

Serving DeepSeek-V4.1-Flash (763B) on Two 96 GB GPUs: A Measurement Study of Host-Memory Expert Offload in vLLM for Coding Agents

Inferencekey

Artifact version of 2026-10-07

Abstract

DeepSeek-V4.1-Flash has 763 B parameters (475.2 GiB of tensors). A third-party serving stack (vLLM, the vllm-exl3 plugin and a recipe published by Diffbot) runs a 3-bit EXL3 version on two 96 GB RTX PRO 6000 GPUs by keeping part of the routed experts, the Engram tables and the token embedding in pinned host memory, read over PCIe. Every model, kernel and placement algorithm in it is third-party; our contribution is an independent, reproducible characterization of what this host offload makes possible and what it costs. From safetensors headers we show that the pack needs 102.1 GiB of weights per GPU against 95.6 GiB available, and that the published plan brings this to 72.4 GiB; controlled boots confirm that without the offload the model does not load. With it, decode for a single request after a 256K-token prompt stays close to the short-prompt rate on the same pod (8.3 against 7.6 ms per token after a 4K-token prompt), and in a coding-agent experiment four simultaneous streams keep at least 40 output tokens/s each, for short prompts, a shared cached prefix of up to 256K tokens, independent cached 128K-token contexts and agent turns shaped by an observed usage profile (independent 256K-token contexts were run only up to three streams, all above that rate). That concurrency, not monthly token volume, bounds how many developer seats one server supports: about 19 under the observed profile, close to the price of 15.1 subscriptions. On fixed subsets of GSM8K, IFEval and HumanEval the stack matches DeepSeek’s own API within the uncertainty. We also find that upstream vLLM’s UVA offload cannot serve the model without Diffbot’s hooks, that vLLM’s KV-cache estimate is conservative for this model, and that the same configuration ran markedly slower on a pod with a different NUMA layout. Each experiment’s measurements come from a single boot of a rented pod, with no measurement repeated across boots and no all-in-VRAM baseline; we re-analyze earlier single runs separately as exploratory, and every number in the paper is generated from its data file.

1 Introduction

Mixture-of-experts (MoE) models keep most of their parameters in routed experts, of which each token uses only a few. DeepSeek-V4.1-Flash [22, 27] routes each token to 6 of 384 experts in each of its 40 layers and adds two Engram lookup tables [14]. Its released checkpoint holds 268.95 GiB of routed experts and 188.83 GiB of Engram tables (Table 5). Serving it on hardware whose GPU memory is smaller than the model needs two ingredients: aggressive weight quantization and a place outside the GPU for whatever still does not fit.

A community stack does exactly this on two RTX PRO 6000 Blackwell GPUs with 96 GB each: vLLM [53] with the vllm-exl3 plugin by Victor Cruz [19], routed experts quantized to 3 bits per weight in the EXL3 format of ExLlamaV3 [18, 81], and a serving recipe by Diffbot [29] that keeps a profile-chosen *cold* subset of the experts, the Engram tables and the token embedding in pinned host memory, read by the GPU kernels over PCIe. Earlier measurements with this stack (ours, single runs, Section 8.1) suggest it serves usable decode speeds to a handful of concurrent users.

This paper asks a narrower and harder question than “does it run”:

What exactly does the host-memory offload code make possible that was not possible before, and at what performance cost?

Answering it requires separating three things that are easy to conflate: who wrote which part of the stack, what the hardware can hold without the offload, and how latency, throughput, concurrency, memory traffic, cost and quality change when the offload is switched on, scaled up, or replaced by upstream mechanisms.

What we did.

1. **Attribution audit** (Section 3). We fetched every component at its pinned commit, applied every patch to its upstream, compared the vLLM patches with their upstream pull requests line by line, and grepped

finding	measurement	section
offload is what makes it run	without offload the pack needs 102.1 GiB per GPU against 95.6 GiB available; both no-offload boots fail with GPU out of memory, the gen30 plan boots and serves (E2)	5, 8.2
long context costs pre-fill, not decode	median TPOT of one request 8.3ms after a 256K-token prompt against 7.6ms after 4K tokens, both on pod 2; a fresh 128K prompt takes 18.33s to its first token (E12)	8.2
coding-agent capacity	4 simultaneous agent-turn streams at 40 output tokens/s with a 10% margin, on a pod running nothing else (E13, one repetition)	10
seats per pod (derived)	19 seats per pod at 1% overflow against a break-even of 15.1 subscriptions per pod-month	10
quality against the vendor API	GSM8K: same outcome on every item; IFEval +0.013 (-0.013 to +0.047), HumanEval -0.018 (-0.049 to +0.012); intervals include zero (E10, E11)	11
upstream UVA cannot replace the hooks the pod matters	vLLM’s UVA offload of the same experts fails at initialization with KV block 64 and on the first request with 128 (E3)	8.2
	same workload on a second host: median TPOT 1.17 to 1.51 times higher (E3, one repetition per pod)	8.2
KV estimate is conservative	vLLM’s estimate is 2.38 full-length requests; five independent 128K and three 256K contexts used at most 48% and 56% of the KV cache without preemption (E13)	10

Table 1: Key results. Pod measurements come from single boots on 2x RTX PRO 6000 pods (one boot per experiment), thinking off; the first row also rests on the static header accounting, the quality reference (E11) is DeepSeek’s API and its prompts are benchmark subsets, and the seats row is derived from E13 under the usage assumptions of Section 10.

all trees for the offload features. All models, kernels, patches and the placement heuristic are third-party. The per-expert host placement is Diffbot’s; zero-copy (UVA) weight offload with parameter selection already exists in upstream vLLM at the pinned commit. Our own share is orchestration, configuration and measurement.

2. **Static accounting** (Section 5). From the safetensors headers of the served pack and of the original checkpoint, fetched with HTTP range requests, we account every byte per tensor class and placement. This shows without a GPU what cannot fit in VRAM.
3. **Exploratory re-analysis** (Section 8.1). We import earlier runs verbatim with hashes, recompute their statistics, and label everything derived from them as exploratory.
4. **A controlled protocol and the part of it we ran** (Sections 6 to 11): an ablation ladder from upstream vLLM to the full stack, a benchmark harness, memory and PCIe instrumentation, a pre-registered cost model and a quality evaluation. Our measurement campaign, three pod sessions on the same GPU model, ran the feasibility boots, the main benchmark, a reduced host-share comparison that includes upstream vLLM’s own UVA offload, reduced quality runs, a long-context experiment and the coding-agent experiment (Section 8.2). Section 6.8 lists the experiments that were designed but not run, and Section 13 what the campaign does not answer.
5. **A coding-agent case study** (Section 10): a workload and cost model that keeps simultaneous streams at 40 output tokens/s apart from developer seats by monthly generation volume, with a Binomial model of concurrent demand and a comparison with a per-seat subscription, measured by a dedicated experiment (E13) on six context-sharing regimes, and a note on what self-hosting means for digital sovereignty and data protection.

Table 1 summarizes the main measured findings; each row points to the section with the data and its caveats.

What this paper does not claim. It does not claim a new algorithm or system, and it does not claim that the stack preserves model quality in general: our own quality evidence covers three small task subsets with thinking off (Section 11). Every measured number in the paper is printed from a data file through a macro that fails the build when the file is missing (Section 14).

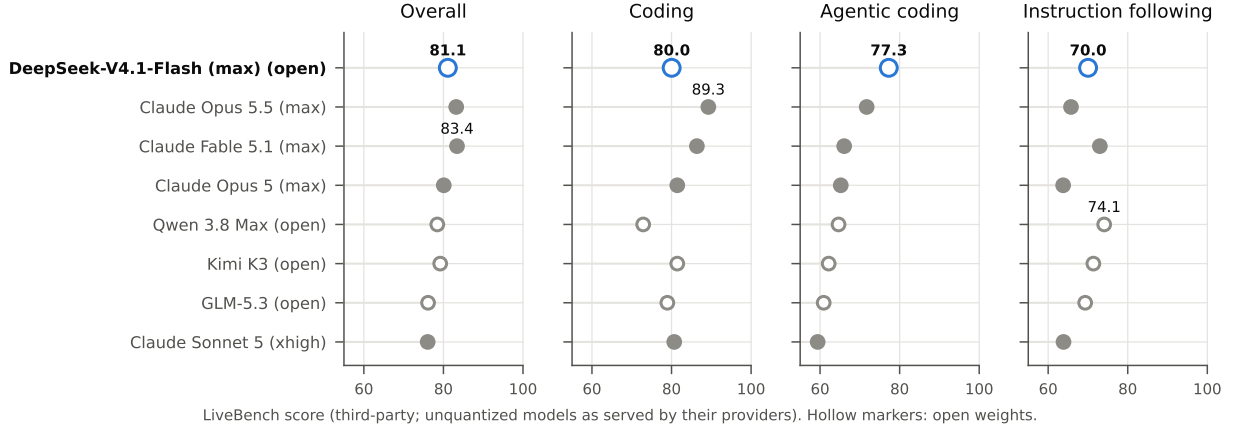


Figure 1: Third-party context, not a measurement of this paper: LiveBench scores (question set 2026-06-25, downloaded 2026-10-07) of the unquantized models as served by their providers at the named reasoning effort. DeepSeek-V4.1-Flash (highlighted) is the maximum-effort, thinking-on configuration; the 3-bit, thinking-off stack studied here was not evaluated on LiveBench. Hollow markers are open-weight models; values are labelled for DeepSeek-V4.1-Flash and the best model of each panel. Source: data/static/livebench/.

model	Overall	Coding	Agentic coding	Instruction following
DeepSeek-V4.1-Flash (max) °	81.1	80.0	77.3	70.0
Claude Opus 5.5 (max)	83.2	89.3	71.7	65.7
Claude Fable 5.1 (max)	83.4	86.4	66.1	73.0
Claude Opus 5 (max)	80.1	81.4	65.2	63.8
Qwen 3.8 Max °	78.5	72.9	64.6	74.1
Kimi K3 °	79.2	81.4	62.2	71.4
GLM-5.3 °	76.1	79.0	60.9	69.3
Claude Sonnet 5 (xhigh)	76.0	80.7	59.4	63.9

Table 2: Third-party context: LiveBench scores of the unquantized models as served by their providers at the named reasoning effort (question set 2026-06-25, data downloaded 2026-10-07, data/static/livebench/). ° open weights. The DeepSeek-V4.1-Flash row is maximum effort with thinking on; the 3-bit stack measured in this paper (thinking off) was not evaluated on LiveBench.

2 Background

The model. DeepSeek-V4.1-Flash [27] is a sparse MoE transformer in the DeepSeek line of DeepSeek-MoE [20], MLA attention [23] and multi-token prediction [24]. The configuration of the served pack (data/static/hub/) has 40 layers of hidden size 5,120; each layer has 384 routed experts of intermediate size 2,304 plus 1 shared expert, and routes each token to 6 routed experts. Attention is sparse, with a sliding window of 128 tokens and an indexer that selects 512 positions, and the context window is 1,048,576 tokens. Two layers (1, 14) carry Engram tables, large embedding tables indexed by hashed n-grams that add conditional memory by lookup [14]. A DSpark drafter provides speculative decoding [15, 55]; its three layers have their own routed experts. The released checkpoint stores routed experts in FP4 and dense weights and Engram tables in FP8 (Table 5).

Why this model. On LiveBench [85], a benchmark that refreshes its questions to limit contamination, the unquantized model at maximum reasoning effort, as served by DeepSeek, scores 81.1 overall against 83.2 for Claude Opus 5.5 and 83.4 for Claude Fable 5.1 at maximum effort, and on agentic coding it ranks first of the 69 models listed, at 77.3 against 71.7 for the next (Figure 1, Table 2; leaderboard data downloaded on 2026-10-07 and stored with hashes in data/static/livebench/ [63]). These are third-party scores of a different configuration from the one we serve: we run a 3-bit EXL3 pack with thinking disabled, which was not evaluated on LiveBench. They explain why the model is worth serving on modest hardware, especially for coding agents (Section 10); they are not results of this paper, and our own quality evidence is limited to the comparison with DeepSeek’s API in Section 11.

EXL3. EXL3 is the weight format of ExLlamaV3 [81], a trellis-coded quantization in the family of QTIP [80]: weights are incoherence-processed with Hadamard transforms and encoded as paths of a trellis with a small codebook, so that a GPU kernel can decode tiles on the fly during the matrix product. The served pack uses 3-bit codes for the routed experts [18] and keeps the non-routed weights in DeepSeek’s FP8 format. A routed expert occupies 12.70 MiB in the pack against 17.93 MiB in the original checkpoint (`data/static/weights/`).

Serving with vLLM. vLLM [53] manages the KV cache in pages, batches requests continuously and splits long prompts into chunks that share iterations with decode steps [2]. It caches prompt prefixes so that requests sharing a prefix skip its prefill [83, 96]. Prefill (compute-bound, one pass over the prompt) and decode (one token per step per request, memory-bound) stress the hardware differently, which is why we measure them separately [99].

Zero-copy host memory. CUDA can register host memory as pinned and map it into the device address space (unified virtual addressing). A kernel that dereferences such a pointer reads the data over PCIe at the time of use, with no explicit copy and no device-side cache [65]. The bandwidth of such reads is bounded by the PCIe link and degrades with small, scattered accesses. Both the Engram tables (in upstream vLLM) and the cold experts (in the Diffbot patch) are served this way in the measured stack. Zero-copy reads do not appear as copy operations in a profiler; they show up in the PCIe throughput counters of NVML [67] and in GPU-metrics sampling of Nsight Systems [66], which is what our sampler and protocol record.

3 Attribution: Who Wrote What, and What Is Ours

We did not assume our modification was novel, or even that there was one. Before designing any experiment we audited the stack as deployed in the exploratory runs (`audit/ATTRIBUTION.md`, evidence under `audit/evidence/`).

Method. `make audit` fetches vLLM at the pinned nightly commit (a blobless sparse clone), `vllm-exl3` at `d3cfd394`, ExLlamaV3 at `5be88657`, the `sfnz` recipe [72] at `460ac89c` and Diffbot’s recipe from the Hugging Face model repository at revision `a1cf1bd2`, without weights. It then (i) hashes every recipe file against the Hub and against the copy the exploratory runs used (identical, 95 of 95 files); (ii) applies every patch to its pinned upstream (all apply, in the recipe’s order); (iii) compares the four vLLM pull-request patches with the upstream pull requests line by line (every non-test line matches); (iv) greps every tree, before and after patching, for the markers of each feature and reads the code at the hits; (v) diffs the run scripts of the exploratory runs against the recipe; and (vi) dates each recipe part with the Hub’s commit history.

Findings. Table 3 summarizes the features this paper measures, and Figure 2 shows the same split by layer of the stack. The foundation of the EXL3 side is Victor Cruz’s `vllm-exl3` plugin [19]: it brings EXL3 routed experts into vLLM (the routed-expert EXL3 path, its pointer tables, the fat-expert path and the drafter’s experts in EXL3), building on ExLlamaV3’s kernels and on earlier work by Mia’s AI Lab. Diffbot’s host-expert patch is a patch to this plugin, so none of the configurations we measure would run without it. Three points matter for the paper’s claims. First, the per-expert host placement (a plan file listing cold experts per layer, kept in exact-size registered host memory behind a device alias), the split launch that runs VRAM experts on the main stream and host experts on a side stream with a bounded number of SMs, and the kernels that make this fast are Diffbot’s, added by its patch to `vllm-exl3`; none of it exists in `vllm-exl3` at the pin or in the recipes it derives from. Second, the mechanism underneath is not new inside the stack: upstream vLLM at the pinned commit already offers zero-copy UVA weight offload with selection by parameter name (`--offload-backend uva --cpu-offload-gb --cpu-offload-params`), at the granularity of whole parameter tensors, and the plugin’s author, Victor Cruz, documented an experimental UVA placement of EXL3 expert payloads two days after the pinned commit and before Diffbot’s publication (`audit/evidence/vllm_upstream_offload.txt`). Engram tables in pinned host memory read over UVA are upstream vLLM; their `int4` format and lookup kernel are Diffbot’s. Third, part of Diffbot’s `sm_120` patch is identical to an open upstream pull request by another author, which Diffbot’s notices credit only in part.

Consequences for this paper. The exploratory runs used Diffbot’s recipe unmodified: our serve script is the recipe’s with docker removed and every offload and VRAM setting at its default, plus a larger maximum batch and CUDA graph sizes capped below the size at which the plugin’s fat-expert path performs a host synchronization that graph capture forbids (`audit/evidence/fat_any_item.txt`). “Our code” is therefore the third-party stack as integrated, configured and measured by us. The contribution of this paper is a

feature	author	where	class
DeepSeek-V4.1 graph, DSpark, Engram UVA offload, selective UVA weight offload	vLLM project	vLLM 36768d1	implementation
EXL3 format and <code>exl3_moe</code> kernel	turboderp	ExLlamaV3 5be88657	implementation
EXL3 support in vLLM	Victor Cruz (vcruz305), from Mia’s AI Lab work	vllm-exl3 d3cfd394	implementation
EXL3 3-bit routed experts	coolbho3k	Hub checkpoint	artifact
per-expert host plan, split launch, side-stream SM budget plan selection from routing counts	Diffbot	<code>exl3-thor.patch</code>	algorithmic + implementation
	Diffbot	recipe plans/	heuristic
xmoe decode and decode-once prefill kernels	Diffbot (from ExLlamaV3)	recipe kernels/	implementation
int4 Engram format and lookup; host embedding; FP8 trims	Diffbot	recipe patches and hooks	implementation
sm_120 fixes	Diffbot and zhao6300 (PR #57028)	<code>sm120-thor-fixes.diff</code>	implementation
four upstream PR patches	ivanium, bluemelov1, bojiang-li, gcanlin	recipe patches/vllm/	implementation
runtime hooks	sfxnz, edited by Diffbot	<code>sitecustomize.py</code>	integration
build without docker, serve settings (≤ 64 sequences, graph sizes ≤ 256)	inferencekey (us)	exploratory scripts	configuration
load generator, samplers, this harness and analysis	inferencekey (us)	this repository	measurement

Table 3: Authorship of the features measured in this paper, from the audit (`audit/ATTRIBUTION.md`, which gives file:line evidence for every row). “Class” follows the audit’s classification: algorithmic contribution, implementation, integration, configuration, or measurement.

reproducible characterization of that stack: an audit with evidence, a byte-level account of what fits where, an ablation design that separates the upstream mechanism from Diffbot’s refinement, a measurement harness that fixes the weaknesses of the earlier runs, a cost and quality protocol, and an honest report of what is and is not yet measured. Where the paper says “the offload” it means Diffbot’s patch on top of upstream vLLM and vllm-exl3.

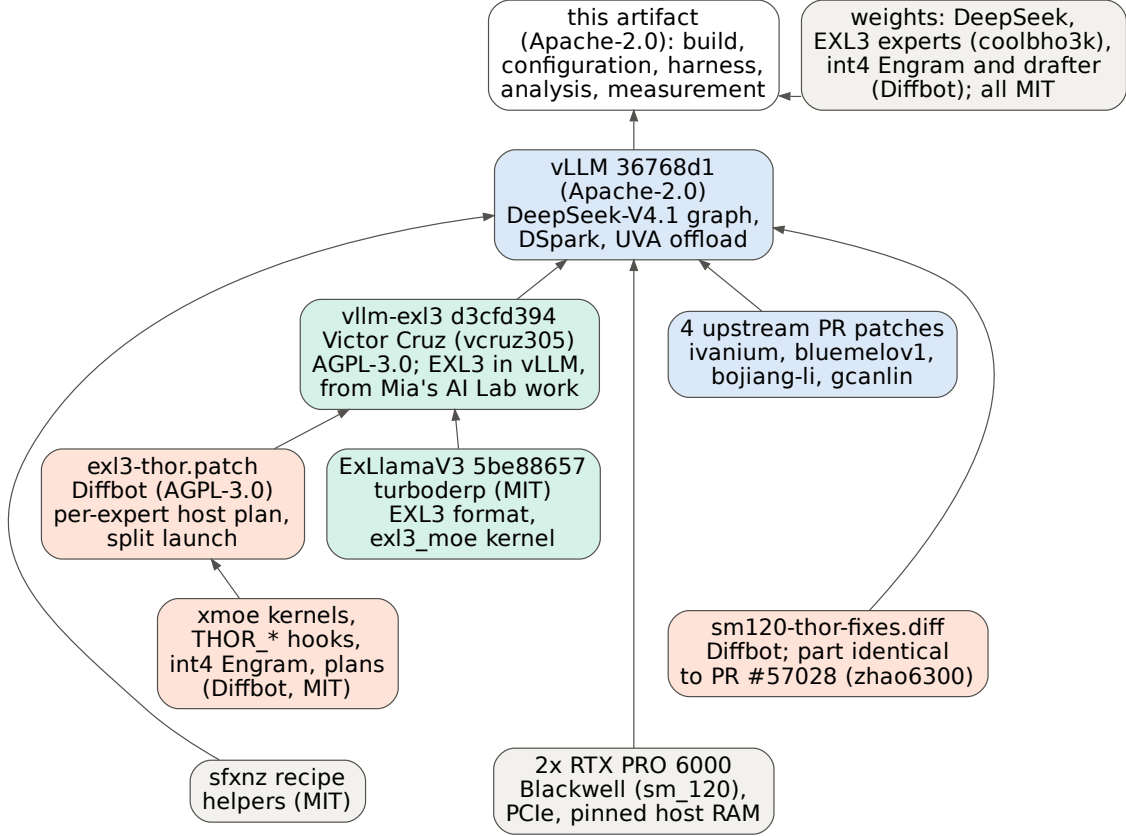


Figure 2: Layers of the measured stack and their authors (from `audit/ATTRIBUTION.md`; rendered from `diagrams/stack_provenance.dot`). Orange: Diffbot’s recipe; blue: vLLM and upstream pull requests; green: EXL3 format and plugin.

4 Related Work

Serving systems and batching. The evaluated stack runs on vLLM [53, 82], whose PagedAttention manages the KV cache in fixed-size blocks, in the spirit of virtual memory paging, so that memory waste is near zero and blocks can be shared within and across requests. Orca [92] introduced iteration-level scheduling, in which requests join and leave the running batch between model iterations instead of between whole batches. Sarathi-Serve [2] splits long prompts into chunked prefills and schedules them without stalling ongoing decodes. DistServe [99] and Splitwise [68] go further and place the prefill and decode phases on different GPUs or machines, optimizing each phase for its own latency target (time to first token, TTFT, and time per output token, TPOT, in the terminology of DistServe). SGLang [96] pairs a frontend language with a runtime that reuses KV cache through RadixAttention, and FlashInfer [40, 90] provides attention kernels with composable block-sparse KV formats and a scheduler compatible with CUDA graphs. These mechanisms define the serving behavior on top of which host-resident experts add PCIe traffic.

MoE offloading and CPU-GPU hybrid execution. Sparse Mixture-of-Experts layers [39, 54, 73] activate a few experts per token, so most expert weights are idle at any moment, which makes them natural candidates for slower memory. DeepSpeed-MoE [69] serves MoE models from aggregate GPU memory with expert parallelism and does not offload. The offloading literature can be organized along four axes: where cold weights live, whether the CPU computes, how host-resident weights reach the GPU, and whether placement is static or adapts at run time. Table 4 summarizes these axes for the systems we discuss.

A first family keeps experts in host memory and *copies* them into a GPU-resident expert cache, hiding the transfer with prediction. Eliseev and Mazur [36] keep an LRU cache of experts per layer and load experts speculatively by applying the next layer’s gate to the current hidden states, combined with mixed HQQ quantization. MoE-Infinity [89] traces request-level expert activation and uses it to drive replacement and

prefetching of its expert cache. Pre-gated MoE [48] changes the model itself: a trained pre-gate selects the experts of the next block one block ahead, so that only those are prefetched. SiDA [30] trains an offline hash function that predicts expert activation and places experts on the GPU ahead of each batch. ProMoE [75] predicts later expert usage from intermediate results and fetches experts proactively, AdapMoE [97] adds sensitivity-based adaptive gating and a per-layer cache allocation, ExpertFlow [44] predicts routing paths for all layers with a learned predictor and schedules tokens with similar routes together, and Fate [37] uses a cross-layer gate for prefetching with a cache that favors shallow layers and quantized cached experts. SwapMoE [52] and EdgeMoE [91] target edge devices, respectively by maintaining a small dynamic set of important experts in memory and by preloading predicted experts from storage with per-expert bit widths.

A second family is throughput oriented and *streams* weights layer by layer, amortizing each transfer over a large batch. FlexGen [74] and ZeRO-Inference, the heterogeneous component of DeepSpeed-Inference [3], do this for dense models, FlexGen with a linear program that splits weights, activations and KV cache across GPU, CPU and disk. MoE-Lightning [9] pipelines paged weight transfers with CPU attention under a performance model, MoE-Gen [88] accumulates tokens in host memory to launch large module-level GPU batches, Klotski [38] overlaps expert loading with multi-batch computation using a correlation-aware prefetcher, and MoE-Lens [94] derives its design from a holistic performance model of CPU memory capacity and GPU compute.

A third family *computes* cold experts on the CPU and moves activations instead of weights. Fiddler [50] places frequently used experts on the GPU from offline popularity profiling and, for the remaining experts, chooses with a latency model between copying weights to the GPU and running the expert on the CPU. KTransformers [12] keeps routed experts persistently in CPU memory and runs them with arithmetic-intensity-aware CPU kernels and NUMA-aware placement, keeps attention and shared experts on the GPU, and uses Expert Deferral to overlap part of the expert work with the next layer’s attention. HybriMoE [98], DAOP [95] and DALI [101] schedule experts dynamically between CPU and GPU; HybriMoE and DALI explicitly argue that fixed or static expert assignments are inefficient because activation patterns vary. eMoE [78] combines learned expert prediction with SLO-aware request scheduling.

Closest to the access pattern we study is DAK [61], which lets GPU kernels read weights and KV cache directly from host memory without staging them in device memory, with static per-operation offload ratios from an offline sweep; its abstract reports that direct access outperforms prefetching, but it targets dense models. CUDA exposes the underlying mechanism as mapped, page-locked host memory in a unified virtual address space [65]. We found no peer-reviewed MoE system in our search whose GPU kernels execute host-resident expert weights in place; the only instance we found is the third-party software we evaluate [19, 29].

Weight quantization and trellis codes. Post-training weight quantization ranges from scalar methods such as GPTQ [42], which uses approximate second-order information, and AWQ [60], which protects salient channels identified from activation statistics, to methods that also quantize activations, such as LLM.int8() [28] with its mixed-precision outlier decomposition and SmoothQuant [87], which migrates quantization difficulty from activations to weights. At three bits and below, incoherence processing and vector codes dominate: QuIP [10] introduced incoherence processing with random orthogonal transforms, QuIP# [79] combined the randomized Hadamard transform with E_8 lattice codebooks, and AQLM [35] adapted multi-codebook additive quantization. QTIP [80] replaces vector quantization with trellis-coded quantization, whose stateful decoder separates codebook size from bitrate and effective dimension, and designs a hardware-efficient bitshift trellis. ExLlamaV3 [81] describes its EXL3 format as a streamlined variant of QTIP; the evaluated plugin dispatches EXL3 routed experts inside vLLM [19]. For MoE models specifically, MoQE [51] reports that expert layers are much more robust to quantization than dense feed-forward layers, QMoE [41] compresses trillion-parameter MoE models below one bit per parameter with custom decoding kernels, MxMoE [31] allocates mixed precision using both quantization sensitivity and expert activation frequency, MiLo [47] adds low-rank compensators to extremely quantized MoE models, and QuantMoE-Bench [56] studies MoE structure-aware quantization heuristics. Quantization reduces the bytes that host-resident experts move over PCIe, so it interacts directly with the offloading choices above.

Speculative decoding. Speculative decoding [11, 55] lets a cheap drafter propose tokens that the target model verifies in parallel, preserving the target distribution; blockwise parallel decoding [76] is an earlier form of draft and verify. Later work removes the separate draft model or improves the drafts: Medusa [8] adds decoding heads verified with tree attention, SpecInfer [64] verifies token trees, and EAGLE [57, 58, 59] drafts at the feature level, with dynamic draft trees and, in EAGLE-3, direct token prediction with multi-layer feature fusion. Gloeckle et al. [43] train with multi-token prediction heads, and DeepSeek-V3 uses a multi-token prediction training objective [24]. Under batching the benefit is not automatic: TurboSpec [62] observes that naive speculation can degrade serving performance and adapts speculation length to predicted goodput, while

MagicDec [71] shows speedups at large batch sizes for moderate to long sequences. A survey is given by Xia et al. [86]. DeepSeek-V4.1-Flash replaces the MTP module with DSpark [27], a semi-autoregressive drafter with confidence-scheduled verification whose abstract reports per-user generation speedups of 60 to 85 percent at matched throughput over an MTP-1 production baseline in DeepSeek’s own serving system [15]. In the evaluated stack, part of the drafter’s experts is also host resident [29], so drafting itself generates PCIe traffic.

Prefix caching and long context. Automatic prefix caching in vLLM [83] and RadixAttention in SGLang [96] reuse the KV cache of shared prefixes, which matters for multi-turn agentic sessions where most of each prompt repeats. Chunked prefill [2] bounds the latency impact of long prompts on concurrent decodes. Long-context capability is commonly assessed with LongBench [6], a bilingual multitask benchmark, and RULER [46], a synthetic benchmark with configurable length and task complexity.

DeepSeek architecture. DeepSeekMoE [20] introduced fine-grained expert segmentation and shared expert isolation, and DeepSeek-V3 added auxiliary-loss-free load balancing [24, 84]. DeepSeek-V2 [23] introduced Multi-head Latent Attention (MLA), which compresses the KV cache into a latent vector. Sparse attention followed with Native Sparse Attention [93] and DeepSeek Sparse Attention in DeepSeek-V3.2 [21, 25]; DeepSeek-V4 [26] combines Compressed Sparse Attention with Heavily Compressed Attention for one-million-token contexts. DeepSeek-V4.1-Flash [22, 27] has 552B backbone parameters, activates 16B parameters per token during decode and 8B during prefill, and reduces its global KV cache with cross-layer reuse in Compressed Sparse Attention 2 and FP4 KV caching. Its model card lists 384 routed experts and one shared expert per MoE layer, six routed experts active per token, and 196B parameters of Engram conditional memory accessed by token-based lookup. Engram [14] (a conference version names the module Deep Sparse Embedding [16]) modernizes N -gram embeddings for constant-time lookup and notes that deterministic addressing allows prefetching from host memory; the V4.1 report describes prefetching Engram embeddings from host memory via background RDMA. The evaluated stack instead keeps int4 Engram tables in pinned host memory and reads them over UVA inside CUDA graphs [29].

Evaluating quantized and offloaded models. Common task suites for quality include GSM8K [17], MATH [45], HumanEval [13], MBPP [5], LiveCodeBench [49] and IFEval [100], typically run through lm-evaluation-harness [77], whose maintainers document the reproducibility pitfalls of language model evaluation [7]. Matching accuracy is not sufficient evidence that a compressed model behaves like its reference: Dutta et al. [32] show that answers flip between correct and incorrect even when aggregate accuracy is unchanged, and propose KL divergence and flips as distance metrics. For DeepSeek-V4.1-Flash, AtomicChat published top-512 log-probability dumps and KL analyses for an NVFP4 quantization [4], and the EXL3 checkpoints we use report KL against the original model under that protocol [18, 29]. On the performance side, MLPerf Inference [70] prescribes rules for comparability across systems, Agrawal et al. [1] argue that TTFT and TPOT do not fully capture user-facing latency, and the tools and statistics relevant to our measurements are NVIDIA Nsight Systems and NVML [66, 67] and the bootstrap [33, 34].

Positioning of this work. The stack we evaluate is third-party software: the vllm-exl3 plugin by Victor Cruz (vcruz305) [19], which serves EXL3 routed experts in vLLM and offers an experimental mode in which packed expert weights are pinned in host RAM while the GPU remains the compute device, and a host-expert patch with kernels by Diffbot [29], built on vLLM [53] and ExLlamaV3 [81]. According to its documentation, the Diffbot configuration keeps a static set of cold experts (4,608 of 15,360 routed experts in the plans it publishes, chosen as those least used in long multi-turn agentic coding sessions) together with the int4 Engram tables and the embedding table in pinned host memory. GPU kernels read these weights in place over PCIe, the cold experts run on a second CUDA stream with a bounded number of SMs while the VRAM experts run on the main stream, and there is neither CPU expert computation nor prefetching. Relative to Table 4, this design shares static, profile-derived placement with Fiddler [50] and KTransformers [12], but executes cold experts on the GPU rather than the CPU; it shares direct kernel access to host memory with DAK [61], which targets dense models, and with the UVA weight offload that upstream vLLM itself provides at the pinned commit, which offloads whole parameter tensors (for a fused MoE layer, all of its experts) until a per-GPU budget is reached (Section 3); and it differs from the cache and prefetch systems in that cold expert weights are read in place rather than copied into a device-side expert cache. Several recent papers argue that static expert placement is suboptimal [44, 98, 101]; our controlled experiments measure a static zero-copy placement on one hardware configuration at two host shares, without comparing against those systems on equal terms and without testing other placement policies (Section 6.8). We claim no new algorithm or system. Our contribution is a reproducible measurement study of what this existing stack makes possible on two RTX PRO 6000 Blackwell GPUs, and at what cost in latency, throughput, memory, PCIe bandwidth and quality.

System	Cold weights in	CPU computes	How weights reach the GPU	Placement	Quantization
<i>Copy into a GPU expert cache, hidden by prediction</i>					
Eliseev and Mazur [36]	host	no	copy, speculative prefetch	dynamic (LRU)	HQQ
MoE-Infinity [89]	host	no	copy, prefetch	dynamic (activation-aware)	n.s.
Pre-gated MoE [48]	host	no	copy, trained pre-gate	dynamic	n.s.
SiDA [30]	host	no	copy ahead of each batch	dynamic (offline-trained predictor)	no
ProMoE [75]	host	no	copy, proactive prefetch	dynamic (LRU)	INT4 in evaluation
AdapMoE [97]	host or SSD	no	copy, prefetch	dynamic (adaptive cache)	4-bit
ExpertFlow [44]	host	no	copy, predictive	dynamic (predictive cache)	no
Fate [37]	host	no	copy, prefetch	dynamic (ARC)	INT4, INT2
<i>Stream weights, amortized over large batches</i>					
Klotski [38]	host, disk	no	copy, pipelined with batches	dynamic (correlation-aware)	4-bit, optional
MoE-Lightning [9]	host	attention only	copy, paged streaming	offline policy search	n.s.
MoE-Gen [88]	host	part of attention	copy, module batching	n.s.	n.s.
FlexGen [†] [74]	GPU, host, disk	optional attention	copy, layer streaming	offline LP	4-bit
ZeRO-Inference [†] [3]	host or NVMe	no	copy, layer prefetch	static	n.s.
<i>Compute cold experts on the CPU</i>					
Fiddler [50]	host	yes	per expert: copy or CPU compute	static (offline profiling)	no
KTransformers [12]	host (routed experts)	yes	none: activations move	static	CPU block-wise
HybriMoE [98]	host	yes	copy or CPU compute	dynamic	4-bit
DAOP [95]	host	yes	activations move to the CPU	calibrated, adjusted per sequence	n.s.
DALI [101]	host	yes	copy or CPU compute	dynamic	n.s.
<i>Read host memory in place from GPU kernels</i>					
DAK [†] [61]	host	no	in place, direct kernel access	static (offline sweep)	n.s.
Evaluated stack [19, 29]	pinned host	no	in place, UVA zero-copy	static (usage profile)	EXL3, int4 Engram

Table 4: Qualitative positioning of offloading systems, grouped by how cold weights reach the computation and restricted to facts stated in each paper (for the evaluated stack, in its documentation). “Host” is system memory (DRAM); “copy” means host-resident weights are transferred into device memory before use, “in place” that GPU kernels read host memory directly. [†] dense model; all others are MoE. n.s.: not stated as part of the design in the text we read.

5 System and Memory Architecture

Placement. Figure 3 shows where the served pack lives under the published plans. Each GPU holds its tensor-parallel shard of the attention and indexer weights, shared experts, routers, the lm head, the KV cache, CUDA graphs and the *hot* routed experts. Pinned host memory holds the *cold* routed experts listed in the plan (cold target experts plus drafter experts), the two int4 Engram tables, sharded by rank, and the token embedding. The plans published with the recipe all put 4,608 of the 15,360 target experts on the host (between 65 and 145 per layer, Figure 4); they differ in which experts, chosen from routing counts of different traffic. At each MoE call the patched plugin launches the VRAM experts on the main stream and the host experts on a second stream limited to a configurable number of SMs, so that the PCIe-bound reads of cold experts overlap with the VRAM work.

Byte accounting. Table 5 accounts every tensor of both checkpoints by class. It is computed from the safetensors headers, fetched with HTTP range requests (190,557 tensors, 309.3 GiB for the pack; no weights downloaded), and from explicit placement rules that mirror the documented configuration (`src/hostoffload/weights/split.py`). The rules are a model, so we check them against what the server reported at load in the exploratory run: the Engram bytes the server pinned (103.0 GiB in four shards of 25.75 GiB, one per table and rank) equal the Engram file bytes, and the per-layer counts of host experts that the server logged equal those of one published plan on every layer, which identifies the plan of each exploratory run (`gen30` for the concurrency sweep, `agent30` for the first run, an earlier single run whose folder is named `a100` after a sizing test but which ran on the same 2x RTX PRO 6000 type; the earlier report had left the plan ambiguous). File bytes are not runtime bytes: vLLM reported 76.78 GiB of weights per GPU, 4.37 GiB more than the plan’s VRAM file bytes per GPU, presumably from repacked and duplicated layouts. We did not attribute that difference per allocation: the controlled runs record per-process memory, but this analysis was not done.

What fits without the offload. Without a plan, the pack’s GPU-resident weights alone come to 204.1 GiB, or 102.1 GiB per GPU, more than the 95.6 GiB the driver reports per card before any KV cache, graph or activation memory. The original checkpoint needs 284.3 GiB outside its Engram tables and embedding. With the `gen30` plan the pack’s VRAM share drops to 72.4 GiB per GPU, while 163.6 GiB of file bytes move to the host (59.3 GiB of experts). The exploratory boot left 6,520 MiB of VRAM unused per GPU at peak; spending all of it on experts would move 1026 target experts back to VRAM, so with this KV budget and these graph sizes at least 23.3% of the target experts must stay on the host. This is an estimate calibrated on one exploratory boot and was not measured: E3 ran two host shares that both fit, and no run searched for the smallest share that boots. The byte accounting is tested (`tests/test_feasibility.py`), and experiment E2 confirms the infeasibility on the GPUs with the server’s own error messages (Section 8.2).

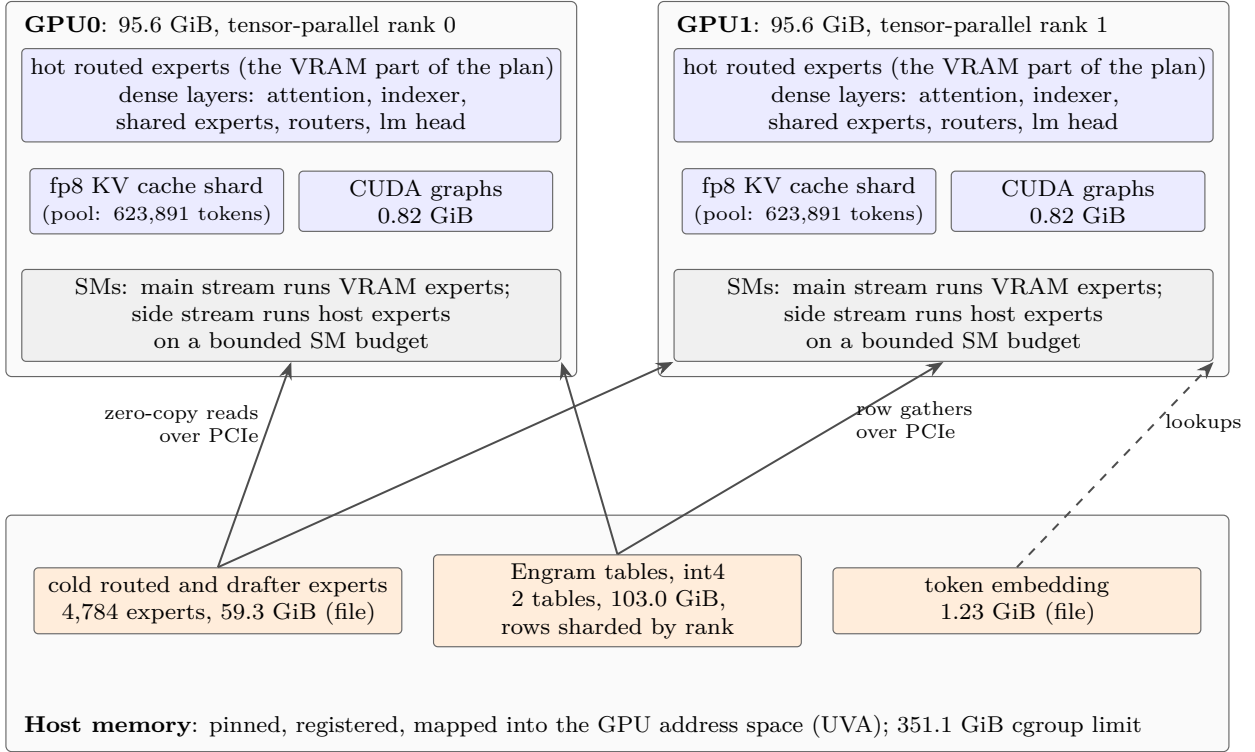


Figure 3: Memory architecture of the measured stack (two GPUs, tensor parallel). Blue: VRAM. Orange: pinned host memory read by GPU kernels through UVA aliases over PCIe. Sizes are checkpoint bytes from the safetensors headers (`data/static/weights/diffbot-a1cf1bd2/placement.json`) or values the server logged at load in the exploratory run (KV tokens, CUDA graph memory, Engram tables), all printed from data files.

tensor class	original	EXL3 pack	VRAM	host	other
routed experts (40 layers)	268.95	190.48	133.34	57.14	
Engram tables	188.83	103.00	0.00	103.00	
attention and indexer	4.80	4.80	4.80	0.00	
drafter experts	6.72	4.76	2.58	2.18	
shared experts	1.42	1.42	1.42	0.00	
token embedding	1.23	1.23	0.00	1.23	
lm head	1.23	1.23	1.23	0.00	
vision tower	0.90	0.90	0.00	0.00	not loaded
Engram projections	0.29	0.59	0.59	0.00	
drafter dense	0.56	0.56	0.56	0.00	
routers	0.15	0.15	0.15	0.00	
norms, hyper-connections	0.15	0.15	0.15	0.00	
total	475.24	309.28	144.81	163.56	0.90

Sources: `data/static/weights/deepseek-ai-2cba9e42/placement.json`,
`data/static/weights/diffbot-a1cf1bd2/placement.json`.

Table 5: Checkpoint bytes by tensor class (GiB, from safetensors headers) for DeepSeek’s original checkpoint and the served EXL3 pack, and where the pack’s bytes go under the `gen30` host-expert plan with the documented placement rules (text-only serving). File bytes, not runtime allocations. Static measurement.

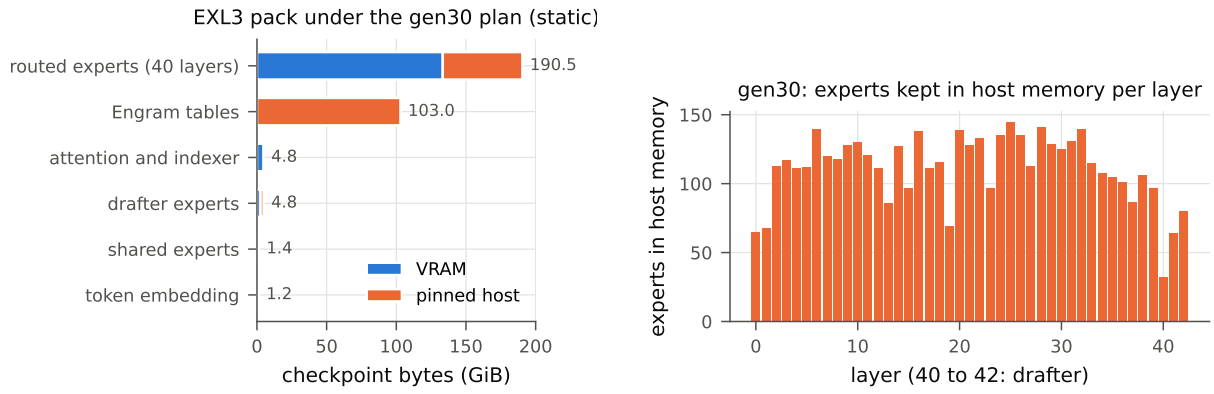


Figure 4: Static measurement. Left: checkpoint bytes per tensor class in VRAM and in pinned host memory under the **gen30** plan. Right: experts the plan keeps in host memory, per layer (layers 40 to 42 are the drafter). Source: `data/static/weights/diffbot-a1cf1bd2/placement.json`.

6 Methodology

6.1 Research questions

- RQ1 (feasibility).** On two 96 GB GPUs, which configurations of the model serve at all without host offload, and with which exact failure do the others stop?
- RQ2 (cost of offload).** How do prefill, decode, TTFT, TPOT and throughput change with the share of experts in host memory, and how much of that is PCIe traffic?
- RQ3 (what the refinement adds).** At equal host bytes, how does Diffbot’s per-expert hot/cold split compare with upstream vLLM’s own selective UVA offload, and how much depends on the placement policy and its match to the traffic?
- RQ4 (frontier).** What is the cost per token and per concurrent request under pre-registered service-level objectives, from concurrency 1 to 64, and how does it compare with an all-in-VRAM reference?
- RQ5 (quality).** How much accuracy does the 3-bit stack lose against the original checkpoint on reasoning, code, instruction following and long context?
- RQ6 (coding agents).** How many simultaneous coding-agent streams keep at least 40 output tokens/s each, and how many intermittently active developer seats can share one pod under an observed usage profile (Section 10)?

6.2 Ablation ladder

Each rung adds one layer on the same hardware and the same serving settings (`stack/variants/`, `experiments/experiments.yaml`):

- (a) **upstream:** vLLM at the pinned commit, unpatched, with DeepSeek’s original checkpoint. Expected to fail at load on two 96 GB GPUs (Section 5); E2 records the error. **upstream-uva:** the same with upstream vLLM’s selective UVA offload of the expert parameters, the closest upstream baseline that may fit.
- (b) **exl3-min:** plus the five vLLM patches the pack needs on `sm_120`, the unpatched `vllm-exl3` plugin and stock ExLlamaV3 kernels, with every expert in VRAM. Expected to fail at load; E2 records it.
- (c0) **exl3-uva:** (b) plus upstream vLLM’s selective UVA offload of the EXL3 expert parameters, sized to the same host bytes as a plan (`hostoffload.weights.plan bytes`).
- (c) **exl3-offload:** Diffbot’s patch, kernels and hooks; factors: the plan (host share 25 to 60% from routing counts of a calibration workload on a disjoint corpus half; the recipe’s three plans; a seeded random plan with the same per-layer sizes), the Engram format (int4 or FP8, if host memory allows), and the side-stream SM budget.
- (d) **DSpark on or off** on top of (c).

A reference rung, **upstream on 2x B300**, serves the original checkpoint fully in VRAM (E9, E11): it is the speed and quality baseline the offload has to be judged against, on different and more expensive hardware.

Rungs that ran. Our three pod sessions ran rung (b) and the offload build with an empty plan (both only to the load failure, E2), rung (c0) (it boots or fails, but never serves; Section 8.2), and rung (c) with DSpark on and two plans only: the recipe’s **gen30** plan and the union of its **gen30** and **agent30** cold sets. Rung (a) and its **upstream-uva** variant (both with the original checkpoint), the calibrated, random and FP8-Engram plans, the SM budget sweep, rung (d) and the B300 reference were not run (Section 6.8).

6.3 Benchmark protocol

The harness (`src/hostoffload/harness/`) replaces the exploratory load generator and fixes its known weaknesses (Table 6). The protocol below is the full design (`experiments/workloads/main.yaml`); each item ends with what our runs actually used (`experiments/workloads/lean_main.yaml`, `lean_probes.yaml`, `lean_subset.yaml`, `long_context.yaml` and `agent.yaml`, run by `experiments/runs/lean.sh`, `long.sh` and `agent.sh`).

- **Prompts** are cut to exact token counts with the server’s tokenizer endpoints from pinned corpora: the vLLM sources at the pinned commit (code) and four public-domain books (prose), both pinned by the sha256 of the assembled text (`experiments/corpora.yaml`). Every prompt of a run is unique except deliberately shared prefixes, and benchmark and calibration workloads use disjoint corpus halves. *In our runs*: as designed, but every measured prompt came from the code corpus; the calibration workload was not run.
- **Scenarios**: A, independent 4K-token code prompts; A-Poisson, the same as an open-loop Poisson stream at five rates; B, a 16K-token shared prefix (cached after one warm request) plus a question; P, 1K-token prose continuation, where speculation is expected to accept fewer tokens; L, fresh 128K-token prompts. All with 256 output tokens, temperature 0.6, top-p 0.95, per-request seeds, thinking off. *In our runs*: A and B only in E8; neither A-Poisson nor P ran. Long prompts were measured by the separate long-context experiment E12 (fresh prompts up to 260,000 tokens, decode after 128K and 256K-token prompts, and 128K and 256K-token shared prefixes), and E13 used 400-token replies (Section 10).
- **Arrivals**: closed loop with $N \in \{1, 2, 4, 8, 16, 32, 64\}$ users whose starts are staggered over 10 s (so no synchronized first wave), and Poisson open loop. A sweep stops when TTFT p95 exceeds 120 s or more than 5% of requests fail; the saturation point is recorded. *In our runs*: closed loop only; starts staggered over 5 s in the scenarios with independent short prompts or a shared prefix (E8, E3, the prefix scenarios of E12 and regimes A and B of E13), not in the long-prompt decode scenarios of E12 or regimes C and D of E13; in E8 and E3 scenario A two requests per user and at least eight per level, in the decode probes one per user and at least four. The minimum of two errors for the error rule applies to E12 and E13 only (session 2’s E3 rerun kept the first session’s rule to match its baseline), and the long-context and agent sweeps stop at a TTFT p95 of 900 s.
- **Prefill and decode separately**: prefill probes send unique prompts of 1K to 64K tokens with one output token, one at a time, so TTFT is prefill plus an empty queue; decode probes send 128-token prompts with 512 forced output tokens at each N , so TPOT is decode under batching. Server-side histograms of queue, prefill and decode time per request are differenced around every level. *In our runs*: prefill probes at 4K, 16K and 64K tokens (three requests each) and decode probes at $N \in \{1, 8, 32\}$, one repetition.
- **Repetitions**: three per load point within a boot, fresh prompts and seeds each time, and three cold boots at $N \in \{1, 8, 32\}$ for the across-boot component. Every request of every repetition is stored raw. *In our runs*: no cold-boot repetitions; each experiment’s measurements come from a single boot (the `gen30` configuration was booted on three pods, but no measurement was repeated across boots). E8 has two repetitions per level in B and, after an early stop (Section 8.2), in A only at $N = 1$; E3, E12 and E13 have one. Every request is stored raw.

6.4 Metrics and statistics

TTFT is the arrival of the first streamed token minus the send time. TPOT is the time from the first to the last token divided by the number of tokens after the first; per-request decode rate is its inverse. Per-token latency weights each chunk’s inter-arrival time by the tokens it carried. E2E is the end of the stream minus the send time. Aggregate output throughput counts the tokens of requests that finished inside a window from the first send (plus a warm-up) to the last send. We report p50, p95 and p99 with linear interpolation, always with n ; with fewer than 20 requests a p95 is close to the maximum. Uncertainty comes from a bootstrap that resamples whole repetitions [34], because requests of one repetition share server state, and from the across-repetition coefficient of variation; a one-way variance decomposition separates between- and within-repetition variance (`src/hostoffload/analysis/stats.py`).

6.5 Memory and PCIe accounting

For every run we record: per-GPU memory over time (NVML); the cgroup’s anonymous, shared and file memory; per-process resident, anonymous, shared, locked and pinned memory; the largest mappings of each server process at start; and the placement the server logs at load (host experts per layer, Engram bytes per rank, weight, KV and graph memory). The static split (Table 5) gives file bytes per tensor class in VRAM and host memory; the runtime records give allocations. PCIe receive and transmit rates per GPU come from NVML at 0.5 s and from `nvidia-smi dmon`, cross-checked by one Nsight Systems capture with GPU metrics, and calibrated by a script that drives a known volume of copy and zero-copy traffic and compares it with the counters (`experiments/tools/pcie_calibrate.py`). The analysis correlates PCIe receive rate with per-step

weakness of the exploratory runs	fix in the controlled protocol	in our runs
single run per configuration	3 repetitions per point plus 3 cold boots; cluster bootstrap over repetitions; variance components	partly: 1 or 2 repetitions inside one boot, no cold boots
usable bars chosen after seeing the data	SLOs registered in <code>experiments/sla.yaml</code> before any controlled run	yes
all N requests sent at once (burst)	staggered closed-loop starts, plus open-loop Poisson arrivals	staggered starts (5 s); no Poisson
speculation makes decode content-dependent	separate code and prose scenarios; DSpark on and off (E7); acceptance length recorded	partly: code only, DSpark always on; acceptance length recorded (Table 14)
quality never evaluated	E10 and E11 with lm-evaluation-harness against the original checkpoint	partly: three task subsets, DeepSeek’s API as reference
per-chunk timing, several tokens per chunk	cumulative token counts on every chunk; per-token latency weighted by tokens	yes
corpus from the installed package, chars-per-token guess	pinned corpora, exact token counts via the server tokenizer	yes
5 s sampler, no PCIe, no per-process memory	0.5 s sampler with NVML PCIe RX/TX, cgroup and per-process memory by kind, mapping dump	sampler yes, PCIe per level reported (Table 13); no Nsight capture, mappings not analyzed
plan of the sweep not recorded	plan path and sha256 in every run’s configuration log	yes
aggregate rate includes ramp and drain	windowed aggregate rate (first send plus warm-up to last send)	yes

Table 6: Known weaknesses of the exploratory measurements, how the controlled protocol addresses them, and which fixes our three pod sessions applied.

decode throughput across load points and plans (Pearson and Spearman with bootstrap intervals). *In our runs* the sampler and `nvidia-smi dmon` ran in every session and the calibration script once, in the first (its zero-copy test on the second GPU is invalid); we report PCIe receive traffic per level and per output token (Section 8.2), but the Nsight capture, the mapping-based attribution of allocations and the PCIe correlation analysis were not done.

6.6 Cost model and service-level objectives

Cost per hour is the pod price (USD, with EUR at the ECB reference rate of 2026-10-05, 1.1204 USD per EUR; `experiments/pricing.yaml`). Cost per million output tokens divides it by the sustained output rate. Splitting cost between input and output tokens is a modelling choice, so we report three rules: all cost to output; dedicated (input cost from the prefill probe at saturation, output cost from the decode probe); and a fixed price ratio solved at a mixed load point. Cost per concurrent request is the hourly price divided by N ; cost per request is the level’s pod time divided by its requests. Cost under an SLO uses the highest-throughput load point that meets the objective in every repetition. The three objectives (interactive: TTFT p95 ≤ 2 s and TPOT p95 ≤ 50 ms; relaxed: 10 s and 100 ms; long context: 5 s and 66.7 ms) were registered on 2026-10-06 before any controlled run, but after we had seen the exploratory data (`experiments/sla.yaml`).

6.7 Quality evaluation

E10 and E11 run lm-evaluation-harness [77] 0.4.13 (task names validated locally; `quality/tasks.yaml`) against the measured stack and the reference: GSM8K [17] and MATH-500 [45] for reasoning, HumanEval [13] and MBPP [5] for code, IFEval [100] for instruction following, and RULER [46] and LongBench [6] subsets for long context. Both systems get identical prompts, greedy decoding, thinking off and fixed seeds, with per-item logs, so that differences are tested per item with a paired bootstrap. As designed, the reference is DeepSeek’s original checkpoint on upstream vLLM on the B300 pod (E11). *In our runs* only fixed-seed subsets of GSM8K, IFEval and HumanEval were evaluated (`quality/subsets_lean2h_seed20261006.json`), and the reference is the official DeepSeek API with thinking disabled, reached through a local proxy that logs token usage (`quality/api_proxy.py`), and reported separately because its serving precision and implementation cannot be verified. MATH-500, MBPP, RULER and LongBench were not run. Quantization can change behaviour that averages hide [32], so we also report the share of items whose outcome flips.

6.8 Designed but not run

Six experiments of the protocol were designed and implemented but not run, for budget reasons; their full specifications, workloads and runners are in `experiments/experiments.yaml` and `experiments/runs/`, and none of their numbers appears in this paper. E9 would serve the original checkpoint fully in VRAM on two B300 GPUs, the reference against which the cost of offload would be measured. E7 would turn DSpark speculative decoding off to separate its effect, on code and on prose. E4 would compare placement policies at a fixed host share, including a random plan and one fitted to our own routing counts. E5 would compare int4 and FP8 Engram tables in host memory. E6 would vary the number of SMs given to the side stream that computes host experts. E1 would rebuild every variant from scratch on a CPU machine and record image digests.

7 Experimental Setup

Hardware. The measured configuration is a RunPod pod with two NVIDIA RTX PRO 6000 Blackwell Server Edition GPUs (sm_120, 97,887 MiB reported per card), connected to the host over PCIe, at 4.18 USD per pod-hour (3.73 EUR); the exploratory pod had a cgroup memory limit of 351.1 GiB. The designed reference (E9, not run) would use two B300 GPUs at 15.78 USD per hour (twice the single-GPU quote of 7.89 USD). A single 80 GB A100 (1.59 USD per hour) is out of reach: according to the exploratory notes the pinned vLLM has no Ampere path for this model’s sparse attention (Ampere support is an open feature request, `audit/evidence/gh_issue_57144.json`), and neither checkpoint fits (Section 5). Every run records the GPU model, driver, PCIe link generation and width, NUMA topology, CPU and cgroup limits (`experiments/lib.sh`).

Software. All inputs are pinned in `stack/pins.env`: the vLLM nightly image by digest (vLLM 0.30.1rc1.dev312+g36768d1bf), vllm-exl3 d3cfd394, ExLlamaV3 5be88657, the sfxnz recipe 460ac89c and Diffbot’s recipe at Hub revision a1cf1bd2, verified file by file against `stack/recipe.lock.tsv`. Each variant is built inside the pinned image by `stack/build/bootstrap_pod.sh` (or the Dockerfile wrapping it), which records versions, git revisions, hashes of patched files and the full package list. Common serving settings for every variant: tensor parallel 2, a context limit of 262,144 tokens, an fp8 KV cache of 4.25 GiB, up to 64 sequences, 16,384 batched tokens per step, CUDA graphs for decode with capture sizes up to 256, prefix caching on, text only (`stack/serve.sh`).

Models. The EXL3 pack is `diffbot/DeepSeek-V4.1-Flash-EXL3-3bpw-2x-RTX-PRO-6000` at revision a1cf1bd2 (309.3 GiB); the reference is `deepseek-ai/DeepSeek-V4.1-Flash` at 2cba9e42 (475.2 GiB). Both are under the mit license as declared on the Hub (`data/static/hub/`); weights are downloaded on the pod at their pinned revisions and never redistributed.

Measurement campaign. Every controlled measurement comes from three sessions on 2x RTX PRO 6000 pods of this type, each on a different host: on 2026-10-06 (101 minutes: E2, E8, E10 and E3), on 2026-10-07 (43 minutes: E12, the E3 baseline again and the UVA rung) and a third session (57 minutes: E13). Table 7 gives the hardware fingerprint of each host: the GPUs, driver and PCIe link are the same, but the CPU model and the NUMA placement of the two GPUs are not (the third session’s record notes the first host’s public IP, but its CPU and GPU bus ids differ, so it is a different machine). The quality reference (E11) ran through DeepSeek’s official API instead of on our own hardware. Appendix A lists the status of every experiment and the budget and process details of the campaign.

session	run	CPU	threads	GPU NUMA	GPU link	driver	PCIe max	bus ids
1 (pod 1)	E8-main/main	AMD EPYC 9534 64-Core	224	1, 1	NODE	595.91.07	Gen5 x16	81:00.0, E1:00.0
2 (pod 2)	E12-long- context/long	AMD EPYC 9535 64-Core	256	0, 1	SYS	595.91.07	Gen5 x16	06:00.0, 86:00.0
3 (pod 3)	E13-coding- agent/agent	AMD EPYC 9535 64-Core	256	1, 1	NODE	595.91.07	Gen5 x16	86:00.0, F4:00.0

Table 7: Hardware fingerprint of the three pod sessions (all 2x RTX PRO 6000 Blackwell Server Edition, same datacenter), from one run folder per session: CPU model and threads visible to the container (`fingerprint.json`), NUMA node of each GPU and the path between GPU0 and GPU1 (`nvidia-smi topo -m`: NODE within one NUMA node, SYS across the inter-socket link), driver, maximum PCIe link and GPU bus ids (`nvidia-smi -q`).

8 Results

Results come in three strictly separated kinds. *Static* results (Section 5) are measurements of public files that anyone can repeat byte for byte. *Exploratory* results (Section 8.1) come from earlier single runs with known weaknesses; they motivate the protocol and are never mixed into controlled tables. *Controlled* results (Section 8.2) come from our three pod sessions, the first on 2026-10-06; the experiments that did not run are listed in Section 6.8.

8.1 Exploratory results (historical single runs)

exploratory. Everything in this subsection comes from runs made on 2026-10-06 before this protocol existed: one run per configuration, a single pod, all requests of a level sent at once, usable bars chosen after seeing the data, and no quality evaluation. The files are imported verbatim with hashes (`data/exploratory/MANIFEST.tsv`); the statistics below are recomputed from them by `src/hostoffload/analysis/exploratory.py`.

Concurrency sweep. Tables 8 and 9 re-analyze the closed-loop sweep from $N = 1$ to 64 on the `gen30` plan. In scenario A (unique 4K-token code prompts) one request decoded at a median of 116.2 tokens/s with a TTFT of 0.96 s; at $N = 8$ the median per-request rate was 23.1 tokens/s and at $N = 64$ 4.6. The aggregate output rate grew only 1.9-fold from $N = 1$ to $N = 8$ and 2.9-fold to $N = 64$ (234.1 tokens/s). With a cached 16K-token prefix (scenario B) the aggregate reached 398.2 tokens/s at $N = 64$ and TTFT stayed at 2.23 s median. There were 0 errors in 264 requests in A and 0 in 264 in B. Figure 5 shows every request.

The first wave inflates the tail: in A at $N = 64$ the TTFT p95 was 33.48 s over all requests and 8.70 s over the requests sent after the first wave (Figure 6). This is the reason the controlled protocol staggers starts (our runs did) and adds Poisson arrivals (designed, not run). The mean DSpark acceptance length was 3.37 in A and 3.90 in B at $N = 8$; in the first run, 480-token prose answers (an essay and two explanations) decoded at 31.2 to 58.6 tokens/s for one stream, against 116.2 for code reviews, which is why the protocol separates content classes and switches speculation off in E7 (designed, not run; all our controlled runs used code prompts with DSpark on).

These numbers cannot say what the offload costs: there is no run without it (it does not fit, Section 5), no run at another host share, and no all-in-VRAM reference. The drop of per-request decode with N may come from host-expert reads, from the growth of the speculative verification batch, from prefill interference, or from the eager execution of large decode steps above the CUDA graph sizes. The ablations designed to separate these (E4 to E7) did not run, so the controlled results below do not separate them either.

Memory. In the exploratory boot the server reported 76.78 GiB of weights per GPU (loaded in 121 s), 0.82 GiB of CUDA graphs and a KV pool of 623,891 tokens, enough for 2.38 requests at the full context limit. GPU memory stayed between 91,345 and 91,367 MiB at every load level: the pools are preallocated. Host memory did not move with load either: anonymous memory 70.1 to 71.3 GiB, shared memory 144.3 GiB, with page cache filling the cgroup to 350.6 GiB. The static split predicts 163.6 GiB of pinned file bytes for this plan, more than the shared memory alone; how pinned buffers divide between anonymous and shared mappings was not recorded; the controlled sampler dumps the largest mappings of every server process, but we have not analyzed those dumps. Raising the context limit to 1M tokens raised the weight memory to 77.15 GiB and graphs to 1.47 GiB per GPU and gave 1,071,558 KV tokens from the same KV budget.

Prefix caching. In the first run a fresh code prompt took 3.09 s to the first token (14,855 tokens) and a second turn on the same prefix 0.83 s, with 14,848 of 14,901 prompt tokens served from the cache (2x RTX PRO 6000 despite the folder name, `data/exploratory/inferencekey-deepseek@546071a/vllm-ex13/a100/results/agent30/bench.json`).

8.2 Controlled results

The first 2x RTX PRO 6000 pod session (101 minutes, `data/controlled/LEAN/session.json`) ran E2, E8, E10 and E3; a second session on 2026-10-07, on another host (43 minutes, `data/controlled/LEAN/session2.json`), added the long-context experiment E12 and retried the UVA rung; and a third, on a third host (57 minutes, `data/controlled/LEAN/session3.json`), ran the coding-agent experiment E13 of Section 10. Table 7 gives the hardware of each host. Boots invalidated by bootstrap bugs (two in the first session, one in the second) and a failed stack fetch in the third, all fixed in the repository, are not in the data (Appendix A). The tables below are generated by `make analyze` from `data/controlled/` and include only experiments that ran. Each experiment’s measurements come from a single boot (the `gen30` configuration was booted on all

N	n	TTFT p50	TTFT p95	p95 later	decode p50 [CI]	p5	aggr.	accept	USD/M out
1	8	0.96	1.02	1.02	116.2 [113.0, 125.0]	111.2	82.1	3.28	14.15
2	8	0.99	1.80	1.80	70.9 [63.0, 94.7]	54.0	112.2	3.39	10.35
4	8	1.48	3.11	2.03	40.1 [33.8, 60.2]	30.0	126.0	3.29	9.22
8	16	2.04	4.99	2.55	23.1 [21.5, 26.3]	16.7	152.1	3.37	7.63
16	32	3.39	9.20	3.42	13.8 [13.3, 15.2]	9.4	176.0	3.38	6.60
32	64	5.05	18.47	5.82	8.2 [7.7, 9.1]	5.8	209.3	3.37	5.55
64	128	8.34	33.48	8.70	4.6 [4.3, 4.8]	3.4	234.1	3.34	4.96

Source:

`data/exploratory/inferencekey-deepseek@546071a/vllm-exl3/pro6000-concurrency/results/scenario_A.json`. “p95 later”: TTFT p95 of requests sent after the first wave. Price: `experiments/pricing.yaml`.

Table 8: **EXPLORATORY** (single run, burst arrivals, not controlled). Scenario A: independent requests, a different 4K-token code prompt each (0 prefix-cache hits); 256 output tokens, closed loop, 2x RTX PRO 6000, DSpark $k = 5$. TTFT and decode percentiles are nearest rank on n requests (p95 is close to the maximum for $n < 20$). Bracket: 95% request-level bootstrap interval of the median, which ignores run-to-run variance. Cost: output-only attribution at the measured aggregate rate.

N	n	TTFT p50	TTFT p95	p95 later	decode p50 [CI]	p5	aggr.	accept	USD/M out
1	8	0.41	0.49	0.42	136.4 [127.1, 149.6]	121.8	113.0	3.87	10.28
2	8	0.43	0.81	0.44	95.5 [84.5, 101.6]	72.0	149.5	3.92	7.77
4	8	0.49	0.94	0.79	61.7 [53.2, 72.4]	51.2	208.2	3.90	5.58
8	16	1.01	1.31	1.31	31.5 [29.7, 37.3]	23.8	223.7	3.90	5.19
16	32	1.44	1.81	1.81	18.8 [18.0, 20.8]	15.0	264.8	3.95	4.38
32	64	2.09	10.32	2.09	11.1 [10.7, 11.6]	9.0	270.1	3.95	4.30
64	128	2.23	4.39	2.45	7.1 [6.9, 7.3]	6.0	398.2	4.03	2.92

Source:

`data/exploratory/inferencekey-deepseek@546071a/vllm-exl3/pro6000-concurrency/results/scenario_B.json`. “p95 later”: TTFT p95 of requests sent after the first wave. Price: `experiments/pricing.yaml`.

Table 9: **EXPLORATORY** (single run, burst arrivals, not controlled). Scenario B: 16K-token shared code prefix (cached) plus a short question; 256 output tokens, closed loop, 2x RTX PRO 6000, DSpark $k = 5$. TTFT and decode percentiles are nearest rank on n requests (p95 is close to the maximum for $n < 20$). Bracket: 95% request-level bootstrap interval of the median, which ignores run-to-run variance. Cost: output-only attribution at the measured aggregate rate.

three pods, but no measurement was repeated across boots): the intervals are over repetitions inside a boot, never across boots.

Feasibility (E2). Both configurations without offload fail at load with a GPU out-of-memory error (Table 10): the EXL3 plugin with every expert in VRAM, and the offload build with an empty plan. With the `gen30` plan the same build boots and serves (E8 below).

This confirms on the hardware what the static accounting of Section 5 predicts: on this pair of GPUs the offload is what makes the configuration run.

Concurrency frontier (E8). Tables 11 (latency) and 12 (throughput and SLOs) give the measured stack with the `gen30` plan. With a 16K-token shared prefix (B) the aggregate output rate grows from 142 tokens/s at $N = 1$ to 307 at $N = 32$ and 340 at $N = 64$, and TTFT p95 stays at or below 2.03s at every level, because the prefix is served from the cache. With unique 4K-token code prompts (A) the rate saturates earlier: 125 at $N = 8$, 186 at $N = 32$ and 180 at $N = 64$, while TTFT p95 rises to 27.29s at $N = 64$. Per-token latency grows with load: in A the median TPOT is 6.6 ms at $N = 1$, 33.6 ms at $N = 8$ and 182.2 ms at $N = 64$. The prefill probes take 0.63, 1.56 and 6.19s for 4K, 16K and 64K-token prompts. B has two repetitions at every level; A has two only at $N = 1$. In A’s second repetition one of the eight requests at $N = 1$ failed with a client-side disconnect, the pre-registered stop rule (error rate above 5%) treated the level as saturated and ended that sweep. We report the data as collected; a stop rule evaluated on eight requests is too sensitive, and Section 13 lists it. A’s $N = 1$ row also pools a slow first repetition (TTFT p95 9.44s), so its aggregate mean is below the second repetition’s and is not a steady-state rate.

Figure 7 shows the same sweep per stream. The median output rate of one request (tokens after the first divided by the time from first to last token) falls from 151.5 tokens/s at $N = 1$ to 29.7 at $N = 8$ and 5.5 at $N = 64$ in A, and from 161.8 to 37.8 and 8.3 in B, while the aggregate keeps growing: concurrency buys total

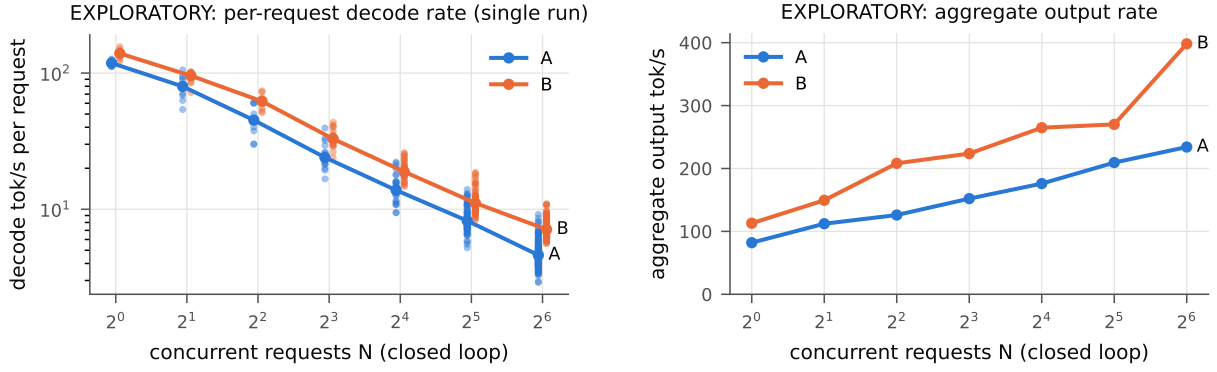


Figure 5: EXPLORATORY (single run). Left: per-request decode rate (dots: requests; lines: medians) against concurrency. Right: aggregate output rate. Sources: `data/exploratory/.../results/scenario_A.json` and `scenario_B.json`.

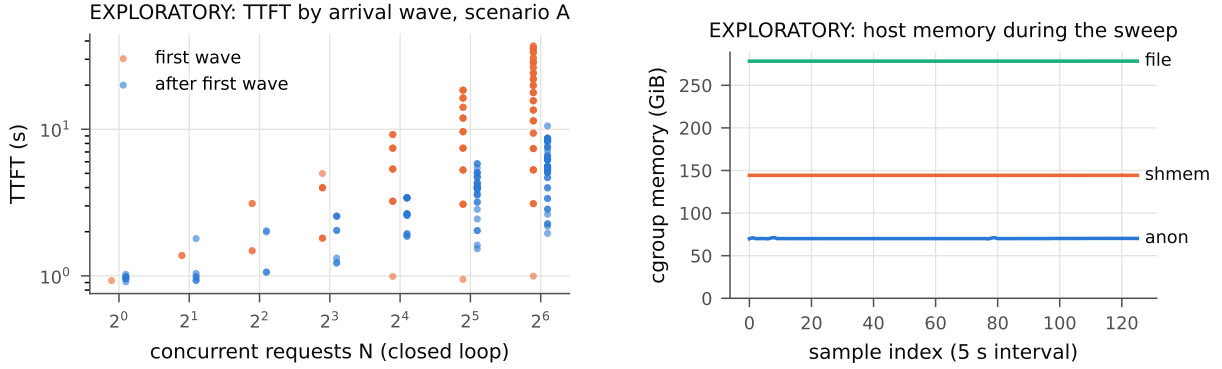


Figure 6: EXPLORATORY (single run). Left: TTFT of first-wave requests (all sent at the start of a level) and of later requests, scenario A. Right: host memory of the pod’s cgroup during the sweep, 5 s samples (`results/mem_samples.csv`).

throughput at the price of each stream’s speed. At $N = 1$ the per-stream distribution of A has 15 requests (one failed) and spans 116.9 to 163.0 tokens/s from p5 to p95; it is not affected by the slow first request, whose cost is in TTFT.

PCIe traffic and speculation (E8, E13). The NVML sampler records the PCIe receive rate of each GPU every 0.5 s. Table 13 gives it per level, summed over both GPUs, and divides the bytes received in a level by the output tokens completed in it. The summed rate reaches 85.0 GB/s at most in E8 (B at $N = 32$) and 91.3 GB/s at most in E13, against a line rate of 63 GB/s per GPU and direction for a PCIe Gen5 x16 link. In E8 the bytes per output token fall with concurrency, from 473 to 231 MB in A and from 326 to 150 MB in B between $N = 1$ and $N = 64$, as expected if one read of a host-resident expert serves several requests of a batch; in E13 they do not fall (365 MB at $N = 1$ and 599 MB at $N = 12$ in its regime A). We cannot explain the difference with these data. The two experiments differ in more than concurrency: E13 forces 400-token replies against 256 in E8, ran on another host (pod 3 against pod 1), and its sweeps stop at levels where the aggregate rate had already started to fall, so fewer tokens complete in a window while host reads continue; and the counter mixes prompt prefill, drafter and expert reads. We report the trend without interpretation. E13 (Section 10) has four regimes, which the tables label by letter: A, independent 4K-token prompts; B, a shared cached prefix of 128K or 256K tokens; C, an independent cached context of 128K or 256K tokens per stream; D, agent turns. E13’s B is not E8’s B, which is a 16K-token cached prefix. These ratios are descriptive: the counter includes the prefill of new prompts, the drafter’s host reads and input transfers, and we did not attribute the bytes to these sources. The counter itself is only partly validated: on pod 1 a calibration script found NVML reporting 1.11 and 1.10 times the true bytes for copies on GPU 0 and 1, and 1.22 times for zero-copy reads on GPU 0; the zero-copy test on GPU 1 is invalid (its kernel did not read over PCIe), and pods 2 and 3 have no calibration. The values are therefore raw and uncorrected.

configuration	status	failure	first error line
exl3-min-allvram	boot_failed	gpu_oom	(Worker_TP1 pid=11812) ERROR 10-06 20:04:41 [multiproc_executor.py:948] torch.OutOfMemoryE
exl3-offload-noplan	boot_failed	gpu_oom	(Worker_TP1 pid=15785) ERROR 10-06 20:08:11 [multiproc_executor.py:948] torch.OutOfMemoryE

Table 10: Controlled: feasibility boots on 2x RTX PRO 6000 (E2). Status and the first matching error line of the server log (`hostoffload.analysis.failures`).

scenario	load	n	TTFT	p50	p95	p99	TPOT	p50	p95	E2E	p50	p99
<i>main</i> , pod 1												
A-code-4k	closed:1	16		0.65	9.44	25.81		6.6	9.6		2.3	29.4
A-code-4k	closed:2	8		0.68	1.31	1.31		9.8	20.8		3.5	6.4
A-code-4k	closed:4	8		0.65	1.28	1.30		22.1	27.7		6.3	8.4
A-code-4k	closed:8	16		1.13	2.26	2.26		33.6	41.5		9.4	12.6
A-code-4k	closed:16	32		2.65	5.40	5.47		54.2	78.1		17.2	25.0
A-code-4k	closed:32	64		3.57	10.15	10.38		87.5	130.0		26.5	39.9
A-code-4k	closed:64	128		4.77	27.29	28.85		182.2	235.2		53.2	83.4
B-prefix-16k	closed:1	16		0.30	0.31	0.32		6.2	7.0		1.9	2.2
B-prefix-16k	closed:2	16		0.42	0.45	0.46		9.0	10.3		2.6	3.1
B-prefix-16k	closed:4	16		0.44	0.58	0.68		12.0	15.6		3.5	4.7
B-prefix-16k	closed:8	32		0.52	0.82	0.89		26.4	31.4		7.3	8.9
B-prefix-16k	closed:16	64		0.78	1.07	1.16		44.3	55.9		12.1	16.3
B-prefix-16k	closed:32	128		1.19	1.59	1.66		65.3	87.7		17.9	24.9
B-prefix-16k	closed:64	256		1.67	2.03	2.18		121.1	153.0		32.3	44.1
<i>probes</i> , pod 1												
decode	closed:1	4		0.19	0.21	0.21		6.7	7.9		3.6	4.3
decode	closed:8	8		0.85	0.85	0.85		21.2	22.6		11.7	12.3
decode	closed:32	32		15.15	15.32	15.32		59.5	66.2		45.3	49.0
prefill-4096	closed:1	3		0.63	0.66	0.66		0.0	0.0		0.6	0.7
prefill-16384	closed:1	3		1.56	1.60	1.60		0.0	0.0		1.6	1.6
prefill-65536	closed:1	3		6.19	6.39	6.41		0.0	0.0		6.2	6.4

Table 11: Controlled: Main benchmark of the measured stack as run (gen30 plan; scenarios A and B at $N = 1$ to 64, two repetitions in B and in A only at $N = 1$; prefill and decode probes, one repetition). Part 1 of 2, per-request latency: time to first token (TTFT, s), time per output token (TPOT, ms) and end-to-end time (E2E, s), with requests pooled over repetitions. Rows are grouped by variant and pod session (Table 7). Throughput and SLOs of the same load points: Table 12.

The same server counters give DSpark’s acceptance (Table 14). It proposes 5 draft tokens per step, and a request gains on average 3.37 tokens per verification step in E8 A and 3.60 in B (47% and 52% of the proposed tokens accepted), almost independent of N (in A 3.39 at $N = 1$ and 3.37 at $N = 64$). The E13 regimes, pooled over their levels, range from 3.37 for short independent prompts to 3.77 for independent 256K-token contexts; single levels at small N rest on a few hundred drafts and are noisier. All these prompts are code or coding-agent turns with thinking off, so the values say nothing about prose.

Host share and UVA (E3). Moving from the `gen30` plan to the union of the `gen30` and `agent30` cold sets (a larger share of the experts on the host) lowers the aggregate rate of scenario A from 112 to 77 tokens/s at $N = 1$, from 147 to 116 at $N = 8$ and from 198 to 140 at $N = 32$ (Table 16; one repetition per point, two boots). The decode scenario of the union run is missing: the load generator stopped on a client-side disconnect while the server kept serving. Upstream vLLM’s UVA offload of the same EXL3 experts, at the same host bytes and without Diffbot’s hooks, could not serve at all. With the KV block size of 64 that the measured stack uses, it fails at initialization: the FlashInfer sparse-attention backend for DeepSeek-V4.1 accepts only 128 (first session, `data/controlled/E3-host-share/uva-equal-gen30/`). With 128 it boots, but the engine dies on the first request, because DeepGEMM’s attention kernel asserts a KV block of 32 or 64 (`data/controlled/E3-host-share/uva-equal-gen30-bs128/`). On this hardware and these pins, Diffbot’s hooks are therefore needed to serve at all, so the comparison measures nothing about the speed of the two

scenario	load	R	n	aggregate tok/s	CV%	SLO
<i>main, pod 1</i>						
A-code-4k	closed:1	2	16	73.6	68.9	-
A-code-4k	closed:2	1	8	97.9	-	i,r,l
A-code-4k	closed:4	1	8	95.6	-	i,r,l
A-code-4k	closed:8	1	16	125.2	-	r,l
A-code-4k	closed:16	1	32	146.5	-	r
A-code-4k	closed:32	1	64	186.0	-	-
A-code-4k	closed:64	1	128	179.9	-	-
B-prefix-16k	closed:1	2	16	141.7	5.2	i,r,l
B-prefix-16k	closed:2	2	16	159.9	0.8	i,r,l
B-prefix-16k	closed:4	2	16	147.7	4.6	i,r,l
B-prefix-16k	closed:8	2	32	177.7	5.7	i,r,l
B-prefix-16k	closed:16	2	64	208.9	7.2	r,l
B-prefix-16k	closed:32	2	128	307.2	4.3	r
B-prefix-16k	closed:64	2	256	339.5	7.5	-
<i>probes, pod 1</i>						
decode	closed:1	1	4	133.2	-	i,r,l
decode	closed:8	1	8	333.3	-	i,r,l
decode	closed:32	1	32	334.5	-	-
prefill-4096	closed:1	1	3	1.6	-	-
prefill-16384	closed:1	1	3	0.6	-	-
prefill-65536	closed:1	1	3	0.2	-	-

Table 12: Controlled: Main benchmark of the measured stack as run (gen30 plan; scenarios A and B at $N = 1$ to 64, two repetitions in B and in A only at $N = 1$; prefill and decode probes, one repetition). Part 2 of 2, throughput and objectives. R: repetitions at that point; aggregate: mean over repetitions of the windowed output rate (tokens/s), CV its coefficient of variation across repetitions (- for a single repetition); SLO: initials of the pre-registered SLOs met in every repetition. Latency of the same load points: Table 11.

offload mechanisms; whether the per-expert split is faster than UVA remains open. Tables 15 and 16 also hold the **gen30** subset repeated on pod 2 (variant **gen30-session2**, pod 2), which is not a third host share; the next paragraph compares it with pod 1.

Across pods. The second session repeated the E3 **gen30** subset (same stack, plan and workload file) on another host. Table 17 compares the two runs point by point: pod 2 was slower at every point, with a median TPOT 1.17 to 1.51 times pod 1’s. Its aggregate rate was 0.62 to 0.83 times pod 1’s, except for A at $N = 1$ (0.46), where a single request with a TTFT of 16.3s explains most of the gap. The GPU model, driver and maximum PCIe link are the same on both hosts (Table 7), but the CPU differs (AMD EPYC 9534 64-Core against AMD EPYC 9535 64-Core) and so does the topology: pod 1 has both GPUs on one NUMA node, pod 2 one GPU on each socket, so with tensor parallelism across the two GPUs and experts read from host memory traffic crosses the socket link. That is a plausible cause, but one repetition per pod cannot separate it from the CPU or from boot-to-boot variance (both servers had served other workloads on the same boot before this subset). The cloud placement of a pod is thus a first-order factor, and numbers from different pods are not directly comparable. The third session’s record notes the same public IP as the first, but its CPU model and GPU bus ids differ, so we treat it as a third host; it did not run the E3 subset.

Long context (E12). Tables 18 and 19 repeat the measurements with prompts near the 262,144-token context, on the second pod (so against its own, slower, baseline above). vLLM sizes the recipe’s KV cache at 623,891 tokens, 2.38 full-length requests at once, which is why this experiment ran unique long prompts only up to $N = 4$ at 128K and $N = 2$ at 256K; E13 later ran five independent 128K and three 256K contexts with at most 48% and 56% KV-cache use and no preemption, so for this model’s compressed attention that estimate is conservative (Section 10). Prefill dominates: a fresh prompt of 128K tokens takes 18.33s to its first token and one of 260,000 tokens 45.07s. Decode itself barely slows down for a single request: the median TPOT is 8.7ms after a 128K prompt and 8.3ms after a 256K one, against 7.6ms after a 4K prompt on the same pod (**gen30-session2**, Table 15), but it grows fast with concurrency (68.4ms at four 128K requests, 48.2ms at two 256K requests). When the long context is a shared, cached prefix, as for an agent working on one large codebase, the stack stays interactive: with 128K tokens of prefix TTFT p95 is 1.18s at eight users, and with 256K tokens it is 1.66s at four users with a median TPOT of 16.6ms. So the E8 rates hold for long

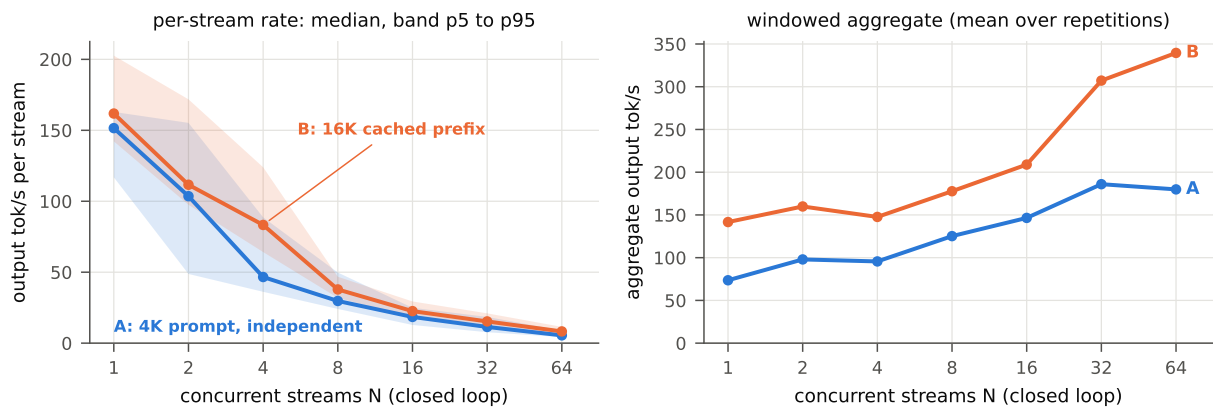


Figure 7: Controlled E8 sweep (pod 1, `gen30` plan). Left: output rate per stream against the number of concurrent users N , median with a p5 to p95 band over the requests of each level (warm requests and errors excluded, repetitions pooled). Right: windowed aggregate output rate, mean over repetitions (two in B; in A two only at $N = 1$, where the mean includes a slow first repetition). A: independent 4K-token code prompts; B: 16K-token cached prefix.

cached prefixes at modest concurrency, but not for many users with distinct long prompts: there prefill time, not the offload, sets the limit. Each point is one repetition on one boot.

regime	N	PCIe RX GB/s	MB per output token
E8 A: 4K independent	1	25.2	473
	8	72.4	395
	64	66.3	231
E8 B: 16K cached prefix	1	36.5	326
	8	76.2	356
	64	68.2	150
E13 A: 4K independent	1	15.1	365
	8	75.3	373
	12	91.3	599
E13 B: 128K cached prefix	1	26.5	290
	8	78.6	324
	12	90.7	515
E13 B: 256K cached prefix	1	27.9	312
	8	76.4	330
	12	88.7	497
E13 C: own 128K context	1	50.2	455
	5	84.2	570
E13 C: own 256K context	1	49.1	464
	3	59.6	325
E13 D: agent turns	1	49.0	412
	8	81.6	386

Calibration on pod 1 (`data/controlled/E8-main/pcie-calibration.json`): NVML over true bytes 1.11 for DMA copies and 1.22 for zero-copy kernel reads on GPU 0. The GPU 1 zero-copy test is invalid: its nominal 294 GB/s exceeds the 63 GB/s PCIe 5.0 x16 line rate and NVML saw almost no traffic, so the kernel did not read over PCIe. Values are not corrected.

Table 13: PCIe receive traffic summed over both GPUs (NVML, raw counter) per level, and megabytes received per output token completed in the level window. E8 ran on pod 1 (two repetitions for B and for A at $N = 1$, pooled), E13 on pod 3 (one repetition). Rows: $N = 1$, $N = 8$ and the largest tested N ; every level is in `values.tex` (`ctl.pcie.*`). Descriptive: the bytes include prefill and drafter traffic.

regime	N	drafts	accepted/draft	accept. length	accepted %
E8 A: 4K independent	1	1,139	2.39	3.39	48
	8	1,192	2.44	3.44	49
	64	9,744	2.37	3.37	47
	all	20,627	2.37	3.37	47
E8 B: 16K cached prefix	1	1,134	2.63	3.63	53
	8	2,318	2.54	3.54	51
	64	18,279	2.58	3.58	52
	all	37,575	2.60	3.60	52
E13 A: 4K independent	1	450	2.57	3.57	51
	8	1,903	2.37	3.37	47
	12	2,852	2.37	3.37	47
	all	9,991	2.37	3.37	47
E13 B: 128K cached prefix	1	472	2.39	3.39	48
	8	1,677	2.82	3.82	56
	12	2,760	2.48	3.48	50
	all	9,336	2.60	3.60	52
E13 B: 256K cached prefix	1	417	2.85	3.85	57
	8	1,763	2.64	3.64	53
	12	2,718	2.54	3.54	51
	all	9,315	2.62	3.62	52
E13 C: own 128K context	1	223	2.60	3.60	52
	5	1,063	2.77	3.77	55
	all	3,272	2.67	3.67	53
E13 C: own 256K context	1	194	3.15	4.15	63
	3	653	2.67	3.67	53
	all	1,275	2.77	3.77	55
E13 D: agent turns	1	302	2.98	3.98	60
	8	2,594	2.71	3.71	54
	all	7,726	2.74	3.74	55

Table 14: DSpark acceptance per level from vLLM’s counters: drafts (`vllm:spec_decode_num_drafts`), accepted draft tokens per draft, acceptance length $1 + \text{accepted}/\text{drafts}$ (tokens a request gains per verification step) and the share of proposed draft tokens accepted. Rows: $N = 1$, $N = 8$, the largest tested N and all levels pooled; E8 on pod 1 (repetitions pooled), E13 on pod 3.

scenario	load	n	TTFT p50	p95	p99	TPOT p50	p95	E2E p50	p99
<i>gen30</i> , pod 1									
A-code-4k	closed:1	8	0.64	0.66	0.66	6.5	7.0	2.3	2.4
A-code-4k	closed:8	16	1.05	1.70	1.72	31.3	35.4	8.8	10.9
A-code-4k	closed:32	64	3.38	9.05	9.56	86.6	111.1	26.0	36.9
decode	closed:1	4	0.21	0.21	0.21	6.0	6.4	3.3	3.5
decode	closed:8	8	0.83	0.84	0.84	21.3	22.1	11.6	12.2
decode	closed:32	32	1.33	1.33	1.46	60.2	67.5	32.1	36.4
<i>gen30-session2</i> , pod 2									
A-code-4k	closed:1	8	0.84	10.89	15.21	7.6	7.9	2.8	17.1
A-code-4k	closed:8	16	1.65	2.81	2.87	43.3	55.2	13.2	16.0
A-code-4k	closed:32	64	4.88	17.75	18.41	130.9	180.1	38.5	61.9
decode	closed:1	4	0.22	0.24	0.24	7.4	9.4	4.0	5.1
decode	closed:8	8	0.94	0.94	0.94	29.4	30.4	15.7	16.6
decode	closed:32	32	12.33	12.33	12.50	77.2	90.3	51.5	58.6
<i>union-gen30-agent30</i> , pod 1									
A-code-4k	closed:1	8	0.88	1.76	2.12	8.7	9.7	3.2	4.3
A-code-4k	closed:8	16	1.92	2.77	2.77	42.0	54.8	12.6	16.5
A-code-4k	closed:32	64	4.51	13.03	13.65	121.2	165.2	36.3	53.7

Table 15: Controlled: Host share as run (rung (c) of the ablation ladder): the gen30 plan against the union of the gen30 and agent30 cold sets on pod 1, and the gen30 subset repeated on pod 2 (variant gen30-session2), one repetition per point. Part 1 of 2, per-request latency: time to first token (TTFT, s), time per output token (TPOT, ms) and end-to-end time (E2E, s), with requests pooled over repetitions. Rows are grouped by variant and pod session (Table 7). Throughput and SLOs of the same load points: Table 16.

scenario	load	R	n	aggregate tok/s	CV%	SLO
<i>gen30</i> , pod 1						
A-code-4k	closed:1	1	8	112.1	-	i,r,l
A-code-4k	closed:8	1	16	146.6	-	i,r,l
A-code-4k	closed:32	1	64	197.9	-	-
decode	closed:1	1	4	159.7	-	i,r,l
decode	closed:8	1	8	335.5	-	i,r,l
decode	closed:32	1	32	446.9	-	r
<i>gen30-session2</i> , pod 2						
A-code-4k	closed:1	1	8	51.4	-	-
A-code-4k	closed:8	1	16	102.2	-	r,l
A-code-4k	closed:32	1	64	125.4	-	-
decode	closed:1	1	4	132.2	-	i,r,l
decode	closed:8	1	8	246.3	-	i,r,l
decode	closed:32	1	32	278.3	-	-
<i>union-gen30-agent30</i> , pod 1						
A-code-4k	closed:1	1	8	76.6	-	i,r,l
A-code-4k	closed:8	1	16	115.9	-	r,l
A-code-4k	closed:32	1	64	139.7	-	-

Table 16: Controlled: Host share as run (rung (c) of the ablation ladder): the gen30 plan against the union of the gen30 and agent30 cold sets on pod 1, and the gen30 subset repeated on pod 2 (variant gen30-session2), one repetition per point. Part 2 of 2, throughput and objectives. R: repetitions at that point; aggregate: mean over repetitions of the windowed output rate (tokens/s), CV its coefficient of variation across repetitions (- for a single repetition); SLO: initials of the pre-registered SLOs met in every repetition. Latency of the same load points: Table 15.

scenario	N	aggr.	pod 1	pod 2	ratio	TPOT p50	pod 1	pod 2	ratio
A: 4K prompt	1		112	51	0.46		6.5	7.6	1.17
	8		147	102	0.70		31.3	43.3	1.38
	32		198	125	0.63		86.6	130.9	1.51
decode	1		160	132	0.83		6.0	7.4	1.23
	8		335	246	0.73		21.3	29.4	1.38
	32		447	278	0.62		60.2	77.2	1.28

Sources: `data/controlled/E3-host-share/gen30/20261006T203206Z`,
`data/controlled/E3-host-share/gen30-session2/20261007T063646Z`. Levels whose slowest request had a TTFT above 10 s: A: 4K prompt $N = 1$ on pod 2, 16.3 s; A: 4K prompt $N = 32$ on pod 2, 18.4 s; decode $N = 32$ on pod 2, 12.6 s.

Table 17: The same E3 subset workload (gen30 plan, identical stack and workload file, one repetition) on pod 1 and pod 2: windowed aggregate output rate (tok/s) and median TPOT (ms) per load point, and the ratio pod 2 / pod 1. With one repetition per pod the ratio mixes the host difference with run-to-run and boot-to-boot variance.

scenario	load	n	TTFT p50	p95	p99	TPOT p50	p95	E2E p50	p99
<i>long, pod 2</i>									
B-prefix-128k	closed:1	4	0.52	0.57	0.57	6.8	7.2	2.3	2.3
B-prefix-128k	closed:2	4	0.54	0.68	0.70	8.7	11.1	2.8	3.4
B-prefix-128k	closed:4	8	0.66	1.19	1.33	13.2	19.2	4.0	6.1
B-prefix-128k	closed:8	16	0.79	1.18	1.24	38.3	46.7	10.5	12.7
B-prefix-256k	closed:1	4	0.88	0.89	0.89	7.3	7.5	2.7	2.8
B-prefix-256k	closed:2	4	0.93	0.94	0.94	9.9	10.8	3.5	3.7
B-prefix-256k	closed:4	8	1.26	1.66	1.70	16.6	19.6	5.4	6.1
decode-128k	closed:1	2	22.45	25.13	25.36	8.7	10.4	26.9	28.9
decode-128k	closed:2	2	27.10	34.05	34.67	29.1	43.1	42.0	42.2
decode-128k	closed:4	4	48.20	71.72	73.77	68.4	118.4	83.7	84.2
decode-256k	closed:1	2	44.75	47.79	48.06	8.3	9.3	49.0	52.9
decode-256k	closed:2	2	58.74	75.79	77.30	48.2	81.8	83.3	83.5
prefill-long-32768	closed:1	3	7.57	10.75	11.04	0.0	0.0	7.6	11.0
prefill-long-65536	closed:1	3	10.14	14.91	15.34	0.0	0.0	10.1	15.3
prefill-long-131072	closed:1	3	18.33	18.46	18.48	0.0	0.0	18.3	18.5
prefill-long-260000	closed:1	3	45.07	58.68	59.89	0.0	0.0	45.1	59.9

Table 18: Controlled: Long context at 128K and 256K tokens as run (pod 2; prefill, decode and cached shared prefix, one repetition). Part 1 of 2, per-request latency: time to first token (TTFT, s), time per output token (TPOT, ms) and end-to-end time (E2E, s), with requests pooled over repetitions. Rows are grouped by variant and pod session (Table 7). Throughput and SLOs of the same load points: Table 19.

scenario	load	R	n	aggregate tok/s	CV%	SLO
<i>long, pod 2</i>						
B-prefix-128k	closed:1	1	4	111.0	-	i,r,l
B-prefix-128k	closed:2	1	4	90.9	-	i,r,l
B-prefix-128k	closed:4	1	8	115.3	-	i,r,l
B-prefix-128k	closed:8	1	16	126.8	-	i,r,l
B-prefix-256k	closed:1	1	4	92.5	-	i,r,l
B-prefix-256k	closed:2	1	4	91.8	-	i,r,l
B-prefix-256k	closed:4	1	8	177.9	-	i,r,l
decode-128k	closed:1	1	2	20.6	-	-
decode-128k	closed:2	1	2	24.3	-	-
decode-128k	closed:4	1	4	24.3	-	-
decode-256k	closed:1	1	2	11.4	-	-
decode-256k	closed:2	1	2	12.3	-	-
prefill-long-32768	closed:1	1	3	0.1	-	-
prefill-long-65536	closed:1	1	3	0.1	-	-
prefill-long-131072	closed:1	1	3	0.1	-	-
prefill-long-260000	closed:1	1	3	0.0	-	-

Table 19: Controlled: Long context at 128K and 256K tokens as run (pod 2; prefill, decode and cached shared prefix, one repetition). Part 2 of 2, throughput and objectives. R: repetitions at that point; aggregate: mean over repetitions of the windowed output rate (tokens/s), CV its coefficient of variation across repetitions (- for a single repetition); SLO: initials of the pre-registered SLOs met in every repetition. Latency of the same load points: Table 18.

9 Cost Analysis

The cost model (`src/hostoffload/cost/model.py`) turns a pod price and a measured rate into cost per million tokens, per request and per concurrent request; it is unit-tested on the exploratory data (`tests/test_stats_cost.py`). All numbers in this section use the output-only attribution, so they are upper bounds on the cost of output tokens and say nothing about input tokens.

Controlled cost. At 4.18 USD per pod-hour (3.73 EUR), the E8 rates (Table 12) put one million output tokens at 15.78 USD for one user in scenario A, 9.27 at $N = 8$ and 6.24 at $N = 32$; with the cached 16K-token prefix (B) it costs 8.20, 6.53 and 3.42 at $N = 1, 8$ and 64. Cheap tokens at high N are slow tokens, and the SLO column of Table 12 marks which pre-registered objectives each level meets in every repetition: with the cached prefix the interactive objective holds at low and moderate concurrency and the relaxed one further, so the cheapest levels that meet an objective are in B; with unique prompts the objectives hold over a narrower range, and A’s first level misses them because of its slow first repetition (Section 8.2). The other attribution rules and the price of the same model without offload need the B300 reference (E9), which did not run. For coding agents, where the binding constraint is the number of simultaneous streams at 40 tokens/s rather than the cost per token, Section 10 turns the E13 capacities into developer seats per pod and compares them with per-seat subscriptions.

Exploratory cost. EXPLORATORY (single run, burst arrivals, percentiles from 8 to 128 requests; Section 8.1). The exploratory sweep implies 14.15 USD per million output tokens for one user in scenario A, 7.63 at $N = 8$ and 4.96 at $N = 64$, and 10.28, 5.19 and 2.92 in B; the cost per concurrent request falls as $1/N$, from 4.180 to 0.065 USD per request-hour. Applying the pre-registered objectives to it (with TPOT p95 approximated by the inverse of the nearest-rank p5 decode rate, because that run did not store per-token times), the interactive objective holds in A up to $N = 2$ at 10.35 USD per million output tokens, the relaxed one up to $N = 8$ at 7.63 USD, and the long-context one in B up to $N = 8$ at 5.19 USD. These are indications of where the frontier lies, not estimates with known uncertainty.

10 Case Study: Interactive Coding Agents

Coding agents are a demanding and common use of a large model: long contexts that are mostly reused from turn to turn, short new inputs, replies of a few hundred tokens, and a person waiting for them. This section turns the measurements into two capacities for that workload and into a cost comparison with a per-seat subscription. It is a workload case study. It compares serving capacity, concurrency, context reuse and infrastructure cost; it does not claim that DeepSeek-V4.1-Flash is equivalent to the models behind a commercial coding agent in quality or capability, and a subscription such as Claude Code buys a complete product (the agent, its tools, model access, infrastructure, reliability and support), while the self-hosted figures below are GPU rental only unless stated otherwise.

The model is a sensible candidate for this use: on a third-party leaderboard its unquantized, maximum-effort configuration ranks first on agentic coding (Section 2). That ranking motivates the case study and does not carry over to our stack, which was not evaluated on it.

Question and hypothesis. RQ6: how many simultaneous coding-agent streams can one pod sustain while each stream generates at least 40 output tokens/s (median per request), and how many intermittently active developer seats can share that pod under an observed usage profile? We expect the two answers to differ by an order of magnitude: monthly generation volume would allow many seats, while the per-stream objective would allow only a few streams at once, and the latter would bound the number of seats a pod can be provisioned for.

Observed usage profile. One developer’s Claude Code usage screen for 7 days (`experiments/agent_workload.yaml`) reports 94,600,000 tokens over 9 sessions on 5 active days, but only 387,700 of them (0.4%) are generated output; 90,800,000 are cache reads, 3,400,000 cache writes and 892 uncached input. Counting all of them as generated tokens would overstate the load by two orders of magnitude, so we keep the categories apart. Per output token the profile reads 234 tokens of cached context and writes 8.8 new ones; with replies of 400 tokens (an assumption) a turn reads about 93,681 tokens of context and adds about 3,508. These are the provider’s billing categories, not vLLM’s prefix-cache accounting, so the shape is an approximation that inspires the synthetic workload, not a token-for-token equivalence. Monthly output per developer is 1.69 million tokens.

regime	source	C	C _{+10%}	p5	p50	p95	TTFT p95	aggr.	fails at
A: short 4K, independent	E13	4	4	45	55	80	1.15	131	5 (32)
B: 128K cached prefix	E13	6	4	40	48	54	0.87	176	8 (37)
B: 256K cached prefix	E13	4	4	62	73	96	1.30	155	5 (40)
C: own 128K context	E13	4	4	49	58	67	1.72	172	5 (32)
C: own 256K context	E13	≥ 3	≥ 3	65	87	95	1.76	155	-
D: agent turns (observed ratios)	E13	6	4	34	40	58	2.99	178	8 (31)

Table 20: Simultaneous streams at a median of at least 40 output tokens/s per stream. C is the largest N whose level and every smaller tested level meet it (failing on 2 or more errors above 5%); per-stream rates (p5, p50, p95, tokens/s) and TTFT p95 (s) at C; the last column is the first failing N with its median rate. C_{+10%} is the same with a 10% margin (44 tokens/s), used for provisioning.

Two capacities. *Simultaneous-stream capacity* C is the largest number of streams N such that the level with N streams and every smaller tested level have a median per-request decode rate of at least 40 tokens/s (TPOT of 25 ms) and fail only by the harness error rule (at least two errors above 5%). Aggregate throughput never defines success: ten streams at 20 tokens/s are not five at 40. *Capacity-equivalent developer seats by observed monthly generation volume* divide the output a pod produces per month, at the aggregate rate measured at C and a stated utilization, by one developer’s monthly output; they are not concurrent users.

E13. E13 (experiments/workloads/agent.yaml, runner experiments/runs/agent.sh) ran six regimes on one boot of the gen30 stack (57 pod-minutes, 3.98 USD, data/controlled/LEAN/session3.json), all with 400-token replies: (A) independent short sessions with unique 4K-token prompts; (B) streams branching from one cached 128K or 256K-token prefix; (C) streams that each own an independent cached 128K or 256K-token context, opened by an unmeasured first turn and then extended by 512 new tokens per turn; and (D) agent turns shaped by the observed ratios, each stream owning a context of 88,000 tokens that grows by 3,500 new tokens per turn over three turns. In A and B, N grew through 1, 2, 3, 4, 5, 6, 8 and 12 and stopped when two levels fell below the objective; C and D ran fixed lists because of KV-cache and time limits (C at 128K up to 5, C at 256K up to 3, D at 1, 2, 3, 4, 6 and 8). Every level run is kept, with its service per stream in Table 21 and its resources in Table 22. The per-stream rate is the reply’s tokens over the time from its first token to the end of the stream, which equals 1/TPOT when tokens stream evenly; it also stays valid for the 6 of 366 requests whose reply arrived mostly in one late chunk, for which TPOT is meaningless. No request failed.

Simultaneous streams. Table 20 and Figure 8 give the answer to the first question. Every regime that was run to four streams keeps the objective up to four simultaneous streams and loses it between five and eight; independent 256K contexts were run only up to three streams, all above the objective. With a 10% margin (44 tokens/s), which we use for provisioning because single-repetition levels near 40 pass or fail by noise, the capacity is 4 for short independent sessions, 4 and 4 for a shared 128K or 256K-token cached prefix, 4 for independent cached 128K contexts, at least 3 for independent 256K contexts (the largest level run), and 4 for agent turns. At exactly 40 tokens/s, the shared 128K prefix (which passed at five by a fraction of a token/s) and the agent turns (no level of five was run; median 40 at six, less than one token/s above the objective) also pass at six streams, while the shared 256K prefix failed at five, just below the objective, and passed at six. These margins are within the noise of single-repetition levels, so we do not read them as a real capacity of six. At four agent-turn streams each gets a median of 59 tokens/s (p5 52), TTFT p95 is 2.34s and the pod produces 168 output tokens/s in total; at eight the median falls to 31. Context length barely matters once the context is cached: the 256K-token regimes are no slower than the short one at equal N; decode time is dominated by the batch, not by attention over the cached context, and five independent 128K contexts, the most run, occupy at most 48% of the KV pool. Independent long contexts did not hit the KV wall that vLLM’s own estimate predicted (Section 8.2): five independent 128K contexts and three of 256K ran with at most 56% KV-cache use (the session’s maximum, 62%, came from eight agent-turn streams) and 0 preemptions, so for this model the estimate is conservative. Over the whole session GPU memory stayed between 86.2 and 90.6 GiB per GPU and host anonymous plus shared memory peaked at 214 GiB, while PCIe reads grew with load (Table 22).

From streams to seats. Developers are not generating all the time. If every output token of the profile were served at exactly 40 tokens/s and every new token prefilled at the rate E12 measured for a 128K prompt (7,152 tokens/s), the developer would be generating for 2.8 hours a week, which is 1.7% of the week. That is

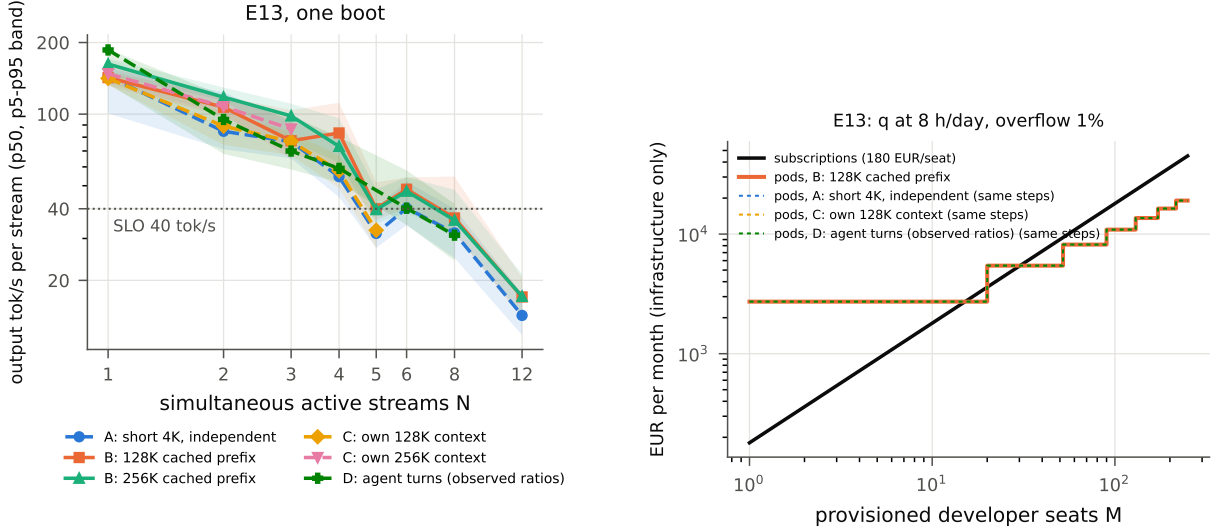


Figure 8: Coding-agent case study, E13 (one boot). Left: median output rate per stream against simultaneous streams (bands: p5 to p95 of requests), with the 40 tokens/s objective; dashed lines are independent contexts. Right: monthly cost of subscriptions and of the pods needed for M provisioned seats, the larger of the volume and the concurrency requirement (capacity with a 10% margin, 50% utilization, duty cycle for an 8-hour window, overflow probability 1%); infrastructure only. The four pod curves coincide because every regime has the same provisioning capacity.

an upper bound, since faster service shortens it. The profile does not say when in the day those hours fall; spread over an 8-hour window on each of the 5 active days, the probability that the developer is generating at a random instant of that window is $q = 7.1\%$ (14.1% for 4 hours, 2.4% for 24). With M developers active independently with the same q , the number generating at once is approximately $\text{Binomial}(M, q)$, and one pod can be provisioned for the largest M with $P(\text{Binomial}(M, q) > C) \leq \varepsilon$. For agent turns ($C = 4$) this is 19 seats at $\varepsilon = 1\%$ and 28 at 5% (Tables 23 and 24). Exceeding C does not fail requests; the streams of that moment fall below 40 tokens/s. The Binomial model ignores correlation between developers (shared working hours, deadlines) and burstiness inside a session, both of which raise the tail, so these numbers are optimistic for a team on one schedule; a trace of request times would be needed for a better arrival model.

By generation volume alone the same pod looks far larger: at the aggregate rate measured at four agent-turn streams its monthly output equals 262 developers of this profile at full utilization and 131 at 50%. That gap is the central result of this case study: *the pod has the monthly volume for several times more intermittent developers than it can give 40 tokens/s to at the same moment, and the concurrency limit, not the volume, decides how many seats it can serve.*

Cost. At 3.73 EUR per hour (the quoted USD price at the ECB rate of Section 9) a pod running 24/7 costs 2,723 EUR per month, as much as 15.1 subscriptions of 180 EUR. A single pod provisioned within its concurrency limit (19 seats) therefore lands just above break-even; the next seat needs a second pod, and 20 seats cost 1,847 EUR per month more than subscriptions. Pooling helps, because the sum of independent demands fluctuates less than each one: with load balanced across pods, 100 seats need 4 pods (39% below the subscription cost) and 250 seats 7 pods (58% below), before engineering and operations (Table 25, Figure 8 right). The volume capacity is never the limit: the pod breaks even at 5.8% of it. Stopping the pod outside a working window of 264 hours per month lowers its cost to 985 EUR (5.5 seats), but each start pays boot and weight loading (3.2 to 8.4 minutes per server boot in our sessions) and needs the 309 GiB pack on persistent storage. Engineering and operations move break-even substantially (Table 26); those overheads are assumptions, not measurements.

Beyond cost: sovereignty, privacy and security. Price is not the only reason to serve a coding agent’s model in-house. With open weights the organization decides which model and version run, when they change, and how long prompts and outputs are kept; the source code, secrets and internal documents that a coding agent reads by design (the profile above re-reads 234 context tokens per generated token) are not sent to a model provider, and the deployment can be placed in a jurisdiction of the organization’s choice, on its own hardware or with a provider bound by contract. These are benefits of control and digital sovereignty that

regime	N	rate p5	p50	p95	TPOT p50	p95	TTFT p50	p95	err %	aggr.
short	1	101	144	156	7.0	10.1	0.66	18.22	0	38
short	2	71	85	124	11.9	14.1	0.68	1.03	0	84
short	3	65	77	93	13.1	15.3	0.72	0.82	0	115
short	4	45	55	80	18.5	22.4	0.73	1.15	0	131
short	5	27	32	41	31.7	36.6	0.85	1.39	0	110
short	6	34	40	52	24.9	29.4	0.85	1.38	0	157
short	8	25	32	40	31.6	40.2	0.90	1.65	0	147
short	12	12	14	18	70.2	85.0	1.53	2.61	0	120
cached128k	1	132	142	152	7.0	7.6	0.61	0.63	0	118
cached128k	2	98	107	119	9.3	10.2	0.62	0.68	0	141
cached128k	3	69	77	104	12.9	14.6	0.72	0.80	0	145
cached128k	4	58	83	112	12.1	17.3	0.64	0.84	0	135
cached128k	5	36	40	52	25.0	27.8	0.74	0.88	0	155
cached128k	6	40	48	54	20.7	24.8	0.79	0.87	0	176
cached128k	8	32	37	55	27.3	31.8	0.84	1.00	0	178
cached128k	12	14	17	21	58.8	72.3	1.04	1.50	0	146
cached256k	1	146	162	171	6.2	6.8	0.96	1.02	0	113
cached256k	2	114	118	129	8.5	8.8	0.99	1.00	0	128
cached256k	3	77	98	111	10.2	13.1	1.02	1.17	0	155
cached256k	4	62	73	96	13.7	16.0	1.04	1.30	0	155
cached256k	5	35	40	48	25.2	28.8	1.08	1.53	0	131
cached256k	6	38	47	54	21.1	26.0	1.09	1.34	0	166
cached256k	8	32	36	48	27.9	31.7	1.34	1.67	0	212
cached256k	12	14	17	21	58.5	70.5	1.31	1.57	0	132
own128k	1	132	141	151	7.1	7.6	0.78	0.82	0	103
own128k	2	74	89	103	5.7	13.0	0.76	1.12	0	121
own128k	3	67	77	84	13.0	15.0	0.93	1.37	0	174
own128k	4	49	58	67	17.2	20.5	0.94	1.72	0	172
own128k	5	27	32	40	27.3	35.6	1.00	2.07	0	120
own256k	1	135	148	161	3.1	5.9	1.05	1.06	0	99
own256k	2	100	108	118	9.3	10.0	1.06	1.39	0	147
own256k	3	65	87	95	11.6	15.7	1.23	1.76	0	155
agent	1	134	186	188	5.4	7.6	0.91	0.94	0	133
agent	2	68	95	119	9.8	14.8	0.91	1.47	0	123
agent	3	59	70	98	14.3	17.2	0.97	2.08	0	146
agent	4	52	59	80	16.9	19.5	1.24	2.34	0	168
agent	6	34	40	58	24.9	29.3	1.37	2.99	0	178
agent	8	24	31	42	31.7	41.3	1.57	4.22	0	186

Table 21: Controlled: E13 per level, part 1 of 2, service per stream. Per-stream rate p5/p50/p95 (tokens/s, tokens over end of stream minus first token), TPOT p50/p95 (ms), TTFT p50/p95 (s), failed requests (%) and aggregate output (tokens/s). Resources at the same levels: Table 22.

the cost tables do not price. They come with two qualifications. First, they hold fully only on hardware the organization controls: on a rented pod, as in our measurements, data still reside on a third party’s machines, and the protection then rests on that provider’s isolation and contractual terms. Second, self-hosting moves responsibility rather than removing it: access control, patching, logging, encryption and incident response become the operator’s work, which is part of the operations overhead of Table 26. We did not measure any security property; this paragraph states what the deployment model allows, not what our setup guarantees.

What the data support. Supported, on one boot of one pod: four simultaneous coding-agent streams at 40 tokens/s or more each, whether the context is short, a shared cached 256K-token prefix, an independent cached 128K-token context or agent turns with the observed token ratios; and a monthly generation volume for several times more developers of the observed profile than that. Derived under stated assumptions (independent developers, an 8-hour window, the reply length): about 19 provisioned seats per pod at 1% overflow, which is at the break-even of subscriptions for one pod and below their cost for large pooled deployments. Not supported: any number across pods (Section 8.2 found a large pod-to-pod difference), a team’s real arrival pattern, and any claim that self-hosting replaces a commercial coding agent. The defensible summary is: *under the observed coding-agent workload, one offloaded 2x96 GB server gives four simultaneous streams at least 40 tokens/s each, which supports about as many intermittently active developer seats as its price buys in*

regime	N	GPU GiB	host GiB	PCIe GB/s	KV %	hit %	preempt.	bursts
short	1	86.2	214	15.09	3	0	0	0
short	2	86.3	214	38.12	4	0	0	0
short	3	86.4	214	55.41	4	0	0	0
short	4	87.6	214	57.15	7	0	0	0
short	5	87.6	214	80.81	5	0	0	0
short	6	87.6	214	72.06	8	0	0	0
short	8	87.6	214	75.27	9	0	0	0
short	12	89.9	214	91.26	17	0	0	0
cached128k	1	90.0	214	26.47	10	100	0	0
cached128k	2	90.0	214	36.93	10	100	0	0
cached128k	3	90.0	214	52.07	10	100	0	0
cached128k	4	90.0	214	57.82	10	100	0	0
cached128k	5	90.0	214	78.94	10	100	0	0
cached128k	6	90.6	214	71.71	11	100	0	0
cached128k	8	90.0	214	78.56	11	100	0	0
cached128k	12	90.0	214	90.66	12	100	0	0
cached256k	1	90.1	214	27.92	19	100	0	0
cached256k	2	90.1	214	42.66	19	100	0	0
cached256k	3	90.1	214	50.80	19	100	0	0
cached256k	4	90.1	214	55.81	19	100	0	0
cached256k	5	90.1	214	75.80	19	100	0	0
cached256k	6	90.1	214	72.85	20	100	0	0
cached256k	8	90.1	214	76.37	20	100	0	0
cached256k	12	90.1	214	88.75	21	100	0	0
own128k	1	90.1	214	50.15	10	100	0	0
own128k	2	90.1	214	70.71	19	100	0	2
own128k	3	90.1	214	65.54	29	100	0	0
own128k	4	90.1	214	72.83	39	100	0	1
own128k	5	90.1	214	84.21	48	100	0	1
own256k	1	90.1	214	49.08	19	100	0	1
own256k	2	90.1	214	55.53	38	100	0	0
own256k	3	90.1	214	59.58	56	100	0	0
agent	1	90.1	214	48.96	7	96	0	0
agent	2	90.1	214	65.56	17	96	0	1
agent	3	90.1	214	64.25	25	96	0	0
agent	4	90.1	214	75.86	34	96	0	0
agent	6	90.1	214	77.57	53	96	0	0
agent	8	90.1	214	81.58	62	96	0	0

Table 22: Controlled: E13 per level, part 2 of 2, resources. Peak GPU memory per GPU (GiB), peak host anon+shmem (GiB), mean PCIe RX summed over GPUs (GB/s), peak KV-cache use (%), prefix-cache hit rate (%), preemptions, and requests whose reply arrived in one late burst (their TPOT is invalid; the per-stream rate of Table 21 stays valid).

subscriptions; larger pooled deployments are cheaper per seat, and the simultaneous-stream limit, not the token volume, sets the number of seats.

regime	C	aggr.	vol. 100%	vol. 70%	vol. 50%	vol. 30%	conc. 1%	conc. 5%	b.e. util. %
A: short 4K, independent	4	131	204	143	102	61	19	28	7.4
B: 128K cached prefix	4	135	210	147	105	63	19	28	7.2
B: 256K cached prefix	4	155	242	169	121	73	19	28	6.3
C: own 128K context	4	172	268	188	134	81	19	28	5.6
C: own 256K context	≥ 3	155	242	170	121	73	12	20	6.2
D: agent turns (observed ratios)	4	168	262	183	131	78	19	28	5.8

Table 23: Two capacities of one pod, provisioned on C with a 10% margin (44 tokens/s). Volume: capacity-equivalent developer seats by observed monthly generation volume, at the aggregate output rate measured at that level and the stated utilization (these are NOT concurrent users). Concurrency: largest number of provisioned seats M with $P(\text{Binomial}(M, q) > C)$ at most 1% or 5%, with q the duty cycle for an 8-hour active window. Last column: pod utilization at which the seats it serves would cost as much in subscriptions as the pod.

active h/day	q %	short	cached128k	cached256k	own128k	own256k	agent
4	14.1	10	10	10	10	7	10
8	7.1	19	19	19	19	12	19
12	4.7	28	28	28	28	18	28
24	2.4	55	55	55	55	36	55

Table 24: Duty-cycle sensitivity. q is one developer’s generating time (output at 40 tokens/s plus prefill of new tokens, from the usage sample) over an active window of the given hours on each of the 5 active days; cells are the largest M with $P(\text{Binomial}(M, q) > C)$ at most 1%.

seats	subs. EUR	pods cached128k	saving	pods short	saving	pods own128k	saving	pods agent	saving
10	1,800	1 (c)	-923	1 (c)	-923	1 (c)	-923	1 (c)	-923
15	2,700	1 (c)	-23	1 (c)	-23	1 (c)	-23	1 (c)	-23
20	3,600	2 (c)	-1,847	2 (c)	-1,847	2 (c)	-1,847	2 (c)	-1,847
25	4,500	2 (c)	-947	2 (c)	-947	2 (c)	-947	2 (c)	-947
50	9,000	2 (c)	3,553	2 (c)	3,553	2 (c)	3,553	2 (c)	3,553
100	18,000	4 (c)	7,106	4 (c)	7,106	4 (c)	7,106	4 (c)	7,106
250	45,000	7 (c)	25,936	7 (c)	25,936	7 (c)	25,936	7 (c)	25,936

Table 25: Subscriptions against self-hosted pods, infrastructure only (no engineering or operations). Pods per regime are the larger of the volume requirement (50% utilization) and the concurrency requirement ($P(\text{demand} > \text{pods} \times C)$ at most 1%, 8-hour window); v or c marks which binds. Savings are subscription cost minus pod cost per month, EUR; negative means self-hosting costs more.

ops EUR/month	24/7 EUR	break-even	scheduled EUR	break-even
0	2,723	15.1	985	5.5
500	3,223	17.9	1,485	8.2
1,500	4,223	23.5	2,485	13.8
4,000	6,723	37.4	4,985	27.7

Table 26: Operational overhead sensitivity for one pod: monthly cost and break-even subscription seats, 24/7 and scheduled (264 h/month). Overhead values are assumptions, not measurements.

11 Quality

We do not assume that quality is preserved. Routed experts at 3 bits per weight, int4 Engram tables and an FP8 lm head are aggressive choices, and MoE models can react to quantization unevenly across experts [56]. Our only accuracy evidence is E10 and its reference E11, below: three small task subsets with thinking off. Before them, the exploratory runs only checked a few answers by hand and one tool call (`data/exploratory/.../docs/REPORT_vllm_excerpt.md`). Diffbot’s model card reports a KL-divergence comparison against the original model under AtomicChat’s protocol [4] and small GSM8K and HumanEval probes [29]; we cite them as third-party claims and do not reuse their numbers.

The quality protocol of Section 6 (E10 for the measured stack, E11 for the reference; in our runs, fixed-seed subsets of GSM8K, IFEval and HumanEval, with the official API as reference) is implemented in `quality/` and its analysis in `src/hostoffload/analysis/quality.py`. Both systems are queried through the same local proxy (`quality/api_proxy.py`). Tasks such as `humaneval_instruct` end with a partial assistant message that the model must continue; plain chat completion opens a new turn instead, and a first API run that did so scored 0 on every HumanEval item. The proxy therefore asks both backends to continue the message (DeepSeek’s chat prefix completion; vLLM’s `continue_final_message`); the invalid run is kept, marked as such, in `data/controlled/E11-quality-reference/api/`. On the vLLM side the same request shape failed differently: vLLM’s DeepSeek-V4.1 tokenizer ignores `continue_final_message` and closes the prefilled message with the end-of-sentence token, so the model generated after it, and a first E10 run scored 0.244 on HumanEval. The proxy now renders the prompt with vLLM’s own encoder, drops that token and continues it as a plain completion; that run is also kept as invalid, and E10 was repeated on the same pod.

On the same items the EXL3 stack with the `gen30` plan matches the reference on GSM8K (0.9600 against 0.9600, the same outcome on every item), and the differences on IFEval (+0.013, 95% interval -0.013 to +0.047) and HumanEval (-0.018, -0.049 to +0.012) have intervals that include zero, with 4.0% and 4.3% of the items changing outcome (Table 27). The subsets have 150 to 164 items, so differences of a few points cannot be excluded, and the reference is DeepSeek’s own service, whose precision and serving code we cannot see: the comparison bounds the quality cost of the whole measured stack against the vendor’s service, not of quantization alone. Tables 28 and 29 give the per-system scores with their standard errors. The LiveBench category scores of Section 2 (coding, instruction following) are third-party results for the unquantized model with thinking on, on different benchmarks and scales, and are not comparable with the HumanEval and IFEval subsets here.

task	metric	n	E10	E11	diff.	95% CI	flips %
gsm8k_cot_llama	exact_match	150	0.960	0.960	+0.000	[+0.000, +0.000]	0.0
humaneval_instruct	pass@1	164	0.963	0.982	-0.018	[-0.049, +0.012]	4.3
ifeval	prompt_level_strict_acc	150	0.867	0.853	+0.013	[-0.013, +0.047]	4.0

Table 27: Controlled: EXL3 stack (E10) against the API reference (E11) on the same items. Difference E10 minus E11 with a paired percentile bootstrap 95% interval (5000 resamples); flips are items whose outcome differs. The reference is a different serving stack, so a difference is not attributable to quantization alone.

run	task	metric	value	stderr
EXL3 stack, gen30 plan	gsm8k_cot_llama	exact_match	0.9600	0.0161
EXL3 stack, gen30 plan	humaneval_instruct	pass@1	0.9634	0.0147
EXL3 stack, gen30 plan	ifeval	prompt_level_strict_acc	0.8667	0.0278

Table 28: Controlled: Quality of the EXL3 stack (lm-evaluation-harness, pinned). lm-evaluation-harness 0.4.13, greedy, thinking off.

run	task	metric	value	stderr
DeepSeek API (deepseek-flash)	gsm8k_cot_llama	exact_match	0.9600	0.0161
DeepSeek API (deepseek-flash)	humaneval_instruct	pass@1	0.9817	0.0105
DeepSeek API (deepseek-flash)	ifeval	prompt_level_strict_acc	0.8533	0.0290

Table 29: Controlled: Quality reference: official DeepSeek API, model deepseek-flash (DeepSeek-V4.1-Flash), same subsets. lm-evaluation-harness 0.4.13, greedy, thinking off.

12 Discussion

What the code makes possible. The static data and the controlled boots agree. On two 96 GB GPUs, the original checkpoint cannot be GPU-resident (284.3 GiB outside its Engram tables), and the 3-bit pack cannot either without moving experts out (102.1 GiB per GPU against 95.6 GiB). Quantization alone shrinks the routed experts from 268.95 to 190.48 GiB, and the Engram tables from 188.83 to 103.00 GiB; host offload of about a third of the experts closes the remaining gap. The offload therefore turns an infeasible configuration into a feasible one; what it costs relative to a configuration that holds everything in VRAM can only be measured on larger hardware (E9) or by extrapolating the host-share curve (E3; at the larger of the two shares measured, the scenario A rate is lower at every level). Upstream vLLM’s zero-copy offload is not a drop-in alternative on these pins: without Diffbot’s hooks it cannot serve the model at any KV block size, so whether the per-expert split is also faster remains open.

The practical frontier. The controlled run confirms the shape the exploratory sweep suggested: with a cached prefix aggregate throughput keeps rising up to $N = 64$ while per-request decode falls steeply, so the operating point is set by the latency objective, not by capacity; with unique prompts the aggregate rate stops growing after $N = 32$ and TTFT tails grow instead. The SLO column of Table 12 shows where the objectives stop holding: with the cached prefix the interactive objective holds at low and moderate concurrency and the relaxed one up to higher levels, while with unique prompts both hold over a narrower range and none holds at the highest levels. The exploratory sweep, analyzed with the same objectives, placed the interactive limit of scenario A at $N = 2$ and the relaxed one at $N = 8$ (Section 9). For coding agents the binding criterion is stricter, a per-stream rate of 40 tokens/s, and E13 places it at four simultaneous streams in every context regime tested that far (independent 256K contexts were run only up to three; Section 10). The memory pools are preallocated and did not move with load, and the KV pool was never the limit at these prompt sizes, nor in E13 at 128K and 256K tokens. Which resource binds is not attributed: PCIe reads of cold experts, the speculative verification batch, prefill interference, or the eager execution of decode steps above the largest CUDA graph; the ablations that would separate them did not run (Section 6.8).

Practical guidance. For someone deploying this stack, the measurements suggest four rules, each from a single boot on a specific pod and therefore a starting point rather than a guarantee. First, check where the two GPUs sit before measuring or serving: the same configuration had a median TPOT 1.17 to 1.51 times higher on a host whose GPUs were on different NUMA nodes (`nvidia-smi topo -m` reports SYS instead of NODE; Table 7). Second, keep long contexts cached: once a 128K or 256K-token context is in the prefix cache, four streams keep a median of 58 and 73 tokens/s with TTFT p95 under two seconds, while a fresh 128K-token prompt costs 18.33s of prefill before its first token. Third, size interactive capacity in simultaneous streams, not tokens per month: about 4 streams at 40 tokens/s per pod for agent turns, which under the observed usage profile serve about 19 intermittently active developers (Section 10). Fourth, do not expect upstream vLLM’s UVA offload to replace Diffbot’s hooks on these pins: it does not serve the model at all.

Static placement. The plans are static and profile-derived. The recipe’s own notes report that plans tuned on one traffic type do worse on another [29]; recent work argues that static placement is suboptimal in general [44, 98, 101]. E4 would measure this sensitivity for the measured stack, with a random plan as a floor and plans fitted on a calibration workload disjoint from the benchmark prompts; it did not run (Section 6.8), so our data say nothing about how much the choice of plan matters beyond the two nested plans of E3.

13 Limitations and Threats to Validity

The central limitation. The controlled evidence is three pod sessions on three different hosts (101, 43 and 57 minutes; Table 7): each experiment’s measurements come from a single boot, and although the `gen30` configuration was booted on all three pods, no measurement was repeated across boots, so the intervals cover repetitions inside a boot and nothing across boots. The one configuration measured on two hosts differed by far more than any interval inside a boot (Section 8.2), so pod placement is a large, uncontrolled factor. E8 has two repetitions per level in scenario B, but in scenario A only at $N = 1$: one client-side disconnect among eight requests tripped the pre-registered stop rule (error rate above 5%) and ended A’s second sweep. A rule evaluated on eight requests is too sensitive; it should require a minimum number of errors or a larger sample. After the first session we closed idle client connections before the server does (the cause of those disconnects) and added a minimum of two errors, which E12 and E13 used; session 2’s rerun of the E3 subset kept the first session’s rule to match its baseline, and the first session’s data were not re-run under the new rule.

What the measurement campaign does not answer. Our three pod sessions (Section 7) answer RQ1 on the measured hardware, give RQ2 two host shares with one repetition each, measure the RQ4 frontier for two scenarios, give RQ5 a first answer on small subsets, and answer RQ6 for one pod and six context regimes with one repetition (E13, Section 10). They leave open: (i) the speed of the model fully in VRAM: without the 2x B300 pod (E9) the cost of offload is bounded only by the slope between two host shares, not against a no-offload baseline; this is future work; (ii) RQ3 as a speed comparison: without Diffbot’s hooks the UVA variant cannot serve DeepSeek-V4.1 on these pins at any KV block size (64 fails at initialization, 128 on the first request), so the cost of the per-expert split against UVA is unknown; (iii) a quality reference on our own hardware and serving code: the API stands in for E11, and differences may come from the API’s precision, prompt handling or sampling rather than from quantization; (iv) upstream vLLM with the original checkpoint on the same GPUs (dropped to avoid its 475 GiB download); (v) plans fitted to routing counts at several shares and the smallest share that fits; the decode scenario at the larger share, lost to a client-side disconnect; (vi) placement policy (E4), Engram format (E5), side-stream SM budget (E6) and speculation off (E7); (vii) the prose scenario, Poisson arrivals, MATH, MBPP, RULER and LongBench; long context (E12) was measured with one repetition, on the slower second pod, and without a quality check at those lengths. Statements about these remain hypotheses.

Model capability. The LiveBench scores in Section 2 belong to the unquantized model at maximum reasoning effort as served by DeepSeek. How much of that capability survives 3-bit quantization and thinking disabled is not measured; our quality evidence covers three small task subsets with thinking off. Running LiveBench on the served stack, with thinking on and off, is future work.

Coding-agent case study. E13 is one repetition on one boot of one pod; levels near the objective pass or fail by noise, which is why we provision on a 10% margin, and its capacities do not transfer to another pod (Section 8.2). Replies have a fixed length of 400 tokens, and the agent turns are synthetic: unique corpus text stands in for tool output, so speculative decoding may accept more or fewer tokens than on real coding work. The usage profile is one developer’s week, read from a provider’s billing categories that need not match vLLM’s prefix-cache accounting; the reply length per turn and the daily active window are assumptions varied in the tables. The duty cycle uses the 40 tokens/s floor (an upper bound on busy time), and the Binomial model assumes independent developers with equal activity, which understates the tail for a team on one schedule. Costs are GPU rental at one quoted price and FX rate; storage, networking, engineering, operations and the difference in model capability and product features are not priced except in the stated sensitivity.

Exploratory data. The imported runs (i) were measured once per configuration; (ii) used usable bars chosen after seeing the data; (iii) started every level with a synchronized burst, which dominates TTFT tails; (iv) depend on content through speculative decoding, with synthetic prompts (vLLM source slices and question templates, or a few essays); and (v) never evaluated quality. They also timed chunks rather than tokens, computed percentiles by nearest rank on 8 to 128 requests, measured aggregate throughput including ramp-up and drain, and ran on one pod on one day.

Static accounting. File bytes are not runtime allocations: repacking, FP8 copies and workspaces change VRAM use (4.37 GiB per GPU in the exploratory boot). The placement rules model the documented configuration and are checked against the server log only where the log reports placement. The minimal host share is an estimate from one boot.

Protocol. The SLOs were registered before the controlled runs but after we saw the exploratory data. Cloud pods share hosts, so CPU, memory bandwidth and PCIe topology vary between pods; we record them but cannot control them, and across-boot repetitions were designed for one pod and were not run. Upstream vLLM’s UVA offload did fail on this model at the pinned commit, at both KV block sizes we tried; we report that failure as a result, but it leaves RQ3 unanswered as a speed comparison. The B300 reference was not run, so RQ4 has no all-in-VRAM comparison; when it runs, it will bound the cost of offload together with the hardware difference, because it uses other GPUs. GSM8K, HumanEval and similar benchmarks may be contaminated in training data; the comparison is between two versions of the same model, which limits but does not remove that concern. NVML’s PCIe counters are sampled and their semantics vary between GPUs, hence the calibration step.

Attribution. The audit relies on pinned public sources, keyword searches confirmed by reading code, and Hub and GitHub metadata. Diffbot’s code is attributed to the organization; individual authors are not visible. Work that is not public or not cited by the recipe could precede it.

14 Reproducibility Statement

Everything in this paper is produced from the artifact repository by documented commands:

- `make audit` re-runs the attribution audit (sources fetched at their pins outside the repository).
- `make static` re-measures the byte accounting and Hub metadata from public files; `make test` runs the test suite, including the no-fake-data guard; `make analyze figures paper` regenerates every table, figure and value and builds this PDF.
- `make stack-fetch` and `make pod-bootstrap VARIANT=...` build each variant from pinned inputs with recorded hashes; `make bench-En` and `make quality-En` run each experiment on a prepared pod; `make plan` regenerates the budget.

No number without a file. Every measured number in the text is printed by a macro that looks up a value written by the analysis pipeline, which refuses values without an existing source file, refuses controlled values computed from exploratory files, and records the sha256 of every source (`paper/generated/provenance.json`). Tables are included only for experiments with data: running one of the experiments of Section 6.8 (`make bench-En`) adds its table on the next `make analyze paper`, and `make paper STRICT=1` fails if any cited value lacks its source. A test fails if a decimal number appears in the paper sources outside these macros unless it is a whitelisted configuration constant with its source (`paper/literal_numbers.yaml`).

Licenses and redistribution. This repository contains no third-party source code: the recipe, the plugin and the kernels are fetched and patched at build time from pinned revisions, and vllm-exl3 and Diffbot’s patch to it are AGPL-3.0, so a built image that includes them carries AGPL obligations when distributed or served with modifications. Our code is Apache-2.0, compatible with vLLM’s license and combinable with the AGPL components in a built image; text and data are CC-BY-4.0. Model weights are never redistributed (`THIRD_PARTY.md`, `PROVENANCE.md`).

Archival. The repository is prepared for an archived release (`CITATION.cff`, `.zenodo.json`, `RELEASE_CHECKLIST.md`); publication and archival await the owner’s approval.

15 Conclusion

The host-memory offload that lets a 3-bit DeepSeek-V4.1-Flash serve on two 96 GB GPUs is third-party work: a per-expert, profile-driven refinement by Diffbot of a zero-copy mechanism that upstream vLLM already provides, on top of Victor Cruz’s vllm-exl3 plugin, which brings EXL3 to vLLM, and turboderp’s kernels. What we contribute is the evidence: who wrote what, a byte-level account of what fits where, a re-analysis of earlier single runs that marks them for what they are, a protocol, and controlled measurements from three pod sessions.

What we found.

- The configuration does not boot without the offload and serves with it.
- With a cached 16K-token prefix the stack reaches 340 output tokens/s at 64 concurrent requests, and 186 with unique 4K-token prompts at 32; moving more experts to the host costs throughput.
- Near the full context a cached prefix stays interactive, while distinct long prompts are bounded by prefill time.
- For coding agents, four simultaneous streams keep at least 40 tokens/s each in every context regime tested that far (independent 256K-token contexts only up to three).
- On small subsets the stack’s answers match DeepSeek’s own service within the uncertainty.
- Upstream UVA offload cannot serve the model without Diffbot’s hooks, and the same configuration ran markedly slower on a host with a different NUMA layout.

What it means. For interactive use the binding limit is concurrency, not token volume: one server has the monthly output for several times more intermittent developers than it can give 40 tokens/s to at the same moment, which places a single server near the break-even of per-seat subscriptions and pooled deployments below it. Where a pod lands matters as much as most settings we varied, so deployments should check the GPU topology of the host they get. Beyond cost, serving open weights in-house keeps code and prompts under the organization’s control, which supports digital sovereignty, privacy and the protection of company data; that benefit is complete only on hardware the organization controls and comes with the security work of operating it. The answer to “at what performance cost” is still partial: the price relative to a configuration that fits in VRAM remains to be measured.

Future work. The protocol is implemented end to end, so each open question is one run away (Section 6.8). First, the all-in-VRAM reference on two B300 GPUs (E9), which would turn the bounds on the cost of offload into a measured difference. Then speculative decoding off (E7), to separate DSpark’s contribution from the offload’s; placement policies, including plans fitted to a team’s own routing counts (E4); the precision of the Engram tables in host memory (E5); the SM budget of the host-expert side stream (E6); and a from-scratch rebuild of every variant (E1). Beyond the protocol: repetitions across boots and pods, to bound the pod-to-pod variation we observed; LiveBench on the served stack with thinking on and off, to measure how much of the model’s capability survives 3-bit quantization; and request traces from a real team, to replace the Binomial model of concurrent demand.

Acknowledgments

This paper measures open work by others. We thank Victor Cruz, whose vllm-exl3 plugin brings EXL3 to vLLM and is the base on which the host-memory offload is built; turboderp for ExLlamaV3 and the EXL3 format; Diffbot for the serving recipe, the host-expert patch and its kernels; coolbho3k for the 3-bit EXL3 checkpoint; sfxnz for the recipe helpers; Mia’s AI Lab for the work vllm-exl3 derives from; the vLLM project and the authors of the pull requests the recipe carries; and DeepSeek for releasing the model’s weights. None of them took part in this study, and any error is ours.

References

- [1] Amey Agrawal, Anmol Agarwal, Nitin Kedia, Jayashree Mohan, Souvik Kundu, Nipun Kwatra, Ramachandran Ramjee, and Alexey Tumanov. Etalon: Holistic performance evaluation framework for LLM inference systems, 2024. URL <https://arxiv.org/abs/2407.07000>. arXiv preprint; v1 was titled Metron: Holistic Performance Evaluation Framework for LLM Inference Systems.
- [2] 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)*, 2024. URL <https://www.usenix.org/conference/osdi24/presentation/agrawal>.
- [3] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. DeepSpeed-Inference: Enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2022. doi: 10.1109/SC41404.2022.00051. URL <https://doi.org/10.1109/SC41404.2022.00051>.
- [4] AtomicChat. DeepSeek-V4.1-Flash-NVFP4-metrics. Hugging Face dataset, 2026. URL <https://huggingface.co/datasets/AtomicChat/DeepSeek-V4.1-Flash-NVFP4-metrics>. Accessed 2026-10-06.
- [5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021. URL <https://arxiv.org/abs/2108.07732>.
- [6] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3119–3137, 2024. doi: 10.18653/v1/2024.acl-long.172. URL <https://aclanthology.org/2024.acl-long.172/>.

- [7] Stella Biderman, Hailey Schoelkopf, Lintang Sutawika, et al. Lessons from the trenches on reproducible evaluation of language models. arXiv preprint arXiv:2405.14782, 2024. URL <https://arxiv.org/abs/2405.14782>.
- [8] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 5209–5235, 2024. URL <https://proceedings.mlr.press/v235/cai24b.html>.
- [9] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. MoE-Lightning: High-throughput MoE inference on memory-constrained GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*, pages 715–730, 2025. doi: 10.1145/3669940.3707267. URL <https://doi.org/10.1145/3669940.3707267>.
- [10] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. QuIP: 2-bit quantization of large language models with guarantees. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/0df38cd13520747e1e64e5b123a78ef8-Abstract-Conference.html.
- [11] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating Large Language Model Decoding with Speculative Sampling, 2023. URL <https://arxiv.org/abs/2302.01318>.
- [12] Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, Yuening Zhu, Qingliang Ou, Jiaqi Liao, Xianglin Chen, Zhiyuan Ai, Yongwei Wu, and Mingxing Zhang. KTransformers: Unleashing the full potential of CPU/GPU hybrid inference for MoE models. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*, pages 1014–1029, 2025. doi: 10.1145/3731569.3764843. URL <https://doi.org/10.1145/3731569.3764843>.
- [13] Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021. URL <https://arxiv.org/abs/2107.03374>.
- [14] Xin Cheng, Rui Tian, Wangding Zeng, et al. Conditional memory via scalable lookup: A new axis of sparsity for large language models. arXiv preprint arXiv:2601.07372, 2026. URL <https://arxiv.org/abs/2601.07372>. Extended version (v2) that names the module Engram; a conference version appeared at ACL 2026.
- [15] Xin Cheng, Xingkai Yu, Chenze Shao, et al. DSpark: Confidence-scheduled speculative decoding with semi-autoregressive generation, 2026. URL <https://arxiv.org/abs/2607.05147>.
- [16] Xin Cheng, Wangding Zeng, Damai Dai, Qinyu Chen, Bingxuan Wang, Zhenda Xie, Kezhao Huang, Xingkai Yu, Zhewen Hao, Han Zhang, Yu-Kun Li, Huishuai Zhang, Dongyan Zhao, and Wenfeng Liang. Conditional memory via scalable lookup: A new axis of sparsity for large language models. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4968–4990, San Diego, California, United States, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.226. URL <https://aclanthology.org/2026.acl-long.226/>. Conference version of arXiv:2601.07372; names the module Deep Sparse Embedding (DSE).
- [17] 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. arXiv preprint arXiv:2110.14168, 2021. URL <https://arxiv.org/abs/2110.14168>.
- [18] coolbho3k. DeepSeek-V4.1-Flash-EXL3-3bpw. Hugging Face model card, 2026. URL <https://huggingface.co/coolbho3k/DeepSeek-V4.1-Flash-EXL3-3bpw>. Accessed 2026-10-06.
- [19] Victor Cruz. vllm-exl3: An out-of-tree vLLM plugin for EXL3 quantized models. GitHub repository, 2026. URL <https://github.com/vcruz305/vllm-exl3>. GitHub user vcruz305. Accessed 2026-10-06.

- [20] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. DeepSeekMoE: Towards ultimate expert specialization in Mixture-of-Experts language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1280–1297, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.70. URL <https://aclanthology.org/2024.acl-long.70/>.
- [21] DeepSeek-AI. DeepSeek-V3.2-Exp: Boosting long-context efficiency with DeepSeek sparse attention. Hugging Face model card, 2025. URL <https://huggingface.co/deepseek-ai/DeepSeek-V3.2-Exp>. Accessed 2026-10-06.
- [22] DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression. Hugging Face model card and technical report (DeepSeek_V41_Tech_Report.pdf), 2026. URL <https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash>. Accessed 2026-10-06.
- [23] DeepSeek-AI et al. DeepSeek-V2: A strong, economical, and efficient Mixture-of-Experts language model, 2024. URL <https://arxiv.org/abs/2405.04434>.
- [24] DeepSeek-AI et al. DeepSeek-V3 technical report, 2024. URL <https://arxiv.org/abs/2412.19437>.
- [25] DeepSeek-AI et al. DeepSeek-V3.2: Pushing the frontier of open large language models, 2025. URL <https://arxiv.org/abs/2512.02556>.
- [26] DeepSeek-AI et al. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL <https://arxiv.org/abs/2606.19348>.
- [27] DeepSeek-AI et al. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression, 2026. URL <https://arxiv.org/abs/2609.19969>.
- [28] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/c3ba4962c05c49636d4c6206a97e9c8a-Abstract-Conference.html. Listed in the NeurIPS proceedings index as “GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale”.
- [29] diffbot. DeepSeek-V4.1-Flash-EXL3-3bpw-2x-RTX-PRO-6000. Hugging Face model card, 2026. URL <https://huggingface.co/diffbot/DeepSeek-V4.1-Flash-EXL3-3bpw-2x-RTX-PRO-6000>. Accessed 2026-10-06.
- [30] Zhixu Du, Shiyu Li, Yuhao Wu, Xiangyu Jiang, Jingwei Sun, Qilin Zheng, Yongkai Wu, Ang Li, Hai Helen Li, and Yiran Chen. SiDA: Sparsity-inspired data-aware serving for efficient and scalable large mixture-of-experts models. In *Proceedings of Machine Learning and Systems (ML-Sys)*, volume 6, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/698cfaf72a208aef2e78bcac55b74328-Abstract-Conference.html.
- [31] Haojie Duanmu, Xiuhong Li, Zhihang Yuan, Size Zheng, Jiangfei Duan, Xingcheng Zhang, and Dahua Lin. MxMoE: Mixed-precision quantization for MoE with accuracy and performance co-design. In *Proceedings of the 42nd International Conference on Machine Learning (ICML 2025)*, volume 267 of *Proceedings of Machine Learning Research*, pages 14793–14806. PMLR, 2025. URL <https://proceedings.mlr.press/v267/duanmu25a.html>.
- [32] Abhinav Dutta, Sanjeev Krishnan, Nipun Kwatra, and Ramachandran Ramjee. Accuracy is not all you need. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. URL https://papers.nips.cc/paper_files/paper/2024/hash/e0e956681b04ac126679e8c7dd706b2e-Abstract-Conference.html.
- [33] B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979. doi: 10.1214/aos/1176344552. URL <https://projecteuclid.org/journals/annals-of-statistics/volume-7/issue-1/Bootstrap-Methods-Another-Look-at-the-Jackknife/10.1214/aos/1176344552.full>.
- [34] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Monographs on Statistics and Applied Probability. Chapman & Hall, 1994. doi: 10.1201/9780429246593. URL <https://www.routledge.com/An-Introduction-to-the-Bootstrap/Efron-Tibshirani/p/book/9780412042317>.

- [35] Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, volume 235 of *Proceedings of Machine Learning Research*, pages 12284–12303. PMLR, 2024. URL <https://proceedings.mlr.press/v235/egiazarian24a.html>.
- [36] Artyom Eliseev and Denis Mazur. Fast inference of mixture-of-experts language models with offloading. arXiv preprint arXiv:2312.17238, 2023. URL <https://arxiv.org/abs/2312.17238>. Technical report.
- [37] Zhiyuan Fang, Zicong Hong, Yuegui Huang, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. Fate: Fast edge inference of mixture-of-experts models via cross-layer gate. arXiv preprint arXiv:2502.12224, 2025. URL <https://arxiv.org/abs/2502.12224>.
- [38] Zhiyuan Fang, Yuegui Huang, Zicong Hong, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. Klotski: Efficient mixture-of-expert inference via expert-aware multi-batch pipeline. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS)*, 2025. doi: 10.1145/3676641.3716261. URL <https://arxiv.org/abs/2502.06888>.
- [39] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL <https://jmlr.org/papers/v23/21-0998.html>.
- [40] FlashInfer Contributors. FlashInfer: High-performance GPU kernels for inference. GitHub repository, 2026. URL <https://github.com/flashinfer-ai/flashinfer>. Accessed 2026-10-06.
- [41] Elias Frantar and Dan Alistarh. QMoE: Sub-1-bit compression of trillion parameter models. In *Proceedings of Machine Learning and Systems 6 (MLSys 2024)*, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/c74b624843218d9b6713fcf299d6d5e4-Abstract-Conference.html. arXiv title: “QMoE: Practical Sub-1-Bit Compression of Trillion-Parameter Models”.
- [42] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *International Conference on Learning Representations (ICLR 2023)*, 2023. URL <https://iclr.cc/virtual/2023/poster/10855>. Widely known as GPTQ; arXiv title: “GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers”.
- [43] Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. Better & Faster Large Language Models via Multi-token Prediction. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 15706–15734, 2024. URL <https://proceedings.mlr.press/v235/gloeckle24a.html>.
- [44] Xin He, Shunkang Zhang, Kaijie Tang, Shaohuai Shi, Yuxin Wang, Zihao Zeng, Zhenheng Tang, Xiaowen Chu, Haiyan Yin, Ivor W. Tsang, and Yew Soon Ong. ExpertFlow: Efficient mixture-of-experts inference via predictive expert caching and token scheduling. In *Proceedings of the 63rd ACM/IEEE Design Automation Conference (DAC)*, 2026. doi: 10.1145/3770743.3804292. URL <https://arxiv.org/abs/2410.17954>.
- [45] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1 (NeurIPS Datasets and Benchmarks 2021)*, 2021. URL <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/be83ab3ecd0db773eb2dc1b0a17836a1-Abstract-round2.html>.
- [46] Cheng-Ping Hsieh, Simeng Sun, Samuel Krman, Shantanu Acharya, Dima Rekeshe, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What’s the real context size of your long-context language models? In *Conference on Language Modeling (COLM)*, 2024. URL <https://arxiv.org/abs/2404.06654>.
- [47] Beichen Huang, Yueming Yuan, Zelei Shao, and Minjia Zhang. MiLo: Efficient quantized MoE inference with mixture of low-rank compensators. In *Proceedings of Machine Learning and Systems 7 (MLSys 2025)*, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/9032e5c9ec394ce768a2fa9bdc56af6c-Abstract-Conference.html.

- [48] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. Pre-gated MoE: An algorithm-system co-design for fast and scalable mixture-of-expert inference. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1018–1031, 2024. doi: 10.1109/ISCA59077.2024.00078. URL <https://doi.org/10.1109/ISCA59077.2024.00078>.
- [49] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/94074dd5a072d28ff75a76dabed43767-Abstract-Conference.html.
- [50] Keisuke Kamahori, Tian Tang, Yile Gu, Kan Zhu, and Baris Kasikci. Fiddler: CPU-GPU orchestration for fast inference of mixture-of-experts models. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. URL <https://iclr.cc/virtual/2025/poster/29900>.
- [51] Young Jin Kim, Raffy Fahim, and Hany Hassan Awadalla. Mixture of quantized experts (MoQE): Complementary effect of low-bit quantization and robustness, 2023. URL <https://arxiv.org/abs/2310.02410>.
- [52] Rui Kong, Yuanchun Li, Qingtian Feng, Weijun Wang, Xiaozhou Ye, Ye Ouyang, Linghe Kong, and Yunxin Liu. SwapMoE: Serving off-the-shelf MoE-based large language models with tunable memory budget. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6710–6720, 2024. doi: 10.18653/v1/2024.acl-long.363. URL <https://aclanthology.org/2024.acl-long.363/>.
- [53] 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 (SOSP)*, pages 611–626, 2023. doi: 10.1145/3600006.3613165. URL <https://arxiv.org/abs/2309.06180>.
- [54] Dmitry Lepikhin, Hyoungho Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations (ICLR)*, 2021. URL <https://iclr.cc/virtual/2021/poster/3196>.
- [55] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast Inference from Transformers via Speculative Decoding. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 19274–19286, 2023. URL <https://proceedings.mlr.press/v202/leviathan23a.html>.
- [56] Pingzhi Li, Xiaolong Jin, Zhen Tan, Yu Cheng, and Tianlong Chen. QuantMoE-Bench: Examining post-training quantization for mixture-of-experts, 2024. URL <https://arxiv.org/abs/2406.08155>.
- [57] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 28935–28948, 2024. URL <https://proceedings.mlr.press/v235/li24bt.html>.
- [58] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7421–7432, 2024. doi: 10.18653/v1/2024.emnlp-main.422. URL <https://aclanthology.org/2024.emnlp-main.422/>.
- [59] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test. In *Advances in Neural Information Processing Systems 38 (NeurIPS 2025)*, 2025. URL <https://neurips.cc/virtual/2025/poster/119930>.
- [60] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. In *Proceedings of Machine Learning and Systems 6 (MLSys 2024)*, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/42a452cbafa9dd64e9ba4aa95cc1ef21-Abstract-Conference.html.

- [61] Shouxu Lin, Zhiyuan Guo, and Jiaxin Lin. DAK: Direct-access-enabled GPU memory offloading with optimal efficiency for LLM inference. arXiv preprint arXiv:2604.26074, 2026. URL <https://arxiv.org/abs/2604.26074>.
- [62] Xiaoxuan Liu, Jongseok Park, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Chen Zhang, Kuntai Du, Xiangxi Mo, Kaichao You, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. TurboSpec: Closed-loop Speculation Control System for Optimizing LLM Serving Goodput, 2024. URL <https://arxiv.org/abs/2406.14066>.
- [63] LiveBench. LiveBench leaderboard, question set 2026-06-25. <https://livebench.ai/>, 2026. Data files downloaded 2026-10-07; stored in data/static/livebench/.
- [64] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS)*, pages 932–949, 2024. doi: 10.1145/3620666.3651335. URL <https://doi.org/10.1145/3620666.3651335>.
- [65] NVIDIA Corporation. CUDA programming guide, section 2.6: Unified and system memory. NVIDIA documentation, CUDA Programming Guide v13.4.2, 2026. URL <https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/understanding-memory.html>. Sections 2.6.1 (Unified Virtual Address Space) and 2.6.3.1 (Mapped Memory). Accessed 2026-10-06.
- [66] NVIDIA Corporation. Nsight Systems user guide. NVIDIA documentation, version 2026.5, 2026. URL <https://docs.nvidia.com/nsight-systems/UserGuide/index.html>. Accessed 2026-10-06.
- [67] NVIDIA Corporation. NVML API reference guide. NVIDIA documentation, 2026. URL <https://docs.nvidia.com/deploy/nvml-api/index.html>. Accessed 2026-10-06.
- [68] Pratyush Patel, Esha Chouksey, 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)*, pages 118–132, 2024. doi: 10.1109/ISCA59077.2024.00019. URL <https://arxiv.org/abs/2311.18677>.
- [69] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*, volume 162 of *Proceedings of Machine Learning Research*, pages 18332–18346. PMLR, 2022. URL <https://proceedings.mlr.press/v162/rajbhandari22a.html>.
- [70] Vijay Janapa Reddi, Christine Cheng, David Kanter, et al. MLPerf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020. doi: 10.1109/ISCA45697.2020.00045. URL <https://arxiv.org/abs/1911.02549>.
- [71] Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. MagicDec: Breaking the Latency-Throughput Tradeoff for Long Context Generation with Speculative Decoding. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/13f972adf12bdf886583d48cd528002f-Abstract-Conference.html.
- [72] sfxnz. DeepSeek-V4.1-Flash-EXL3-vLLM-2x-DGX-Spark. GitHub repository, 2026. URL <https://github.com/sfxnz/DeepSeek-V4.1-Flash-EXL3-vLLM-2x-DGX-Spark>. Accessed 2026-10-06.
- [73] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarczyk, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations (ICLR)*, 2017. URL <https://arxiv.org/abs/1701.06538>.
- [74] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR, 2023. URL <https://proceedings.mlr.press/v202/sheng23a.html>.

- [75] Xiaoni Song, Zihang Zhong, Rong Chen, and Haibo Chen. ProMoE: Fast MoE-based LLM serving using proactive caching. arXiv preprint arXiv:2410.22134, 2024. URL <https://arxiv.org/abs/2410.22134>.
- [76] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise Parallel Decoding for Deep Autoregressive Models. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, 2018. URL <https://proceedings.neurips.cc/paper/2018/hash/c4127b9194fe8562c64dc0f5bf2c93bc-Abstract.html>.
- [77] Lintang Sutawika, Leo Gao, Hailey Schoelkopf, et al. EleutherAI/lm-evaluation-harness: Major refactor. Zenodo, software release v0.4.0, 2023. URL <https://zenodo.org/records/10256836>. Accessed 2026-10-06.
- [78] Suraiya Tairin, Shohaib Mahmud, Haiying Shen, and Anand Iyer. eMoE: Task-aware memory efficient mixture-of-experts-based (MoE) model inference. arXiv preprint arXiv:2503.06823, 2025. URL <https://arxiv.org/abs/2503.06823>.
- [79] Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP#: Even better LLM quantization with Hadamard incoherence and lattice codebooks. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, volume 235 of *Proceedings of Machine Learning Research*, pages 48630–48656. PMLR, 2024. URL <https://proceedings.mlr.press/v235/tseng24a.html>.
- [80] Albert Tseng, Qingyao Sun, David Hou, and Christopher De Sa. QTIP: Quantization with trellises and incoherence processing. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. doi: 10.52202/079017-1904. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/6de2e84b8da47bb2eb5e2ac96c63d2b0-Abstract-Conference.html.
- [81] turboderp-org. ExLlamaV3: An optimized quantization and inference library for running LLMs locally on modern consumer-class GPUs. <https://github.com/turboderp-org/exllamav3>, 2026. Software, MIT license. EXL3 format is described in the README as a streamlined variant of QTIP. Accessed 2026-10-06.
- [82] vllm-project. vLLM: A high-throughput and memory-efficient inference and serving engine for LLMs. GitHub repository, 2026. URL <https://github.com/vllm-project/vllm>. Accessed 2026-10-06.
- [83] vLLM Project. Automatic prefix caching. vLLM documentation, 2026. URL https://docs.vllm.ai/en/latest/features/automatic_prefix_caching.html. Accessed 2026-10-06.
- [84] Lean Wang, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. Auxiliary-loss-free load balancing strategy for Mixture-of-Experts, 2024. URL <https://arxiv.org/abs/2408.15664>.
- [85] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Micah Goldblum, et al. LiveBench: A challenging, contamination-limited LLM benchmark. In *International Conference on Learning Representations (ICLR)*, 2025. URL <https://arxiv.org/abs/2406.19314>.
- [86] Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. Unlocking Efficiency in Large Language Model Inference: A Comprehensive Survey of Speculative Decoding. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 7655–7671, 2024. doi: 10.18653/v1/2024.findings-acl.456. URL <https://aclanthology.org/2024.findings-acl.456/>.
- [87] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 2023. URL <https://proceedings.mlr.press/v202/xiao23c.html>.
- [88] Tairan Xu, Leyang Xue, Zhan Lu, Adrian Jackson, and Luo Mai. MoE-Gen: High-throughput MoE inference on a single GPU with module-based batching. arXiv preprint arXiv:2503.09716, 2025. URL <https://arxiv.org/abs/2503.09716>.
- [89] Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. MoE-Infinity: Efficient MoE inference on personal machines with sparsity-aware expert cache. arXiv preprint arXiv:2401.14361, 2024. URL <https://arxiv.org/abs/2401.14361>.

- [90] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems (MLSys)*, 2025. URL <https://arxiv.org/abs/2501.01005>.
- [91] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. EdgeMoE: Empowering sparse large language models on mobile devices. *IEEE Transactions on Mobile Computing*, 2025. URL <https://arxiv.org/abs/2308.14352>.
- [92] 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)*, 2022. URL <https://www.usenix.org/conference/osdi22/presentation/yu>.
- [93] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23078–23097, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1126. URL <https://aclanthology.org/2025.acl-long.1126/>.
- [94] Yichao Yuan, Lin Ma, and Nishil Talati. MoE-Lens: Towards the hardware limit of high-throughput MoE LLM serving under resource constraints. arXiv preprint arXiv:2504.09345, 2025. URL <https://arxiv.org/abs/2504.09345>.
- [95] Yujie Zhang, Shivam Aggarwal, and Tulika Mitra. DAOP: Data-aware offloading and predictive pre-calculation for efficient MoE inference. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2025. URL <https://arxiv.org/abs/2501.10375>.
- [96] 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 *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.
- [97] Shuzhang Zhong, Ling Liang, Yuan Wang, Runsheng Wang, Ru Huang, and Meng Li. AdapMoE: Adaptive sensitivity-based expert gating and management for efficient MoE inference. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2024. doi: 10.1145/3676536.3676741. URL <https://arxiv.org/abs/2408.10284>.
- [98] Shuzhang Zhong, Yanfan Sun, Ling Liang, Runsheng Wang, Ru Huang, and Meng Li. HybriMoE: Hybrid CPU-GPU scheduling and cache management for efficient MoE inference. arXiv preprint arXiv:2504.05897, 2025. URL <https://arxiv.org/abs/2504.05897>.
- [99] 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)*, 2024. URL <https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin>.
- [100] Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. arXiv preprint arXiv:2311.07911, 2023. URL <https://arxiv.org/abs/2311.07911>.
- [101] Zeyu Zhu, Gang Li, Peisong Wang, Zitao Mo, Minnan Pei, Zhuoran Song, Xiaoyao Liang, and Jian Cheng. DALI: A workload-aware offloading framework for efficient MoE inference on local PCs. arXiv preprint arXiv:2602.03495, 2026. URL <https://arxiv.org/abs/2602.03495>.

A Experiment Plan and Status

Table 30 lists every experiment of the protocol (`experiments/experiments.yaml`), whether it ran, and where its data are. Hours, costs and commands of the full and minimal plans are in `experiments/PLAN.md`. The experiments that did not run keep their workloads, runners and data folders in the repository; running one makes its table appear in the paper on the next build.

Budget and the measurement campaign. The full plan (27.9 pod-hours, 196 USD with a 20% contingency) and a minimal subset (85 USD) were not approved. What ran instead is a reduced campaign, called the *lean plan* in the repository (`experiments/runs/lean.sh`). Its first session runs on one 2x RTX PRO 6000 pod of at most 120 minutes, paying boot, build and the weight download once: setup; E2 feasibility reduced to the two EXL3 boots without offload, capped at eight minutes; E8 with scenarios A and B at all seven concurrency levels, two repetitions and the probes; E10 on fixed-seed subsets of GSM8K, IFEval and HumanEval; and E3 reduced to the `gen30` plan against upstream UVA offload at the `gen30` host bytes, one repetition at $N \in \{1, 8, 32\}$; then one stretch step, a larger nested plan (the union of the `gen30` and `agent30` cold sets). The core steps were estimated at 98 pod-minutes (6.83 USD), 118 minutes (8.19 USD) with the contingency, and 108 minutes with the stretch step. The runner checks the budget before every step against a soft cap of 117 minutes (8.15 USD), and a watchdog deletes the pod at 120 minutes (8.36 USD).

Sessions. The first session ran on 2026-10-06 for 101 minutes (7.06 USD at the quoted price; `data/controlled/LEAN/session.json`): every core step and the stretch step, plus a repeat of E10 after the continuation fix of Section 11. Two bootstrap bugs (git refusing the copied sources, and a patch check that cannot tell whether overlapping patches were applied) forced two restarts before the first valid boot; both are fixed in `stack/build/bootstrap_pod.sh`. A second session on 2026-10-07 (43 minutes, 3.01 USD, `experiments/runs/long.sh`), on a different host, ran, after a first attempt aborted by a third bootstrap bug (a git configuration step that ran before git was installed; its boots were invalid and stayed on the pod), the long-context experiment E12, the E3 `gen30` point again as a same-pod baseline, and the UVA rung with the KV block size it needs. A third session (57 minutes, 3.98 USD, `experiments/runs/agent.sh`), on a third host (its record notes the first host’s public IP, but the CPU model and GPU bus ids differ; Table 7), ran the coding-agent experiment E13 after a first attempt that stopped before any measurement because the image lacked git.

Quality reference. The quality reference is the official DeepSeek API, whose `deepseek-flash` model is served by DeepSeek-V4.1-Flash according to its pricing page (fetched 2026-10-06, `data/static/pricing/`; 0.30 and 1.20 USD per million input and output tokens at peak). With estimated token counts the subsets were budgeted at about 0.23 USD; the run sent 464 requests, all answered, for 0.12 USD at peak prices from the real token counts logged by the proxy.

id	GPUs	question	status
E0	none	static byte accounting	done (Section 5)
E1	CPU	reproducible build of every variant, image digests and provenance	not run
E2	2x PRO 6000	feasibility without host offload; exact failure	ran (session 1; EXL3 runs only)
E3	2x PRO 6000	host share; upstream UVA offload at equal host bytes	ran (two shares, sessions 1 and 2; UVA cannot serve)
E4	2x PRO 6000	placement policy at 30%: recipe plans, random, our profile plan	not run
E5	2x PRO 6000	Engram int4 vs FP8 in host memory	not run
E6	2x PRO 6000	side-stream SM budget for host experts	not run
E7	2x PRO 6000	DSpark on vs off, code vs prose	not run
E8	2x PRO 6000	main benchmark: $N = 1$ to 64, probes, memory, PCIe	ran (session 1; closed loop; two repetitions in B, one in A beyond $N = 1$ and in the probes)
E9	2x B300	upstream all-in-VRAM reference, original checkpoint	not run
E10	2x PRO 6000	quality of the EXL3 stack	ran (session 1, fixed subsets)
E11	2x B300	quality of the reference	ran through DeepSeek’s API instead
E12	2x PRO 6000	long context at 128K and 256K tokens	ran (session 2)
E13	2x PRO 6000	coding-agent workload, streams at 40 tokens/s	ran (session 3)

Table 30: Controlled experiments and their status. Sessions: `data/controlled/LEAN/session.json`, `session2.json`, `session3.json`.