frameworks such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024), it enables phase-specific resource allocation but leaves decode resources partitioned by model in multi-LLM deployments.

2.2 Motivation: Multi-LLM Serving Optimization

Characteristics of Multi-LLM Serving Modern deployments increasingly host task- or domain-specialized models with uneven request rates (Duan et al., 2024; Yao et al., 2025; Luo et al., 2025; Yang et al., 2025b; Wooldridge et al., 2026). Public model-usage rankings illustrate heterogeneous demand (OpenRouter, 2026). Similar skew arises in multi-agent systems, where shared roles such as a manager may be invoked more frequently than individual specialists. This motivates pooling resources across models with different demands.

Challenges in Multi-LLM Serving Disaggregation addresses intra-model interference but retains model-specific decode execution. Popularity-aware allocation assigns more workers to popular models, while GPU colocation places multiple model-specific workers on one GPU, using NVIDIA Multi-Process Service (MPS) for concurrent execution (NVIDIA, 2026). However, colocated workers retain separate weights and batches, requiring model-specific forward passes and additional memory for resident weights. SUN shares identical decode weights across compatible models, enabling cross-model batching that amortizes weight reads over more output tokens and reduces the weight footprint, leaving more memory for request-specific KV caches. Section 4.3 evaluates these serving benefits.

Limitations of Multi-LoRA Serving Multi-LoRA serving reduces inter-model isolation by sharing a frozen backbone across task-specific adapters within a shared GPU pool (Hu et al., 2022; Chen et al., 2023; Sheng et al., 2024), but introduces an accuracy–efficiency trade-off. LoRA restricts task adaptation to low-rank updates, which can limit accuracy relative to full fine-tuning (Biderman et al., 2024; Shuttleworth et al., 2025). SUN instead trains the full prefill copy, supporting richer task adaptation through the prompt KV cache while retaining a shared decoder. During serving, heterogeneous adapters require task-specific computation at each decode step. Punica (Chen et al., 2023) and S-LoRA (Sheng et al., 2024) reduce adapter overhead through specialized batching kernels and memory management, but still execute task-specific adapter operations during decoding. SUN eliminates these operations by confining task adaptation to prefill and using identical frozen decode weights across tasks. It therefore enables cross-model batching with standard decoding kernels, preserving full-parameter adaptation in prefill without adapter-induced decode overhead.

3 Methodology

We present **Shared Use of Next-token Prediction (SUN)**, which keeps prefill execution task-specific while sharing a frozen decode module across compatible models. We first formulate prefill–decode execution, then describe prefill-only tuning and model-agnostic decode routing.

3.1 Formulation

Decoder-only Transformer. A decoder-only Transformer defines an autoregressive distribution over output tokens conditioned on an input prompt. Given a prompt $X = (x_1, \ldots, x_n)$ and previously generated tokens $y_{<t} = (y_1, \ldots, y_{t-1})$, a model with parameters θ defines the next-token distribution as

$$p_\theta(y_t \mid X, y_{<t}) = F_\theta(X, y_{<t}), \tag{1}$$

where $F_\theta(\cdot)$ denotes the forward computation over the full context.

Prefill–Decode Decomposition. To align this model with the execution pattern of disaggregated LLM serving, we decompose its execution into a prefill module and a decode module. Both modules execute the full Transformer with their respective parameters; they are not consecutive groups of layers. The prefill module processes the entire prompt in parallel and produces both the first-token distribution and the prompt KV cache:

$$\big(p_{\theta_p}(y_1 \mid X),\ \mathcal{C}_X\big) = P_{\theta_p}(X), \tag{2}$$

where θ_p denotes the parameters used during prefill and $\mathcal{C}_X$ is the KV cache generated from the prompt.

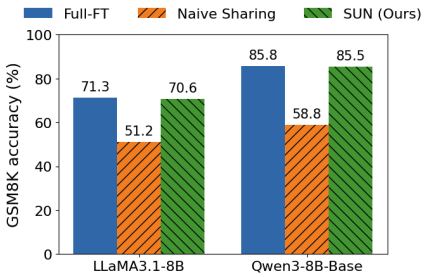


Figure 2: GSM8K accuracy with naive decode sharing and SUN. Prefill-only tuning resolves the KV-cache mismatch of naive sharing and matches Full-FT on both models.

After the first token is generated, the decode module is invoked once per autoregressive step. At decode step t, it consumes the most recent token y_t and the accumulated KV cache $\mathcal{C}_{<t}$, and produces the next-token distribution together with the KV cache for the newly processed token:

$$\left(p_{\theta_p,\theta_d}(y_{t+1} \mid X, y_{\leq t}),\ \mathcal{C}_{y_t}\right) = D_{\theta_d}\left(y_t,\ \mathcal{C}_{<t}\right), \tag{3}$$

where θ_d denotes the decode-side parameters. The accumulated KV cache before processing y_t is

$$\mathcal{C}_{<t} = \mathcal{C}_X \| \mathcal{C}_{y_{<t}}, \tag{4}$$

where $\mathcal{C}_{y_{<t}}$ denotes the KV caches generated by previous decode steps and $\|$ denotes cache concatenation.

For a conventional model, this decomposition is purely conceptual: both phases use the same parameters, i.e., $\theta_p = \theta_d = \theta$. SUN changes this execution model by adapting the prefill-side parameters while keeping the decode-side parameters frozen and shared across tasks.

3.2 SUN: Shared Use of Next-token Prediction

3.2.1 KV-Cache Compatibility Challenge

A straightforward approach reuses a conventionally fine-tuned prefill module $P_{\theta_p^\tau}$ with the frozen base decoder $D_{\theta_d^{\text{base}}}$. Although all tasks then decode with identical parameters, the prompt KV cache $\mathcal{C}_X^\tau$ was optimized jointly with the task-specific decoder $D_{\theta_d^\tau}$, creating a train–inference mismatch. Fig. 2 illustrates the resulting degradation: GSM8K accuracy falls from 71.3% to 51.2% on LLaMA3.1-8B and from 85.8% to 58.8% on Qwen3-8B-Base. Shared decoding therefore requires prefill modules trained to produce caches that the shared decoder can interpret.

3.2.2 Prefill-Only Tuning

SUN trains each task-specific prefill module with the frozen decoder used at inference time, aligning its prompt KV cache with shared decoding. It trains the full prefill copy, using the same number of unique trainable parameters as Full-FT. Gradients pass through the frozen decoder and prompt cache, but only prefill parameters are updated.

For each task τ, SUN initializes both the prefill module and the decode module from the same base model. It then fine-tunes a task-specific prefill module $P_{\theta_p^\tau}$ while keeping the shared decoder $D_{\theta_d^{\text{base}}}$ frozen. Given an input prompt X, the task-specific prefill module produces the first-token distribution and prompt KV cache:

$$\left(p_{\theta_p^\tau}(y_1 \mid X),\ \mathcal{C}_X^\tau\right) = P_{\theta_p^\tau}(X). \tag{5}$$

Given target tokens $Y = (y_1, \ldots, y_T)$, the frozen shared decoder processes the preceding target tokens with teacher forcing and computes next-token likelihoods for $t \geq 2$; the first token is predicted by Eq. 5:

$$p_{\theta_p^\tau,\theta_d^{\text{base}}}(y_t \mid X, y_{<t}) = D_{\theta_d^{\text{base}}}\left(y_{<t},\ \mathcal{C}_X^\tau\right), \tag{6}$$

where $\mathcal{C}_X^\tau$ is produced by the task-specific prefill module, and the decode computation is performed by the frozen shared decoder.

SUN minimizes the negative log-likelihood with respect to the prefill parameters:

$$\mathcal{L}(\theta_p^\tau) = -\sum_{t=1}^{T} \log p_{\theta_p^\tau, \theta_d^{\text{base}}}\left(y_t \mid X, y_{<t}\right). \tag{7}$$

Gradients from Eq. 7 pass through the frozen decoder and $\mathcal{C}_X^\tau$, updating only θ_p^τ. Training with the inference-time decoder aligns the prompt cache with shared decoding, avoiding the train-inference discrepancy. SUN incorporates decode sharing into task-specific model preparation: tuning is performed once per model version, while GPU savings recur throughout serving. Appendix C.2 quantifies this one-time cost and the recurring savings that can amortize it.

3.2.3 Model-Agnostic Decode Routing

SUN uses model-specific routing for prefill and model-agnostic routing for decode. Incoming requests first reach the prefill worker for their task-specific model, which produces $\mathcal{C}_X^\tau$. The cache is then transferred to a shared decode worker. Because all decode requests use the same frozen parameters, requests from different prefill modules can enter one decode batch. Routing over the shared pool allows capacity to follow active requests instead of remaining reserved for individual models.

We implement this pipeline on a vLLM-based disaggregated serving stack (Kwon et al., 2023), using its KV-cache management and transfer mechanisms. Sharing requires a common backbone, tokenizer, attention configuration, and KV layout. Prefix caching must distinguish the prefill model that produced a cache; tensor parallelism must preserve the compatible KV interface.

4 Experimental Results

4.1 Experimental setup

Training Setup We compared full fine-tuning (Full-FT) and SUN on LLaMA3.1-8B (Grattafiori et al., 2024) and Qwen3-Base models (1.7B/8B/14B) (Yang et al., 2025a). For domain adaptation, the models are trained on MetaMathQA-40K (Yu et al., 2024), EvolInstruct-Code-80K (Luo et al., 2024), and xLAM-function-calling-60K (Zhang et al., 2025), targeting math, code generation, and tool use, respectively. We then evaluate each adapted model on its corresponding benchmark: GSM8K/GSM+ (Cobbe et al., 2021; Li et al., 2024) (math), HumanEval/HumanEval+ (Chen et al., 2021; Liu et al., 2023) (coding), and BFCL (Patil et al., 2025) (tool calling), using the Simple Python and Multiple subsets. The details are provided in Appendix A.1.

Inference Setup We consider a multi-LLM serving setup on a single DGX A100 node equipped with 8 GPUs, hosting four LLaMA3.1-8B-based models. The baseline adopts a disaggregated serving configuration that assigns 1 prefill GPU and 1 decode GPU per model instance (4×1P/1D). In contrast, SUN uses one task-specific prefill GPU per model, for a total of four prefill GPUs, while pooling the N decode GPUs across all models as shared decode workers (4P/ND).

We evaluate how far SUN can consolidate the shared decode pool while maintaining the baseline system throughput. Under a fixed offered request rate (RPS), we vary the output sequence length (OSL) and sweep the number of decode GPUs from 1 to 4. We vary the per-model request distribution as described in Section 4.3. Detailed experimental settings are provided in Appendix A.2. We additionally evaluate SUN with eight Qwen3-14B-based models across four nodes at TP=2. In a separate experiment, we compare SUN against popularity-aware isolated serving and MPS-based GPU colocation. Both evaluations are presented in Section 4.3.

4.2 Accuracy after Prefill-Only Tuning

Table 1 demonstrates that SUN achieves accuracy comparable to Full-FT across math, coding, and tool-calling tasks while enabling shared decode execution. On Qwen3-1.7B-Base, SUN is within 0.9 percentage points of Full-FT on both math benchmarks and matches its HumanEval accuracy. On Qwen3-8B-Base, SUN exceeds Full-FT on both coding benchmarks and matches its Simple Python score. Although the decode module is not updated for each task, the task-specific prefill

Table 1: Task accuracy of Full-FT and SUN across LLaMA-3.1-8B and Qwen3-1.7B/8B/14B. SUN uses prefill-only tuning with shared decode execution.

Model Configuration	Decode Execution	Math		Coding		Tool calling	
		GSM8K	GSM+	HumanEval	HumanEval+	Simple Python	Multiple
LLaMA3.1-8B	N/A	25.9	18.0	36.6	29.9	70.5	45.5
+Full-FT	Independent	**71.3**	**49.8**	48.2	**45.7**	**90.0**	88.0
+SUN (Ours)	**Shared**	70.6	49.3	**49.4**	41.5	89.8	**90.5**
Qwen3-1.7B-Base	N/A	9.9	9.8	48.2	39.6	82.5	60.0
+Full-FT	Independent	**75.0**	**54.8**	**67.1**	**59.8**	**89.8**	**90.0**
+SUN (Ours)	**Shared**	74.1	53.9	**67.1**	58.5	88.8	89.0
Qwen3-8B-Base	N/A	11.8	12.5	68.3	61.6	81.5	80.0
+Full-FT	Independent	**85.8**	**65.7**	83.5	74.3	**93.3**	**92.0**
+SUN (Ours)	**Shared**	85.5	64.5	**84.1**	**76.2**	**93.3**	91.5
Qwen3-14B-Base	N/A	25.9	24.5	68.9	62.3	88.0	77.0
+Full-FT	Independent	**88.1**	**67.5**	85.3	78.0	**93.8**	**92.5**
+SUN (Ours)	**Shared**	85.4	64.6	**88.4**	**79.9**	93.5	**92.5**

module learns prompt KV caches that the frozen shared decoder can interpret. These results show that prefill-only adaptation can preserve task accuracy while keeping decode execution shared.

Task-dependent differences remain. On Qwen3-14B-Base, SUN trails Full-FT by 2.7 and 2.9 percentage points on GSM8K and GSM+, respectively, while exceeding Full-FT on both coding benchmarks and remaining comparable on tool calling. LLaMA3.1-8B also shows a 4.2-point gap on HumanEval+, despite higher HumanEval accuracy. Overall, the results support prefill-only tuning as an effective way to retain task specialization across model sizes, with accuracy comparable to Full-FT in most evaluated settings.

4.3 Performance under Model-Agnostic Routing Policy

We evaluate the serving benefits of shared decoding. In controlled experiments, per-model request rates follow a Zipf distribution, proportional to $i^{-\alpha}$ for popularity rank i (Yao et al., 2025): $\alpha = 0$ gives uniform demand, and larger α concentrates requests on fewer models. We evaluate consolidation at $\alpha = 0$ and 1.5, then sweep α to assess robustness. We further evaluate multi-node serving with variable-length requests and practical serving optimizations, and compare SUN against popularity-aware allocation and MPS colocation.

Decode-GPU consolidation. SUN pools decode requests from compatible task-specific models to consolidate decode workers and improve utilization. Fig. 3 shows the throughput–interactivity trade-off as we vary the decode pool size. We define interactivity as the inverse of TPOT, representing the per-request token generation rate; higher interactivity means faster token delivery.

Under uniform demand at OSL$= 256$, SUN maintains total throughput, reduces the decode pool from 4 to 3 workers, improves throughput per decode GPU by 33%, and incurs only 7% TPOT degradation.

Under skewed demand, shared decoding lets heavily requested models use capacity that would otherwise remain assigned to lightly requested models. With $\alpha = 1.5$, SUN achieves up to $2.0\times$ higher throughput per decode GPU while keeping TPOT degradation within 10% even when using only 2 decode workers for OSL$= 128$ and 256. Thus, SUN reduces decode-GPU demand by up to 50% while maintaining total throughput with a modest TPOT increase.

Robustness to workload skew. We vary α at fixed total offered load to isolate request imbalance (Fig. 4). As skew increases, popular models bottleneck their dedicated decode workers while others remain underutilized. At $\alpha = 3.0$, the model-isolated baseline throughput drops by 19.0% relative to uniform demand.

SUN maintains stable interactivity and throughput by redistributing requests over shared decode workers. At $\alpha = 3.0$, SUN improves interactivity by 44.1% and total throughput by 29.2% relative to the baseline; gains persist across the evaluated skew levels.

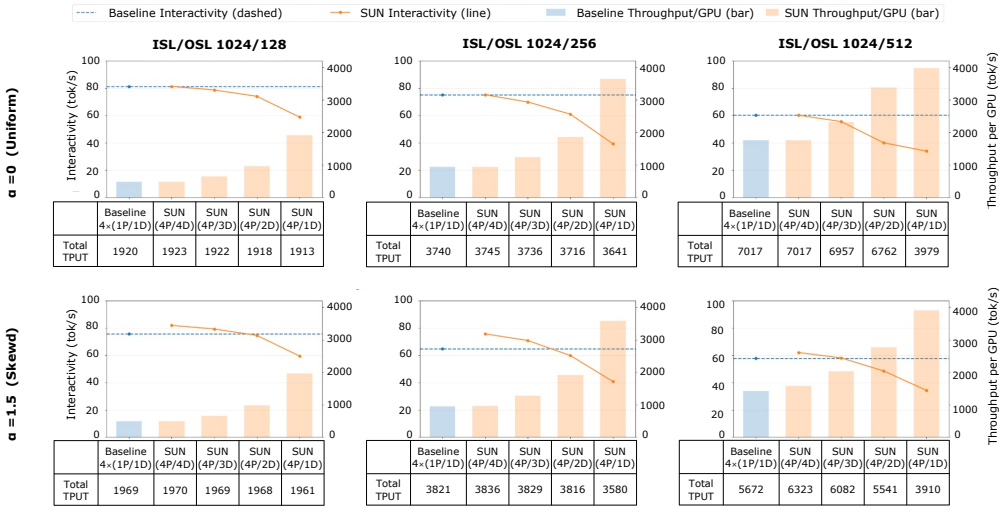


Figure 3: Decode-GPU consolidation under uniform (α=0) and skewed (α=1.5) demand. SUN (4P/KD) shares decode workers across models; the baseline uses 4×(1P/1D). Bars show throughput per decode GPU, lines show interactivity ($= 1/\text{TPOT}$), and tables show total throughput.

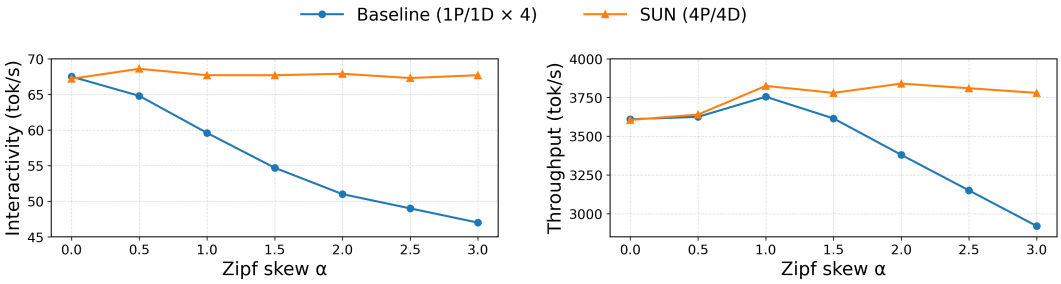


Figure 4: Interactivity and throughput under varying Zipf skew α at fixed total offered load: model-isolated serving (1P/1D $\times$ 4) versus SUN (4P/4D).

Multi-node serving under realistic conditions. We consider eight task-specific models fine-tuned from Qwen3-14B across four eight-H100 nodes at TP=2. This deployment combines variable-length DeltaZip-based requests (Yao et al., 2025), 70% hot-model demand, prefix caching, chunked prefill, and cross-node KV transfer. Table 2 compares model-isolated serving with SUN using two or one shared decode nodes, keeping two prefill nodes fixed. Appendix A.4 details the workload and configuration.

At 8 RPS, SUN and isolated serving have nearly identical throughput on 32 GPUs. SUN can instead consolidate to 24 GPUs while retaining $0.99\times$ the baseline throughput: removing one decode node halves decode GPUs and reduces total GPU usage by 25%. Throughput per total GPU reaches $1.32\times$ the baseline, with p95 TPOT rising from 10.6 to 14.3 ms. Pooling thus converts underutilized capacity into GPU savings while preserving nearly all output throughput.

At 16 RPS, SUN uses the full 32-GPU pool to achieve $1.29\times$ the baseline throughput and lower p95 TPOT from 14.1 to 12.7 ms. Hot-model requests can use all shared decoders instead of being confined to one model-specific worker. Consolidating to 24 GPUs at this load retains $0.91\times$ the baseline throughput. Shared decoding therefore supports GPU consolidation at moderate load and higher throughput on the same hardware at higher load.

Popularity-aware allocation and GPU colocation. We compare SUN with popularity-aware allocation and MPS colocation under A/B/C demand of 70/15/15 (Table 3), using this distribution to configure both baselines in advance. Isolated serving assigns four decode GPUs as A/A/B/C; MPS

Table 2: Eight-model multi-node serving. Parentheses normalize throughput to isolated serving at the same RPS. P/D denote eight-GPU nodes; GPU counts include prefill and decode. Bold marks the best performance values within each RPS group.

RPS	Configuration	GPUs	Throughput ↑ (tok/s)	Throughput/GPU ↑ (tok/s)	p95 TPOT ↓ (ms)
8	Isolated 2×(1P/1D)	32	1,838.6 (1.00×)	57.5 (1.00×)	10.6
8	SUN 2P/2D	32	**1,842.7 (1.00×)**	57.6 (1.00×)	**9.0**
8	SUN 2P/1D	24	1,814.6 (0.99×)	**75.6 (1.32×)**	14.3
16	Isolated 2×(1P/1D)	32	2,481.3 (1.00×)	77.5 (1.00×)	14.1
16	SUN 2P/2D	32	**3,204.1 (1.29×)**	**100.1 (1.29×)**	**12.7**
16	SUN 2P/1D	24	2,248.0 (0.91×)	93.7 (1.21×)	19.7

Table 3: Popularity-aware allocation and GPU colocation under 70/15/15 demand. Throughput per GPU includes both prefill and decode resources: eight GPUs in the first three rows and seven in the last. Bold marks the best value per metric.

System	Decode placement	p95 TPOT ↓ (ms)	Throughput ↑ (tok/s)	Throughput/GPU ↑ (tok/s)
Popularity-aware isolated	A/A/B/C	19.36	3,681.5	460.2
MPS colocation	A/A/AB/AC	23.90	3,187.8	398.5
SUN	4 shared	**17.11**	**3,940.7**	492.6
SUN	3 shared	18.90	3,740.6	**534.4**

uses A/A/AB/AC, where AB denotes separate A and B workers on one GPU. MPS gives A access to every decode GPU, with exactly 30 same-length requests per GPU. All workers remain resident, without loading or reconfiguration costs (Appendix A.5).

At the same eight-GPU budget, SUN improves throughput by 7.0% over popularity-aware allocation and 23.6% over MPS, with 28.4% lower p95 TPOT than MPS. With one fewer decoder, SUN still exceeds isolated throughput by 1.6% and improves throughput per total GPU by 16.1%.

These gains persist after accommodating the hot model and balancing requests across GPUs. MPS retains separate model-specific engines and decode batches; SUN shares identical decode weights, enabling cross-model batching and avoiding multiple task-specific decoder copies on colocated GPUs. SUN thus improves efficiency beyond demand-aware placement and physical GPU sharing. Appendix C.1 extends the comparison to twelve services across four H100 nodes with longer outputs and higher concurrency.

4.4 Comparisons to multi-LoRA

We compare SUN with multi-LoRA serving, which shares a frozen base model across task-specific adapters. Detailed setup for multi-LoRA can be found in Appendix A.3. We first evaluate whether SUN preserves task accuracy compared with full fine-tuning and LoRA. As shown in Fig. 5(a), rank-32 LoRA reduces average accuracy by 2.5% on math and 4.5% on coding relative to Full-FT. SUN trains the full task-specific prefill copy while sharing the decoder, with corresponding accuracy gaps of 0.8% and 3.2%. This retains full-parameter adaptation capacity in prefill while removing task-specific adaptation from the decode path.

Fig. 5(b) compares serving performance under the same 4P/4D budget. Multi-LoRA enables task-agnostic scheduling by colocating the base model with task-specific adapters, but incurs adapter execution overhead. Under the uniform workload ($\alpha = 0$), this increases TTFT and TPOT by 17.0% and 60.0%, respectively, and reduces throughput by 17.4% compared with the baseline. By avoiding adapter computation in the decode path, SUN reduces TPOT by 37.5% and improves throughput by 21.0% over Multi-LoRA. Under the skewed workload ($\alpha = 3$), Multi-LoRA still remains 8.7% below the baseline throughput. SUN instead improves throughput by 29.2% over the baseline, while reducing TPOT by 28.4% over Multi-LoRA. Unmerged adapters require task-specific computation at every output token, rather than only during initialization. Offline merging removes those kernels but

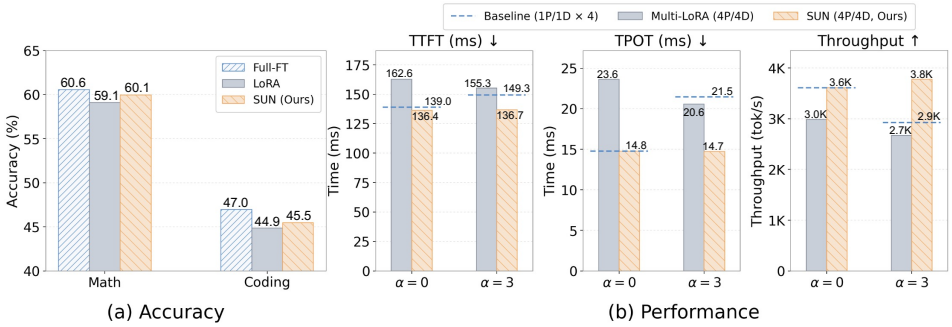


Figure 5: **Comparison with multi-LoRA serving.** (a) Average math (GSM8K/GSM+) and coding (HumanEval/HumanEval+) accuracy. (b) TTFT, TPOT, and throughput at the same 4P/4D budget. Dashed lines mark the baseline; bars show Multi-LoRA and SUN.

yields task-specific weights and separate decode batches. SUN keeps identical decode weights across tasks, combining load balancing with adapter-free shared decoding.

5 Related Works

5.1 Disaggregated LLM Inference

Disaggregating prefill and decode has become a dominant paradigm in production LLM serving: Splitwise (Patel et al., 2024), DistServe (Zhong et al., 2024), and Mooncake (Qin et al., 2025) show that phase-level disaggregation resolves resource contention and maximizes throughput. SUN builds on this foundation but extends it to multi-LLM settings by sharing the decoder across models, addressing resource fragmentation not handled by prior work.

5.2 Multi-LLM Serving Architectures

AlpaServe (Li et al., 2023) combines statistical multiplexing with model parallelism to share serving resources across models. Multi-LoRA systems such as Punica (Chen et al., 2023) and S-LoRA (Sheng et al., 2024) share a frozen base model across adapters, but require specialized runtime support and incur adapter-induced decoding overhead. In contrast, SUN shares a single frozen decode module across models without runtime adapter switching, enabling efficient multi-LLM serving with standard decoding kernels.

6 Conclusion

In this work, we address a key challenge in disaggregated multi-LLM serving: inter-model isolation arising from model-specific partitioning of memory-bound decoding resources. We propose SUN, which removes this isolation by sharing a common decode module across compatible task-specific models while fine-tuning task-specific prefill modules. We show that prefill-only tuning aligns each prefill module to produce KV caches that are directly interpretable by a shared decoder, enabling shared decode execution with accuracy close to full fine-tuning in the evaluated settings. By pooling decode workers and flexibly redistributing capacity under skewed multi-LLM workloads, SUN improves utilization and can reduce TCO in multi-LLM disaggregated serving.

AI USE STATEMENT

In this work, we used generative AI tools to assist with experiment execution and with drafting and polishing the manuscript. We have not used generative AI tools to formulate the central research idea, design the experiments, determine the manuscript's structure, organize its main content, or establish the direction of revisions. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Dan Biderman, Jacob Portes, Jose Javier Gonzalez Ortiz, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley, Jonathan Frankle, Cody Blakeney, and John Patrick Cunningham. LoRA learns less and forgets less. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=aloEru2qCG. Featured Certification.

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving, 2023. URL https://arxiv.org/abs/2310.18547.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Jiaxi Cui, Munan Ning, Zongjian Li, Bohua Chen, Yang Yan, Hao Li, Bin Ling, Yonghong Tian, and Li Yuan. Chatlaw: A multi-agent legal assistant based on a role-aligned mixture-of-experts architecture, 2026. URL https://arxiv.org/abs/2306.16092.

Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: Flexible spatial-temporal multiplexing for multiple llm serving, 2024. URL https://arxiv.org/abs/2404.02015.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. Inference without interference: Disaggregate llm inference for mixed downstream workloads, 2024. URL https://arxiv.org/abs/2401.11181.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph 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*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Qintong Li, Leyang Cui, Xueliang Zhao, Lingpeng Kong, and Wei Bi. GSM-plus: A comprehensive benchmark for evaluating the robustness of LLMs as mathematical problem solvers. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2961–2984, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.163. URL https://aclanthology.org/2024.acl-long.163/.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 663–679, Boston, MA, 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/li-zhouhan.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025. URL https://arxiv.org/abs/2503.21460.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=UnUwSIgK5W.

NVIDIA. CUDA Multi-Process Service documentation, 2026. URL https://docs.nvidia.com/deploy/mps/. Accessed September 23, 2026.

OpenRouter. LLM Rankings, 2026. URL https://openrouter.ai/rankings. Accessed September 26, 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 *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, 2024. doi: 10.1109/ISCA59077.2024.00019.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=2GmDdhBdDk.

Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving, 2025. URL https://arxiv.org/abs/2407.00079.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218. USENIX Association, 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shahrad.

Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. Small LLMs are weak tool learners: A multi-LLM agent. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 16658–16680, Miami, Florida, USA, November

2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.929. URL https://aclanthology.org/2024.emnlp-main.929/.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters, 2024. URL https://arxiv.org/abs/2311.03285.

Reece S Shuttleworth, Jacob Andreas, Antonio Torralba, and Pratyusha Sharma. LoRA vs full fine-tuning: An illusion of equivalence. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=xp7B8rkh7L.

Karan Singhal, Tao Tu, Juraj Gottweis, Rory Sayres, Ellery Wulczyn, Le Hou, Kevin Clark, Stephen Pfohl, Heather Cole-Lewis, Darlene Neal, Mike Schaekermann, Amy Wang, Mohamed Amin, Sami Lachgar, Philip Mansfield, Sushant Prakash, Bradley Green, Ewa Dominowska, Blaise Aguera y Arcas, Nenad Tomasev, Yun Liu, Renee Wong, Christopher Semturs, S. Sara Mahdavi, Joelle Barral, Dale Webster, Greg S. Corrado, Yossi Matias, Shekoofeh Azizi, Alan Karthikesalingam, and Vivek Natarajan. Towards expert-level medical question answering with large language models, 2023. URL https://arxiv.org/abs/2305.09617.

Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, Yi Dong, Evelina Bakhturina, Tao Yu, Yejin Choi, Jan Kautz, and Pavlo Molchanov. Toolorchestra: Elevating intelligence via efficient model and tool orchestration, 2025. URL https://arxiv.org/abs/2511.21689.

Vighnesh Subramaniam, Yilun Du, Joshua B. Tenenbaum, Antonio Torralba, Shuang Li, and Igor Mordatch. Multiagent finetuning: Self improvement with diverse reasoning chains. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=JtGPIZpOrz.

Junlin Wang, Jue WANG, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=h0ZfDIrj7T.

Michael J. Wooldridge, Attila Bagoly, Jonathan J. Ward, Emanuele La Malfa, and Gabriel Paludo Licks. Fetch.ai: An architecture for modern multi-agent systems, 2026. URL https://arxiv.org/abs/2510.18699.

Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann. Bloomberggpt: A large language model for finance, 2023. URL https://arxiv.org/abs/2303.17564.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

Yingxuan Yang, Huacan Chai, Shuai Shao, Yuanyi Song, Siyuan Qi, Renting Rui, and Weinan Zhang. Agentnet: Decentralized evolutionary coordination for llm-based multi-agent systems. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 107309–107336. Curran Associates, Inc., 2025b. doi: 10.52202/085713-3578. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/9a379c1b05793d1c42dc832269834515-Paper-Conference.pdf.

Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. Deltazip: Efficient serving of multiple full-model-tuned llms. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 110–127, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3717468. URL https://doi.org/10.1145/3689031.3717468.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA,

2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T. Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models, 2024. URL https://arxiv.org/abs/2309.12284.

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Zhiwei Liu, Yihao Feng, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. xLAM: A family of large action models to empower AI agent systems. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 11583–11597, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.578. URL https://aclanthology.org/2025.naacl-long.578/.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

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 *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

A Experimental Setup

A.1 Training Setup

Prefill-Only Tuning Configurations. For prefill-only tuning of SUN, we fine-tune the task-specific prefill module while keeping the shared decoder frozen. We use the AdamW optimizer with a linear learning rate scheduler and a warmup ratio of 0.1. We perform a learning rate search over {2e-6, 5e-6, 1e-5, 2e-5, 5e-5}, and report the best results. The global batch size is set to 128, and models are trained for a single epoch, which we find sufficient for convergence.

A.2 Controlled Four-Model Inference Setup

Testbed and serving stack. We run the controlled four-model inference experiments on a single DGX A100 node with 8 NVIDIA A100 GPUs. We serve N_{model}=4 model instances (referred to as agents) using a disaggregated prefill–decode serving architecture built on vLLM (Kwon et al., 2023). Each agent corresponds to one deployed model instance (a task-specialized fine-tuned variant of the same backbone with identical architecture and size). Each request is split into a prefill phase (processing the input prompt and producing KV cache) and a decode phase (streaming token generation from the KV cache).

Serving configurations. We compare two configurations. Baseline (model-isolated 4×1P/1D): each agent is assigned one dedicated prefill GPU and one dedicated decode GPU (4 prefill + 4 decode GPUs). SUN (decoder pooling 4P/KD): each agent keeps a dedicated prefill GPU, while decode GPUs are pooled across agents with pool size $K \in \{1, 2, 3, 4\}$. When $K < 4$, SUN consolidates decode capacity while keeping the prefill footprint unchanged.

Routing policy. Prefill routing is agent-specific: each request is deterministically routed to its corresponding agent prefill worker. Decode routing is configuration-dependent: the baseline routes decode to a fixed per-agent decode worker (model-isolated), whereas SUN dispatches decode requests using a model-agnostic load distribution rule across the shared decode pool of size K, independent of agent identity. This isolates the effect of pooling from model-specific scheduling heuristics.

Workloads and offered load. We use streaming generation with fixed input sequence length (ISL) and target output sequence length (OSL). To ensure comparable decode work across configurations, we enforce the generated output length to stay close to the target OSL and verify this empirically; for example, with OSL= 512 we observe a mean realized output length of 511.8 tokens with std. 0.5 (min 509, max 513). We control the offered request rate per agent at the client and define the offered total request rate as the sum across agents.

Skewed workloads (Zipf). To emulate traffic skew in multi-LLM serving, we distribute the offered total request rate R_{total} across the four agents according to a Zipf distribution (Yao et al., 2025):

$$P(i) = \frac{i^{-\alpha}}{\sum_{j=1}^{N_{\text{model}}} j^{-\alpha}}, \qquad R_i = R_{\text{total}} \cdot P(i),$$

where i is the popularity rank and α controls skew (α=0 is uniform).

Caching configuration. We disable prefix caching for the controlled four-model experiments to avoid cache-hit artifacts in TTFT/TPOT and to ensure that results reflect scheduling and pooling effects rather than prompt reuse.

Metrics. We report output-token throughput (tok/s) and token-level latency metrics from the streaming client: TTFT, ITL, and TPOT. Figure 3 normalizes throughput by decode GPUs; the per-GPU values in Tables 2 and 3 count all prefill and decode GPUs. Tables report mean or p95 TPOT as indicated in their column headings. We define interactivity as the inverse of TPOT, $I = 1/\text{TPOT}$ (token/ms). For readability, we report interactivity in tok/s as $I_{\text{tok/s}} = 1000/\text{TPOT}_{\text{ms}}$ and use this definition consistently across figures.

Measurement protocol. Each controlled four-model experiment runs continuously with an initial grace period, followed by a fixed measurement window. Unless stated otherwise, we use a 30 s grace period and a 60 s measurement window. For long-output settings (e.g., ISL=1024, OSL=2048), we extend the window (e.g., 60 s grace + 180 s measurement) to reduce variance by collecting more completed requests. All metrics are computed over requests completed within the measurement window.

Achieved vs. offered load sanity check. The main Zipf-skew sweep uses ISL$= 1024$, OSL$= 2048$, and offered total RPS$= 2$. Appendix B reports achieved/offered request-rate ratios across load and skew levels to contextualize this operating point.

A.3 MULTI-LORA SETUP

This setup applies to the multi-LoRA comparison in Section 4.4. For training, we follow the same setup as in Appendix A.1 unless otherwise specified. For LoRA fine-tuning, we search the learning rate over $\{5 \times 10^{-5}, 1 \times 10^{-4}, 2 \times 10^{-4}\}$ and use rank $r = 32$ with LoRA scaling factor $\alpha = 64$. All other optimization hyperparameters, datasets, and evaluation protocols are kept identical to the main training setup.

For inference, we evaluate Multi-LoRA under the same disaggregated serving setting as the main experiments, using a 4P/4D configuration. Each prefill and decode worker hosts the shared base model together with all four task-specific LoRA adapters. Therefore, unlike model-isolated serving, Multi-LoRA does not require model-dependent worker assignment in either the prefill or decode phase. Incoming requests can be routed to any prefill worker, and their decode phases can likewise be served by any decode worker. This gives Multi-LoRA the benefit of task-agnostic scheduling, while still requiring adapter-specific computation during both prefill and decode.

A.4 Multi-Node Serving Setup

This experiment extends the fixed-length evaluation to variable-length requests, more models, cross-node KV transfer, and additional serving optimizations. We consider eight Qwen3-14B-based task-specific models across four eight-H100 nodes at TP=2, with four workers per node. The model-isolated baseline uses two prefill and two decode nodes (32 GPUs), assigning each model its own decode worker. SUN uses the same two prefill nodes and either two shared decode nodes (32 GPUs total) or one shared decode node (24 GPUs total). Here, P/D count nodes, unlike the GPU counts in the controlled 4P/KD setup.

Workload provenance. We use DeltaZip-based workloads (Yao et al., 2025), combining prompts sampled from LMSYS Chatbot Arena with Poisson request arrivals and a skewed model-popularity distribution derived from Azure Functions serverless traces (Shahrad et al., 2020). The Azure component supplies a popularity proxy, not LLM prompts, input/output lengths, or the original request arrival sequence.

The source workload file is `azure.ar=1.0.jsonl` in the DeltaZip artifact repository.[1] Each record contains a prompt, a benchmark-relative timestamp in seconds, a target model, and generation-length settings (`min_tokens`, `max_tokens`, and `ignore_eos`). These output lengths are benchmark generation settings rather than Azure LLM completion-length measurements. In the filename, `azure` identifies the popularity source, while `ar=1.0` denotes the nominal Poisson arrival rate of one request/s.

Evaluated configuration. The eight-model experiments use offered rates of 8 and 16 requests/s and assign 70% of requests to the hot model. These are the evaluated workload settings, distinct from the source file's nominal rate and finite-sample model frequencies. Prefix caching and chunked prefill are enabled in every arm, and each request transfers KV across nodes.

Table 2 and Section 4.3 report the resulting consolidation and throughput gains at 8 and 16 RPS.

A.5 Popularity-Aware Allocation and MPS Setup

We detail the controlled comparison in Table 3, which evaluates SUN's additional benefit over demand-aware placement and physical GPU sharing.

Three-model setup. We replay 120 requests at 4 RPS on one eight-A100-80GB node, with ISL/OSL=1024/2048. The A/B/C demand distribution of 70/15/15 gives 84/18/18 requests and is used to configure both baselines in advance. All arms use four prefill workers allocated A/A/B/C. The popularity-aware isolated baseline also allocates decode workers A/A/B/C. MPS places A on all four decode GPUs, with per-GPU requests $30A$, $30A$, $12A + 18B$, and $12A + 18C$. Thus, every physical decode GPU receives 30 same-length requests. All workers remain resident; every MPS client has a 100% active-thread cap, and loading, migration, and reconfiguration costs are excluded. SUN uses the same prefill allocation with four or three shared decode workers.

These placements give the hot model access to every MPS decode GPU and equalize request counts across physical GPUs, while preserving separate model-specific engines and batches. Section 4.3 discusses the results; Appendix C.1 extends the comparison to twelve services with longer outputs and higher concurrency.

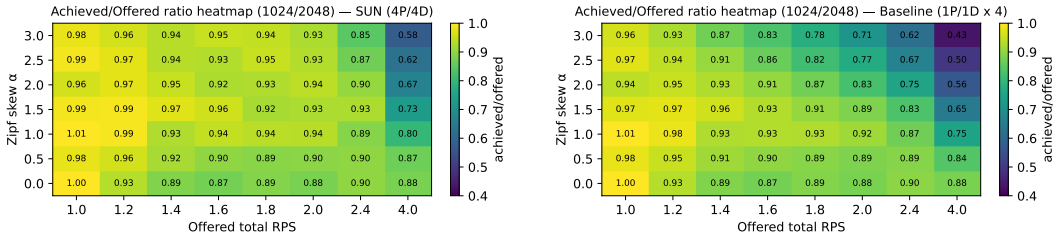
[1] `https://github.com/eth-easl/deltazip-ae/blob/main/scripts/workload/azure.ar=1.0.jsonl`

B ACHIEVED VS. OFFERED LOAD SANITY CHECK

To validate the Zipf-α sweep in Section 4.3, we analyze whether the measured trends arise from robustness to request imbalance rather than artifacts of extreme overload. Specifically, we report the achieved/offered request-rate ratio over a grid of offered total RPS and Zipf skew α in Fig. 6. The ratio is defined as the measured achieved request rate divided by the offered total request rate. A value below 1 indicates that the system does not fully sustain the injected open-loop load due to queueing, backlog, or possible request drops under severe overload.

At the operating point used in the main skew sweep, namely ISL$= 1024$, OSL$= 2048$, and offered total RPS$= 2$, the achieved/offered ratio can fall below 1 as skew increases. This indicates a near-saturation regime with incipient backlog. However, the model-isolated baseline degrades much more sharply with increasing α, whereas SUN remains consistently closer to 1. This shows that SUN sustains a larger fraction of the offered request rate under heavy skew by redistributing decode requests across the shared worker pool.

Since the realized output length remains close to the target OSL in our runs, the throughput differences in the skew sweep mainly reflect differences in achieved request rate rather than output-length drift. Overall, Fig. 6 supports interpreting the main skew-sweep results as improved robustness to workload skew under the same offered load.



(a) SUN with a shared decode pool (4P/4D)

(b) Model-isolated baseline (1P/1D $\times$ 4)

Figure 6: Achieved/offered request-rate ratio under varying offered load and Zipf skew. Values close to 1 indicate that the system sustains the injected open-loop load, while values below 1 indicate backlog or overload. Compared with the model-isolated baseline, SUN remains closer to 1 under high skew, showing that shared decoding sustains a larger fraction of the offered load under request imbalance.

C Additional Experiments

C.1 Twelve-Service MPS Evaluation

We extend the MPS comparison in Table 3 to twelve services with longer outputs and higher concurrency. This tests whether SUN's cross-model batching retains its advantage when MPS has more concurrent requests available for model-specific batching.

Setup. We evaluate twelve logical Qwen3-14B services on four eight-H100 nodes, with sixteen prefill and sixteen decode GPUs at TP=1. The 5,650-request workload doubles the benchmark output-length settings and arrival rate; maximum total request length is 9,447 tokens. Demand weights for A–L are $23/4/3/3/2/2/2/2/1/1/1/1$. Request counts are 2,888 for A, 502 for B, 377 each for C/D, 251 each for E–H, and 125–126 each for I–L.

The isolated baseline has sixteen decode engines: five for A and one for each remaining service. Its first-node placements are A, A, A, A, A, B, C, D; second-node placements are E, F, G, H, I, J, K, L. MPS has thirty engines. Its first-node placements are A, A, AB, AB, AC, AC, AD, AD; second-node placements are AE, AF, AG, AH, AI, AJ, AK, AL. Concatenated letters denote independent colocated engines. Thus, A occupies all sixteen GPUs, B/C/D each have two replicas, and the others have one. SUN uses sixteen shared decode workers, each serving A–L, and includes cross-node KV transfer.

MPS uses full-workload demand knowledge, resident workers, least-active replica routing, and a 100% active-thread cap for each client, without loading or reconfiguration costs. Replicating A across all sixteen decode GPUs gives the hot service access to available capacity throughout the pool; the 100% active-thread cap does not reserve a fixed SM share for each engine. We call this deployment *workload-informed MPS*.

Table 4: Twelve-service evaluation on 32 H100 GPUs. All systems complete the same 5,650 requests and output-token volume. Throughput is total output tok/s.

System	TPOT (ms) ↓		Completion ↓	Throughput ↑
	Mean	p95	(s)	(tok/s)
Popularity-aware isolated	**14.16**	**16.80**	388.8	3,598.4
Workload-informed MPS	43.56	51.71	348.9	4,009.9
SUN	18.96	20.62	**263.9**	**5,301.2**

Results and interpretation. All systems complete the workload without failure (Table 4). MPS improves throughput by 11.4% over popularity-aware isolated allocation, confirming the benefit of physical colocation. SUN increases throughput by a further 32.2% over MPS and reduces mean/p95 TPOT by 56.5%/60.1%. SUN's throughput advantage over MPS rises from 23.6% in the small-scale comparison to 32.2% here, showing that longer outputs and higher concurrency do not eliminate the benefit of shared decoding. Relative to isolated allocation, SUN delivers 47.3% higher throughput, with higher p95 TPOT (20.62 versus 16.80 ms).

The advantage persists with long outputs and substantial concurrency: MPS sustains running-batch p95/max of 20/26, queue p95/max of 25/29, and p95 KV occupancy of 99.23%. Longer outputs keep requests active, but independent model-specific engines continue to form separate batches. High aggregate concurrency therefore does not guarantee equally dense batching for every service under skewed demand.

To explain this distinction, let λ be the aggregate request arrival rate, p_i the demand share of service i, and r_i its replica count. Under balanced routing, each of SUN's sixteen workers receives approximately $\lambda/16$ arrivals eligible for a mixed-model batch, whereas each MPS engine for service i receives $p_i\lambda/r_i$ arrivals for its own batch. For the hot service, $p_i = 23/45$ and $r_i = 16$, so SUN has approximately $1.96\times$ the batch-forming arrival rate per worker. This comparison describes batching opportunities, rather than a measured batch-size or throughput ratio. Longer outputs and higher aggregate concurrency therefore do not remove the model-specific batching boundary: placement and replication redistribute capacity, whereas SUN also combines requests across services into a shared decode batch.

C.2 Measured Training Cost

SUN targets task-specific models fine-tuned from a shared backbone. In this setting, the relevant cost is task adaptation, rather than large-scale continual pretraining. We measure one epoch of Qwen3-14B prefill-only tuning on one eight-GPU node: math, tool calling, and coding require 3.0, 4.5, and 5.9 GPU-hours, respectively, totaling 13.4 GPU-hours for the three models. This modest cost is incurred once per model version and is paid again only when that model is updated.

Table 5: One-time adaptation cost and recurring serving resource requirements. Tuning costs cover three Qwen3-14B specialists; serving GPU counts include prefill and decode in the eight-model deployment of Table 2 at 8 RPS.

Cost	When incurred	Measured requirement
Prefill-only tuning	Once per model version	3.0–5.9 GPU-hours per model
Serving GPU allocation	Every hour of operation	32 $\rightarrow$ 24 GPUs (25% fewer)

Table 5 highlights the difference in how these costs accumulate. In the consolidated deployment, SUN releases eight GPUs while retaining 98.7% of baseline output throughput. This saves eight GPU-hours per operating hour, or 192 GPU-hours over 24 hours at that allocation. The adaptation cost is paid once, whereas these savings recur for as long as the consolidated deployment operates, allowing continued serving to amortize the upfront tuning cost.

D Limitations and Future Work

SUN requires a compatible backbone, tokenizer, attention configuration, and KV interface (Section 3.2.3); sharing across heterogeneous architectures or model sizes requires additional compatibility mechanisms. Prefill remains task-specific, so skewed prompt-processing demand can still create prefill bottlenecks. Combining shared decoding with adaptive prefill scheduling and evaluating larger model fleets are directions for future work.