https://doi.org/10.32815/jitika.1288

Volume 20, Issue 2, 2026

Jurnal Ilmiah Teknologi Informasi Asia

Journal Homepage: https://jurnal.asia.ac.id/index.php/jitika

Article

Implementation and performance evaluation of a private AI cloud based on an OpenAI-compatible API in a higher education environment

Mabrur Roh Bintang Jaya *, Lia Farokhah

Applied Data Science, Politeknik Artificial Intelligence Budi Mulia Dua, Yogyakarta, 55584, Indonesia

Abstract—The use of Large Language Models (LLMs) through public cloud services still faces challenges such as token-based cost schemes, dependence on internet connectivity, data privacy risks, and limited control over infrastructure. These conditions drive the need for a private AI cloud implementation capable of providing inference services independently within an institutional environment. This study designs and implements a private AI cloud in a higher education environment using a client–server architecture based on two virtual machines (VMs). It uses a Tesla T4 GPU to run the quantized language model Qwen3.6-28B-REAP20-A3B (Q4_K_M) via llama.cpp. At the same time, it provides an OpenAI-Compatible API service, and the Service VM runs Open WebUI as the user interface. Evaluation was conducted using a synthetic burst workload to represent worst-case conditions. The evaluation follows a partial factorial design covering 13 of 30 possible combinations. A complete concurrency sweep (1, 3, 5, 10, and 20 users) was performed for the chat load class under both parallelism configurations (--parallel 1 and --parallel 4). In comparison, the long-prompt load classes were tested at a single concurrency point (5 users). However, validation using real agent applications is outside the scope of this study. Results show that at a single processing slot configuration (--parallel 1), system throughput reached a stable plateau of around 38.5 tokens/second even though GPU utilization was only around 60%, while latency increased linearly as the number of users grew. This finding indicates that the pattern is consistent with a bottleneck arising from queue serialization rather than from GPU computational capacity limits. Activating continuous batching with four processing slots (--parallel 4) increased throughput to around 65 tokens/second (~1.7×) and reduced latency under high load by around 35%, while GPU utilization remained below maximum capacity (~58%). In contrast, long-prompt loads increased GPU utilization to 97–100% peak (60–63% mean), indicating brief GPU saturation during prefill as the primary limiting factor. All test scenarios achieved a 100% protocol success rate (600 s timeout); practically viable capacity is up to 3 concurrent users when both SLO thresholds (TTFT ⩽ 2 s, RT ⩽ 30 s) are applied simultaneously, or up to 10 if only the response-time criterion is considered. This study contributes architectural documentation, a validated evaluation protocol, and service capacity characterization that educational institutions with limited computing resources can replicate.

Keywords—continuous batching; large language model deployment; openai-compatible api; performance evaluation; private ai cloud.

1. Introduction

The use of Large Language Models (LLMs) is increasingly widespread in supporting activities such as chatbots, programming assistants, knowledge management, and work automation. In higher education environments in particular, LLM-based chatbots have been widely adopted by students to support learning, ranging from reference searching to assignment preparation (Morell-Mengual et al., 2025; Yigci et

* Corresponding author.
E-mail Address: mabrur@plai.ac.id (M.R.B. Jaya)

Author E-mail(s): MRBJ (mabrur@plai.ac.id), LF (liafarokhah@plai.ac.id)
Digital Object Identifier 10.32815/jitika.1288
Manuscript submitted 19 July 2026; revised 10 August 2026; accepted 10 August 2026.
ISSN: 2580-8397(O), 0852-730X(P).

al., 2025). However, most implementations still rely on public cloud services, which entail token-based costs, dependence on internet connectivity, data privacy risks, and limited control over infrastructure (Khalil et al., 2025). This privacy risk is not an abstract concern: commercial LLMs are vulnerable to data leakage via mechanisms such as model inversion, training data extraction, and membership inference (K. Chen et al., 2025; Yan et al., 2025). For institutions that process sensitive data, including academic and research data, these constraints drive the need for alternative, self-managed AI service provisioning. This shift toward on-premises provisioning is supported by strategic analyses showing that offering a local inference option alongside cloud services can improve user surplus and address data sovereignty and latency concerns, even though it may reduce provider profits through market cannibalization (Zhang et al., 2025). This study treats data privacy and infrastructure control as its primary motivation; cost is noted as context and is not evaluated. Total cost of ownership analysis is outside the scope of this study. This trend coincides with growing institutional emphasis on GenAI governance frameworks that prioritize data privacy, equitable access, and infrastructure investment in higher education settings (Crompton et al., 2026).

The development of quantizable open-weight language models enables organizations to provide LLM services on-premises with far lower hardware requirements than data-center-class or large-scale GPU clusters (Malakhov, 2025). A commonly used approach is to provide an OpenAI-Compatible API on top of an inference runtime such as llama.cpp, so that client applications that already support the OpenAI API scheme — ranging from chat interfaces (Open WebUI) and programming assistance tools (OpenCode) to internal agents — are compatible with local models without significant changes on the application side (Stuhlmann et al., 2025). In this study, these applications are positioned as API-compatibility targets, and performance evaluation is conducted using synthetic workloads that reflect their access profiles.

Most available LLM inference benchmarking studies focus on data-center-class GPUs (H100, A100, A30) or high-end consumer GPUs (RTX 5090), which are not necessarily available to budget-constrained educational institutions (Khalil et al., 2025; Stuhlmann et al., 2025). In addition, concurrent evaluation (requests processed simultaneously) in these studies is generally conducted under generic chat loads rather than agentic-type load patterns (programming assistants, MCP-based agents), which are characterized by long prompts and repeated calls. This gap underlies the present study: implementing and evaluating a private AI cloud based on an OpenAI-Compatible API in a higher education environment using mid-range GPU hardware (Tesla T4), with particular attention to the differences between chat and agentic-type load profiles.

Based on the above description, this study is designed to answer three questions: how to implement a private LLM-based AI cloud using an OpenAI-Compatible API on mid-range GPU hardware; how the resulting service performs in terms of latency, throughput, resource usage, and service-stability parameters across various concurrency levels and parallel-slot configurations; and how bottleneck characteristics differ between chat and agentic-type loads, along with their implications for service capacity planning. In line with these questions, this study aims to (1) implement a private AI cloud as an internal AI service based on an OpenAI-Compatible API; (2) evaluate service performance under various load scenarios and parallel-slot configurations; (3) analyze computational resource utilization during inference for chat and agentic-type loads; and (4) identify service capacity characteristics and limits as a basis for implementation recommendations.

This study contributes: (a) implementation of a private AI cloud using local mid-range GPU infrastructure; (b) provision of an OpenAI-Compatible API service for various AI application profiles (chat and agentic-type, as represented by synthetic long-prompt workloads); (c) empirical characterization of service capacity and bottlenecks, including a contrast between parallel-slot configurations and differences between chat and agentic-type load profiles; and (d) documentation of a verified implementation and testing protocol that other institutions with limited resources can replicate.

This study focuses on system implementation and on the evaluation of infrastructure performance. It therefore does not address LLM model development, fine-tuning processes, inference algorithm optimization, or evaluation of model output quality (accuracy, relevance, and output alignment). Accordingly, the findings of this study are limited to infrastructure performance aspects. They cannot be used to conclude model output quality, particularly given that quantized Mixture-of-Experts (MoE) models have been reported to be sensitive to precision reduction.

All test scenarios use uniform synthetic prompts within each load class to preserve comparability of results. This approach simplifies the analysis but limits generalization to load variations in real-world environments. Agentic load is represented through synthetic long prompts that emphasize the prefill phase, rather than through real agent applications running concurrently. Accordingly, this study represents the computational characteristics of agentic-type load. However, it does not yet cover request arrival patterns, session management, or tool-calling as commonly found in applications based on the Model Context Protocol (MCP). Hermes and OpenCode are positioned as API-compatibility targets, not as testing subjects. Therefore, validation using real-agent applications is recommended for future research.

2. Methods

2.1. Literature review

2.1.1. Local LLM serving frameworks

Several local inference runtimes are available for self-hosting LLM services, including vLLM, HuggingFace TGI, SGLang, LMDeploy, and llama.cpp. Stuhlmann et al. (2025) show that each runtime has advantages in different usage scenarios. LMDeploy excels in single-stream responsiveness, SGLang in batch throughput on mid-range GPUs, while vLLM is more optimal for high concurrency through efficient scheduling and batching mechanisms. Meanwhile, llama.cpp is designed to run on both CPU and GPU with lightweight dependencies and support for the GGUF format (Malakhov, 2025). These characteristics make a llama.cpp suitable for implementation on a single mid-range GPU, such as the NVIDIA Tesla T4 used in this study. In addition, llama.cpp already provides a built-in continuous batching mechanism, so no additional components are required to handle parallel requests. Continuous GPU utilization measurement is also important for distinguishing

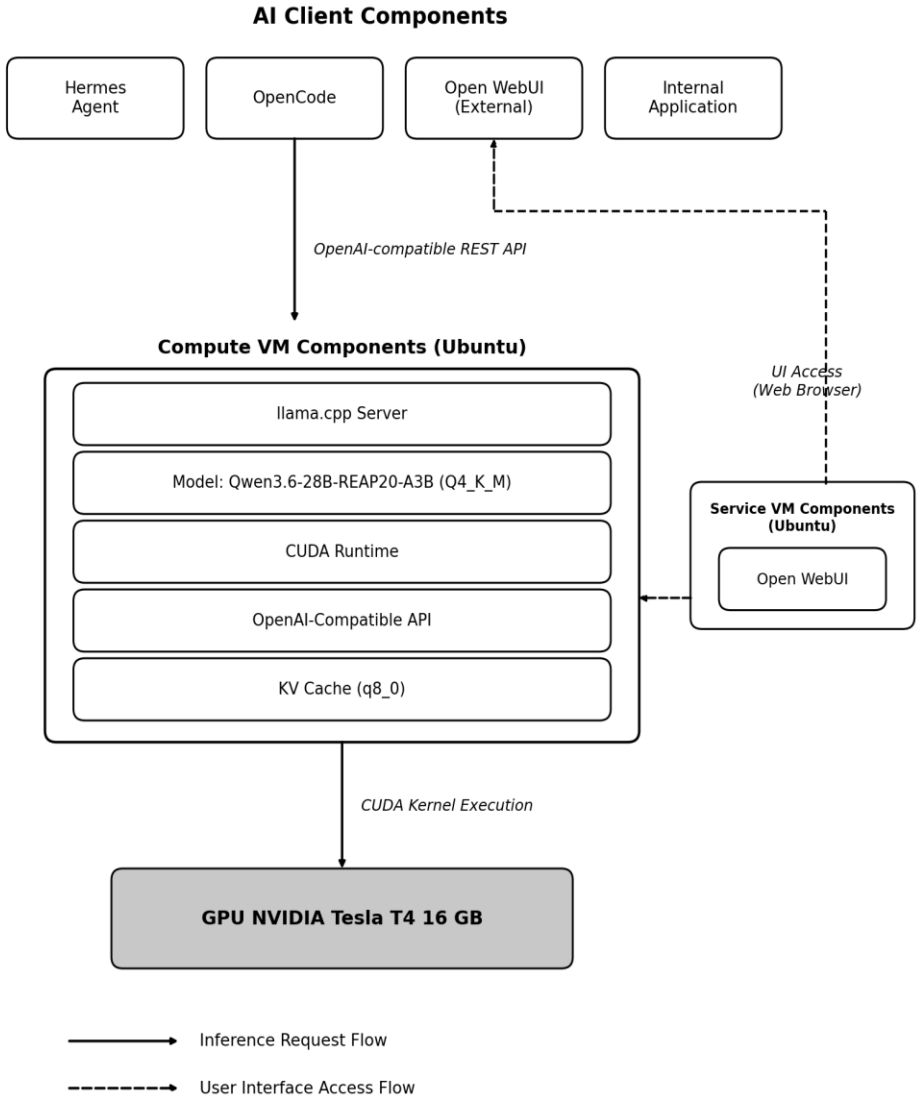


Fig. 1. Private AI cloud architecture.

computational bottlenecks from non-computational bottlenecks, such as queue serialization or scheduling overhead (Kakolyris et al., 2024). The continuous-batching mechanisms underlying these runtimes trace back to the iteration-level scheduling first proposed by Orca (Yu et al., 2022) and to the PagedAttention memory-management scheme introduced by vLLM (Kwon et al., 2023), both of which established the throughput gains that motivate this study's baseline expectations.

2.1.2. Quantization and compression of MoE models

Model-weight quantization is a key determinant of the feasibility of running LLMs on limited hardware. Khalil et al. (2025) show that a 30B-parameter model with an aggressive quantization scheme can be run on a single consumer GPU and achieve performance competitive with commercial cloud services for single-user loads but report a decline in per-request efficiency as the number of concurrent users increases, due to prefill costs and request serialization at the serving stage. Malakhov (2025) reports llama.cpp throughput based on 4-bit quantization (Q4_K), ranging from 2–120 tokens/second, depending on model size and hardware class. In addition to quantization, MoE architecture compression through expert pruning is a relevant technique for reducing memory footprint: Lasby et al. (2025) introduce the REAP (Router-weighted Expert Activation Pruning) method, which prunes experts with minimal contribution while preserving routing behavior, and report near-lossless compression on code generation tasks. These findings confirm that the quantization level, its accompanying scheme, and the compression method must be reported explicitly, as they directly affect footprint, throughput, and latency. More broadly, a comprehensive review of LLM compression techniques shows that pruning, quantization, and knowledge distillation are three complementary approaches, each with different trade-offs between compression ratio and performance degradation (Zhu et al., 2024) while a review of mixed-precision frameworks emphasizes that mixed-precision strategies need to be tailored to the characteristics of the target hardware so that memory efficiency does not excessively compromise accuracy (Rakka et al., 2024). In the specific context of code generation, extreme quantization has also been reported to degrade output quality on resource-constrained programming-language benchmarks, underscoring the need for

Table 1. Test scenarios

Block	Load Class (Concurrencies)	--parallel / Cells
B1	A: short/short (204 in, 256 out) × {1,3,5,10,20}	p=1 / 5 cells
B2	A: short/short (204 in, 256 out) × {1,3,5,10,20}	p=4 / 5 cells
B3	B: long-in/short-out (2,990 in, 256 out) × 5	p=1 / 1 cell
B4	B: long-in/short-out (2,990 in, 256 out) × 5	p=4 / 1 cell
B5	C: long-in/long-out (2,576 in, 1,500 out) × 5	p=1 / 1 cell

Table 2. Test categories and parameters

Category	Parameter
API Performance	Response time; time to first token (TTFT); tokens per second (TPS); request success rate
Resource utilization	CPU utilization; RAM utilization; GPU utilization (time-series); GPU memory (VRAM)
Stability	Concurrent requests handled; API availability; error rate

Table 3. Hardware and software configuration

Component	Specification
GPU	NVIDIA Tesla T4, 16 GB VRAM
Host CPU (Compute VM)	8 vCPU
Host RAM (Compute VM)	16 GB
Host CPU/RAM (Service VM)	8 vCPU / 16 GB RAM
Operating system	Ubuntu 24.04.3 LTS (Noble Numbat)
Inference runtime	llama.cpp, commit ad39cca
Language model	Qwen3.6-28B-REAP20-A3B (MoE, ~3B active per token)
Quantization scheme	Q4_K_M (GGUF)
Server flags	--ctx-size 16384 --batch-size 2048 --ubatch-size 512 --flash-attn on --fit on --cache-type-k q8_0 --cache-type-v q8_0
Parallel slots tested	--parallel 1 and --parallel 4 (continuous batching default: enabled)
CUDA/driver version	12.9
CPU Model	Intel Xeon Silver 4210R @ 2.40 GHz
GPU Access Method	PCIe Pass-Through (VMware ESXi)
NVIDIA Driver	575.57.08
GGUF File Size	17 GB (SHA256: dcd137ca7984ebdaa041ef80281322773bc1a9911ef639b7e698a20979953a00)
GPU Layers	All 40 layers on GPU (--fit on)
VRAM Breakdown	~12.6 GB weights + ~0.7 GB KV + ~0.3 GB buffers

output-quality evaluation that accounts for the interaction between the compression method and the task domain (Nyamsuren, 2025). The relevance of MoE architecture for deployment on edge devices or single GPUs is also supported by broader literature. MoE enables increased model capacity without a proportional increase in computational cost, making it suitable for memory- and compute-constrained environments, such as edge devices (Wang et al., 2025), and for distributed inference scenarios with limited storage capacity for expert parameters (Q. Chen et al., 2025). These weight-only quantization approaches build on foundational post-training quantization methods such as GPTQ (Frantar et al., 2022), while the underlying Mixture-of-Experts architecture that REAP prunes originates from the Switch Transformer framework (Fedus et al., 2022).

2.1.3. Performance evaluation metrics and methodology

Local inference benchmarking frameworks such as Bench360 use time-to-first-token (TTFT), time-per-output-token (TPOT), generation latency, and throughput (tokens per second, TPS) as standard performance indicators, alongside resource and serving metrics (Stuhlmann et al., 2025). The selection of test parameters in this study is aligned with this metric taxonomy. Unlike prior studies that rely on utilization snapshots, this study measures GPU utilization as a time series during the inference phase using DCGM/Prometheus, enabling claims about GPU saturation or available capacity to be based on continuous measurement. Complementing throughput- and latency-based indicators, recent work has also questioned whether conventional SLO and goodput definitions adequately capture user experience in LLM serving, proposing smoothed alternatives that penalize artificial latency-shifting (Wang et al., 2024).

2.1.4. Research position

The reviewed literature reveals gaps on two fronts. First, most local inference benchmarking studies focus on data-center-class GPUs or high-end consumer GPUs with large memory capacity (24–32 GB). In contrast, budget-constrained educational institutions are more likely to have mid-range GPUs such as the Tesla T4 (16 GB). Second, most evaluations are conducted under generic chat loads rather than agentic-type load patterns with long prompts that are now common in programming assistants and protocol-based agents such as MCP. This study addresses this gap by documenting the architecture, a verified evaluation protocol, and the capacity characterization of a private AI cloud running on a Tesla T4, which explicitly distinguishes between chat and agentic-type load profiles. This study was conducted at Politeknik Artificial Intelligence Budi Mulia Dua, a higher education institution.

2.2. Research method

2.2.1. Research stages

This study consists of eight stages. First, analyzing internal AI service requirements. Second, designing private AI cloud architecture. Third, implementing a virtual environment consisting of a virtual machine (VM) Service and a Compute VM. Fourth, implementing an inference service on the Compute VM

Table 4. API performance results at --parallel 1 (chat class, 15 iterations per scenario (12 analyzed))

Load	Resp. Time (ms)	TTFT (ms)	TPS	Sys. Throughput
1	6,736 ± 279	271 ± 20	39.7 ± 1.6	29.8
3	13,134 ± 5,397	6,798 ± 5,408	40.4 ± 0.8	35.6
5	20,331 ± 9,643	13,794 ± 9,632	39.2 ± 0.8	35.9
10	37,219 ± 19,443	30,688 ± 19,448	39.2 ± 0.8	36.9
20	68,899 ± 37,948	62,520 ± 37,939	40.2 ± 1.5	38.5

Note: TPS = per-request (decode); Sys = wall-clock

Table 5. API performance results at --parallel 4 (chat class, 15 iterations per scenario (12 analyzed))

Load	Resp. Time (ms)	TTFT (ms)	TPS	Sys. Throughput
1	6,718 ± 122	263 ± 13	39.7 ± 0.7	29.8
3	12,623 ± 177	555 ± 73	21.2 ± 0.3	52.9
5	16,536 ± 2,928	3,422 ± 6,138	21.5 ± 8.3	52.9
10	26,262 ± 9,837	12,664 ± 11,446	19.4 ± 4.0	60.7
20	46,121 ± 21,722	31,396 ± 21,727	17.4 ± 0.2	65.1

Note: TPS = per-request (decode); Sys = wall-clock; At a load of 5 users, the TTFT distribution is bimodal (see Fig. 3); the mean ± SD value is less representative and is discussed via median and two-cluster structure instead.

Table 6. Difference between chat and agentic-type load classes (5 users, 15 iterations per scenario (12 analyzed))

Class	Prompt	Resp. T (ms)	TTFT (ms)	GPU% (peak/mean)	Throughput
A: short/short (--p1)	204	20,331	13,794	68% / 61.0%	36.0
B: long/short (--p1)	2,990	20,387	13,691	97% / 62.6%	35.3
C: long/long (--p1)	2,576	118,280	78,938	100% / 62.8%	37.6
B: long/short (--p4)	2,990	17,618	3,774	100% / 60.4%	46.4

using llama.cpp and the quantized LLM Qwen3.6-28B-REAP20-A3B (Q4_K_M). Fifth, integrating Open WebUI via an OpenAI-Compatible API. Sixth, testing at five user concurrency levels with two inference parallelism configurations (i.e., --parallel 1 and --parallel 4). Seventh, evaluating performance based on predetermined parameters. Finally, drawing conclusions and formulating the implementation recommendations.

2.2.2. System architecture

The system architecture consists of four components: the AI Client, the Service VM, the Compute VM, and an NVIDIA Tesla T4 GPU (Fig. 1). The AI Client encompasses applications that support the OpenAI-Compatible API, such as Hermes, OpenCode, Open WebUI, and internal institutional applications. In this study, these applications were not run directly; instead, they were represented through synthetic test scenarios. The Service VM runs Open WebUI as the user interface, while the Compute VM runs llama.cpp with the Qwen3.6-28B-REAP20-A3B (Q4_K_M) language model and simultaneously provides the OpenAI-Compatible API service. The entire inference process is accelerated using an NVIDIA Tesla T4 GPU via the CUDA runtime.

2.2.3. Model selection

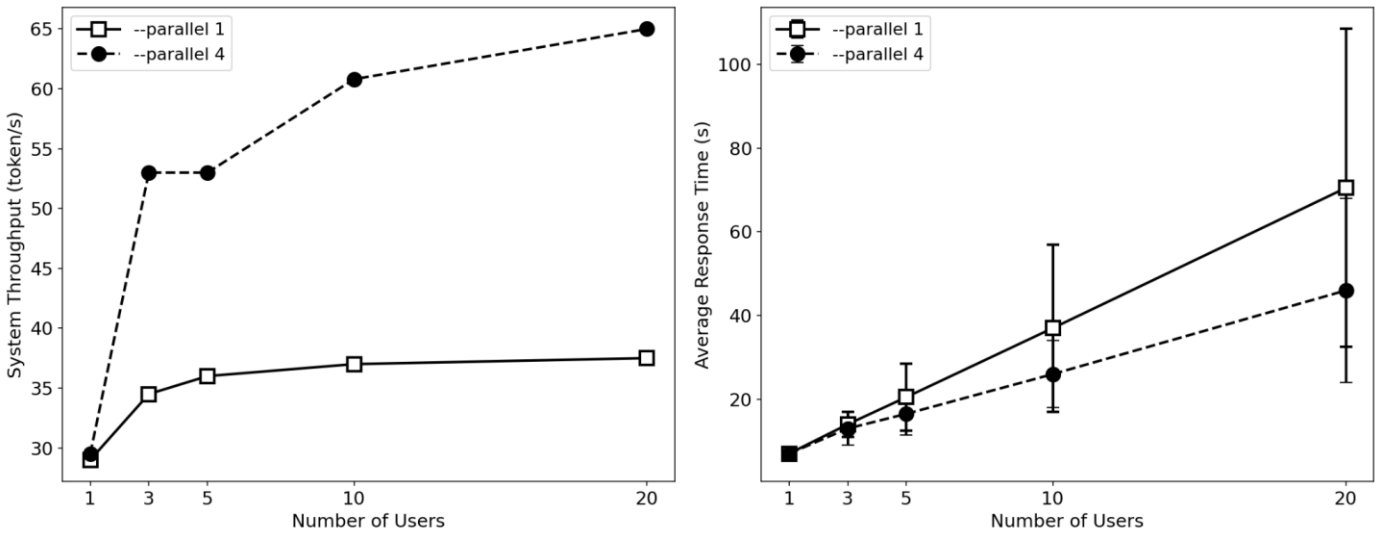
The language model used is Qwen3.6-28B-REAP20-A3B with GGUF Q4_K_M quantization. This model is a compressed variant of Qwen3.6-35B-A3B, a sparse MoE model with 35 billion total parameters but only about 3 billion active parameters per token (the "A3B" notation). This MoE characteristic is the basis for the selection: the model provides representational capacity equivalent to a large model, but with inference computational cost close to that of a dense 3B model, making it feasible to run on a mid-range GPU with limited VRAM such as the Tesla T4 (16 GB).

The variant used is the result of compression using the REAP (Router-weighted Expert Activation Pruning) method, which prunes about 20% of experts in the MoE architecture, reducing model size by about 25% compared to the base model, without changing the number of active parameters per token (~3 billion). The selection of this pruning-based compression method is supported by Lasby et al.'s (2025) finding that REAP achieves near-lossless compression on code generation tasks, which is relevant to the service's orientation toward agentic-type loads and coding assistants. Q4_K_M quantization was chosen as a common balance point between memory footprint and output quality for running LLMs on limited hardware (Khalil et al., 2025; Malakhov, 2025).

It should be noted that the variant used is a community derivative: expert-pruned with REAP from Qwen3.6-35B-A3B by 0xSero, then GGUF-quantized by barozp (Hugging Face: barozp/Qwen3.6-28B-REAP20-A3B-GGUF, arXiv: 2510.13999). It is therefore not identical to any official Qwen release. In accordance with the scope of this study, model output quality was not evaluated; model selection was based on the suitability of memory footprint and deployment characteristics.

2.2.4. Testing parameters and configuration

Throughout this paper, a "scenario" (or "cell") refers to one of the 13 tested combinations from the full factorial of 30 (Table 1). Each cell was run for 15 iterations, with the first 3 iterations used as a warm-up and excluded from per-request analysis (n = 12 effective). Test parameters are grouped into three categories as presented in Table 2. GPU utilization is measured as a time series during the inference phase via DCGM/Prometheus at a 1-second resolution. This synthetic burst-load design is consistent with recent large-scale characterizations of production LLM serving workloads, which likewise emphasize



(a) (b)

Fig. 2. Comparison of --parallel 1 vs 4: (a) system throughput rises from a plateau of ~38.5 to ~65 tokens/second; (b) response time under high load drops by ~35%.

controlled, reproducible load generation to complement field traces (Xiang et al., 2026). Meanwhile, the test-environment configuration is summarized in Table 3.

Two throughput metrics are reported separately and are not interchangeable. Per-request TPS is computed as the number of output tokens per request divided by that request's decode duration, as reported by the llama.cpp server, as follows:

$$\mathrm{TPS}_{\mathrm{req}} = \frac{N_{\mathrm{output}}}{t_{\mathrm{decode}}}. \quad (1)$$

It excludes prefill time and queue waiting time. System throughput is computed as the total output tokens across all requests in a scenario divided by the wall-clock interval from dispatch of the first request to completion of the last, as follows:

$$\mathrm{TPS}_{\mathrm{sys}} = \sum \frac{N_{\mathrm{output}}}{\left(t_{\mathrm{last\_complete}} - t_{\mathrm{first\_dispatch}}\right)}. \quad (2)$$

The gap between the two at single-user load (39.7 vs 29.8 tokens/second) arises from the inclusion of the prefill phase in the denominator of the second metric. For illustration, using the first measured iteration at one concurrent user (raw data: block_P1_u1, iteration 3):

- N_{output} = 256 tokens,
- t_{decode} (streaming duration) = 6,322 ms ≈ 6.3 s,
- yielding $\mathrm{TPS}_{\mathrm{req}} = 256/6.3 \approx 40.5$ tokens/second.

The per-iteration wall-clock (response time) was 6,577 ms, or around 6.6 s, yielding a per-iteration system throughput of 256 / 6.6 ≈ 38.9 tokens/second. The benchmark script computes the summary-level system throughput of 29.8 tokens/second as the total output tokens across all 15 iterations, including warm-up (3,840 = 15 x 256), divided by the scenario's wall-clock time (128.9 s from first request dispatch to last completion). Warm-up iterations are included in this system-level metric because they are part of the measured campaign wall-clock; excluding them yields a nearly identical value (29.9 tokens/second) because the warm-up overhead and tokens are proportionally small. Per-request metrics ($\mathrm{TPS}_{\mathrm{req}}$, TTFT, response time) reported in Table 4-Table 6 use only the 12 non-warm-up iterations. The gap between $\mathrm{TPS}_{\mathrm{req}}$ (40.5) and summary system throughput (29.8) arise from the inclusion of the prefill phase and inter-iteration scheduling in the denominator. All cross-configuration comparisons use system throughput.

Testing was conducted at five concurrent-user load levels (1, 3, 5, 10, and 20) and two llama.cpp parallel-processing-slot configurations (--parallel 1 and --parallel 4), as summarized in Table 1. In addition, three prompt-length-based load classes were tested to represent chat and agentic-type load patterns.

Each test was run for 15 iterations, with the first three iterations used as a warm-up and excluded from analysis (n = 12 effective iterations, 12–240 requests per cell, depending on concurrency). This sample size is adequate for the claims advanced, for two reasons. First, for the --parallel 1 configuration, per-request TPS variability is low (SD 0.8–1.6 tokens/second across all concurrency levels), indicating stable throughput that 12 iterations are sufficient to characterize. Second, for the --parallel 4 configuration, TPS standard deviation is higher (up to 8.3 tokens/second at 5 users), but this is not sampling noise. It is the quantitative signature of the deterministic bimodal scheduling pattern described in Section 4.3, where exactly 4 of 5 requests per iteration receive fast service, and 1 receives delayed service. This pattern recurs identically in all 15 iterations; increasing the sample size would confirm the pattern, not change it. Tail percentile estimates (P95/P99) are therefore not reported and are not used as the basis for any conclusion; stable tail-latency characterization requires at least 100 iterations per cell and is identified as future work in Section 5. Median, range, and standard deviation (SD) are reported in the tables; 95% confidence intervals (CIs) are reported in the text where inferential comparison is made. SD and CI are not interchangeable: SD describes variability across individual measurements, while CI quantifies uncertainty around the mean estimate. At n = 12, both are meaningfully estimable under the assumption of approximate normality for

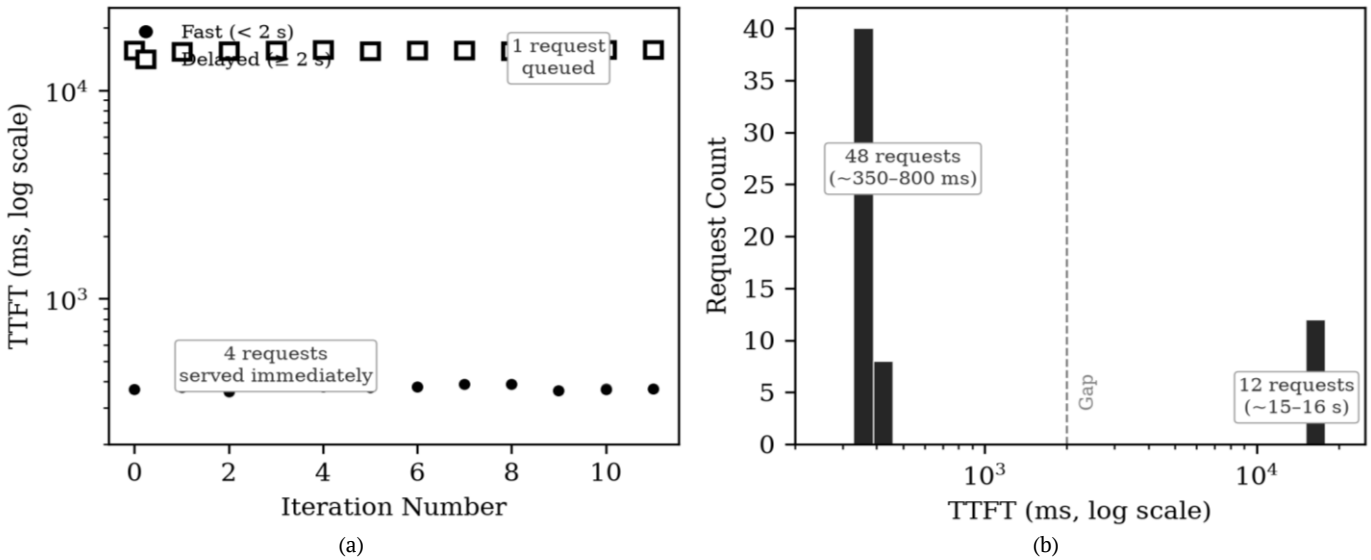


Fig. 3. Bimodal TTFT distribution at 5 users, --parallel 4 (60 measured requests): (a) recurring pattern of 4 fast + 1 slow per iteration; (b) two separate clusters (48 fast vs 12 delayed) with an empty gap. Log-scale x-axis; a dashed gap marker separates the two clusters.

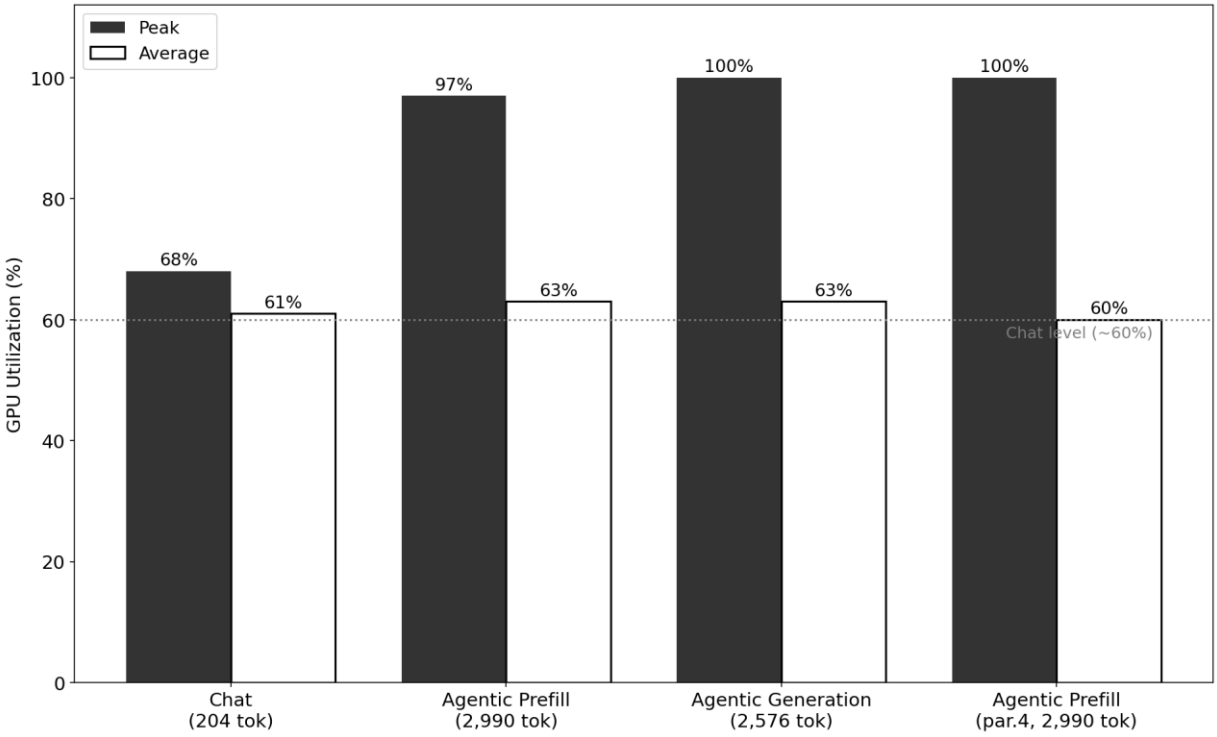


Fig. 4. Peak and average GPU utilization by load class (5 users). Chat load does not saturate the GPU (~68% peak), while agentic-type load with long prompts reaches 97–100%.

the chat-load cells.

All performance measurements were taken at the OpenAI-Compatible API boundary on the Compute VM. The Open WebUI component, the Service VM, user authentication, and the client–server network paths are shown in Fig. 1 were implemented and operational, but lie outside the measurement boundary. Reported latency figures therefore characterize the inference layer; end-to-end latency experienced by end users will be higher due to the interface-layer and network overhead not measured in this study.

In each scenario, all requests were sent in a burst (nearly simultaneously) to represent worst-case conditions. This approach was chosen to test the system at maximum concurrency, allowing capacity limits and queueing characteristics to be observed clearly. The test results, therefore, represent the upper bound of system performance,

not the request arrival pattern of everyday use. In practice, agent applications generally generate requests that are bursty but not fully simultaneous, so the results of this study should be understood as an evaluation under the most challenging conditions for the system.

3. Results

This section reports the empirical measurements obtained from the test campaign, organized by configuration and load class. All 13 test scenarios achieved a 100% success rate (0% error rate) with no failed requests. Interpretation of these results, including comparison with prior work and their practical implications, is presented in Section 5.

3.1. Performance at the single-slot configuration (--parallel 1)

The --parallel 1 configuration is the initial deployment configuration (the default value used in the production service prior to this study). Table 4 presents the API performance test results for the chat load class at the --parallel 1 configuration. Response time increased almost linearly with the number of users (from 6,736 ms at 1 user to 68,899 ms at 20 users), while per-request TPS remained stable at ~39.7 tokens/second across all load levels. Notably, aggregate system throughput plateaued at ~38.5 tokens/second even as the number of users increased twentyfold, and GPU utilization remained at ~60% without saturating.

3.2. Effect of multi-slot continuous batching (--parallel 4)

Activating four slots (multi-slot continuous batching) increased aggregate system throughput from ~38.5 to ~65.1 tokens/second under high load (~1.7×) and reduced response time at 20 users from 68,899 ± 5,391 ms (95% CI) to 46,121 ± 3,086 ms (95% CI) (~35% faster), as contrasted in Fig. 2. However, GPU utilization remained unsaturated (~58%), and per-request TPS decreased (from ~39 to ~17–21 tokens/second) because decode capacity is shared among concurrently processed requests.

The two clusters are reported separately: fast cluster (n=48 requests) median 376 ms, range 357–433 ms; delayed cluster (n=12 requests) median 15615 ms, range 15366–15701 ms. Aggregate summary statistics for this cell should not be used for capacity planning.

Under the --parallel 4 configuration, aggregate throughput plateaued at approximately 65 tokens/second, with GPU utilization at approximately 58%. VRAM consumption remained at 13,591 MiB with no growth when slots were increased from 1 to 4, and no cache eviction events appeared in the server logs. Host-side utilization also remained below saturation (CPU <65%, RAM <63%).

3.3. Deterministic queueing effect

At a load of 5 users with 4 processing slots, TTFT values show high variation (SD of 6,138 ms, exceeding the mean of 3,422 ms). Analysis of the per-request distribution (Fig. 3) shows a consistent bimodal pattern. In each iteration, four requests are processed first, with a TTFT of around 335–446 ms, while the remaining request is processed only after one generation completes, resulting in a TTFT of around 15,500 ms. This "four fast, one slow" pattern appears consistently across all 12 analyzed iterations (the same pattern was confirmed in the 3 warm-up iterations, but those data points are excluded from quantitative analysis), with no TTFT values between approximately 446 ms and 15,150 ms.

3.4. Differences between chat and agentic-type load profiles

Fig. 4 and Table 6 show the difference in characteristics between chat and agentic-type loads. Chat load reaches peak GPU utilization of about 66–67%, with a mean of 61% across the full 1-second DCGM time series, while agentic-type load with long-prompt load classes (B prefill-heavy, C decode-heavy) increases peak GPU utilization to 97–100%. The sustained fraction — defined here as the percentage of 1-second GPU samples at or above 90% utilization — is 0.1–2.5% for agentic-type load, compared to 0% for chat load. Class C produces the highest response time, around 118,280 ms at the --parallel 1 configuration.

3.5. Resource utilization and stability

VRAM consumption remained within 13.6 GB of a total of 16 GB across all test scenarios, both at the single- and four-processing-slot configurations (--parallel 1 and --parallel 4). GPU utilization was approximately 61% for chat load and increased to 97–100% for agentic-type load.

Layer allocation was verified from the llama.cpp server initialization log. With --fit on, all 40 model layers were placed on the GPU. The 13.6 GB of VRAM consumption comprises approximately 12.6 GB of model weights (Q4_K_M quantization), 0.7 GB of quantized KV cache (q8_0 at --ctx-size 16384, 4 slots), and 0.3 GB of compute buffers. Partial off-GPU placement would alter the interpretation of the GPU utilization metrics; none was observed.

Companion measurements from the original test campaign show host CPU utilization of 0.5–57.3% (increasing with concurrency but never exceeding 65%) and RAM consumption of 57–62% (9.1–9.9 GB of 15 GiB) across all test cells, indicating neither host CPU nor host memory saturation.

All 13 test scenarios achieved a 100% success rate with no errors (0% error rate), including at a load of 20 concurrent users, indicating service stability across the evaluated load range.

This study did not define a service-level objective (SLO) prior to testing; results are instead evaluated against two reference thresholds common to interactive services: TTFT ⩽ 2 seconds and total response time ⩽ 30 seconds. Against these thresholds, the --parallel 4 configuration satisfies the TTFT criterion up to 3 concurrent users (median TTFT 533 ms). In contrast, at 5 users, a subset of requests incurs a TTFT of ~15.6 s due to non-preemptive scheduling. The total-response-time criterion is met up to 10 users (26,262 ms) but not at 20 users (46,121 ms).

4. Discussion

This section interprets the results presented in Section 4, situates the findings within the broader literature on local LLM serving, and discusses their practical implications and limitations.

4.1. Single-slot serialization bottleneck

One of the study's practical contributions is precisely to highlight the consequences of this default value on aggregate throughput, which is relevant for administrators using similar configurations. The invariance of per-request TPS (39.7 ± 1.0 tokens/second (95% CI: 38.7–40.7; n=12 measured iterations)) across a twentyfold increase in concurrency is the primary evidence of serialization: had GPU computational capacity been the binding constraint, per-stream decode speed would have degraded. This pattern indicates that at a single slot, requests are processed serially: adding users does not reduce per-stream processing speed (TPS remains ~39.7) but instead lengthens the queue, causing latency to grow linearly. Because the GPU never reaches saturation (~60%), this throughput plateau is consistent with a bottleneck arising from single-slot queue serialization rather than from GPU computational limitation. These findings, from a different mechanistic angle, complement the report by Khalil et al. (2025) on efficiency degradation at low concurrency, where the degradation observed here is queueing-bound rather than purely compute-bound. Note that Khalil et al. evaluated a different model (Qwen3-30B) on consumer-grade hardware; the comparison is limited to the general serving-behavior pattern (serialization at low concurrency), not to model-level performance.

4.2. Multi-slot batching: a persistent, unidentified bottleneck

These results directly answer the capacity question: multi-slot continuous batching increases aggregate throughput and reduces latency under high load, but the system still does not push performance beyond the GPU's limit under chat load. This plateau indicates that throughput is constrained by factors other than GPU computational utilization. Identifying the specific mechanism (related to memory bandwidth, slot-scheduling overhead, or other limits) requires kernel-level profiling, which is narrowed using data already collected. KV cache capacity exhaustion can be excluded based on the stable VRAM footprint reported above, and host-side saturation was not observed either. Because host CPU and RAM profiling was not performed under the --parallel 4 configuration, the study cannot definitively identify whether the throughput plateau is caused by slot-scheduling overhead, GPU memory bandwidth, or host-side factors. Separating these candidates requires dedicated profiling under --parallel 4 using kernel-level tools (e.g., Nsight Systems), which is identified as the priority direction for future work. In other words, under chat load, the observed results are consistent with serving-side limitations rather than GPU computational speed as the primary constraint, suggesting that unused computational capacity may remain.

4.3. Non-preemptive scheduling and its implications

This pattern indicates that the slot-scheduling mechanism is non-preemptive, meaning a request being processed cannot be interrupted. As a result, when the number of requests exceeds the number of processing slots, the next request must wait until one of the slots finishes processing the previous request. This condition causes latency spikes under loads with long-generation processes, such as programming assistants or agentic-type workloads (long-context, prefill-heavy), rather than gradually increasing latency.

It should be emphasized that the observed latency cliff is a direct consequence of the simultaneous-burst-load model. In real deployments, request arrivals may follow a Poisson-like distribution with natural staggering induced by reading time, typing time, and conversational turn-taking. However, this characterization is hypothesized rather than measured in this study; real institutional traffic may also be clustered, periodic, or synchronized around class activities and deadlines. This staggering allows slot reuse before the next request arrives, so the queueing penalty under real load will be lower than reported. Results should therefore be read as an upper bound on queueing severity, and capacity figures derived from them are conservative. Characterizing system behavior under Poisson arrivals is identified as future work. This non-preemptive behavior mirrors the request-level scheduling limitation originally identified in Orca (Yu et al., 2022); subsequent systems, such as FastServe (Wu et al., 2026), address this head-of-line blocking pattern through token-level preemptive scheduling, suggesting a concrete architectural path to mitigate the queueing cliff observed here.

4.4. Queueing-bound vs. compute-bound behavior

The peak-to-mean ratio (97–100% peak vs. 60–63% mean) indicates that GPU saturation occurs during brief prefill bursts rather than as a sustained condition, reflecting the high computational load imposed during the prefill and decode processes for Class C. Long-prompt workloads therefore produce brief GPU saturation peaks, not sustained high utilization.

For load Class C, a mean response time of 118,280 ms falls far outside any interactive threshold; this load class is therefore not viable for synchronous interactive use on the tested hardware and is suitable only for asynchronous or batch usage patterns.

The term "agentic load" in this study refers specifically to a long-prompt computational profile, not to complete agent applications. The tests do not include repeated model calls, tool-calling, session-history management, or external API calls that characterize real agent applications. Results generalize to the computational characteristics of the prefill phase, not to agent applications as a whole.

The difference in characteristics between the two load classes is the study’s main finding. Chat load shows queueing-bound characteristics, meaning throughput is more influenced by queue serialization and scheduling mechanisms, even though GPU utilization remains below maximum capacity. In contrast, agentic-type load shows compute-bound characteristics, with GPU utilization rising to near-maximum capacity due to the computational demand of long-prompt loads (prefill-heavy for Class B, decode-heavy for Class C). This finding indicates that service capacity for agentic-type load cannot be directly estimated from chat-load test results. Therefore, private AI cloud capacity planning intended for applications such as programming assistants or Model Context Protocol (MCP)-based agents should use test scenarios that represent prefill-heavy load characteristics.

These findings apply to the configuration used in this study, namely the Qwen3.6-28B-REAP20-A3B model run using llama.cpp on an NVIDIA Tesla T4 GPU. Queueing-bound and compute-bound characteristics may differ across other runtimes with different batching strategies (e.g., vLLM with Paged Attention), GPU devices with different memory characteristics, or models with different prefill-to-decode

ratios. Generalizing the results to other configurations, therefore, requires separate testing. This compute-bound behavior under long-prompt load is consistent with the rationale behind prefill/decode disaggregation architectures such as DistServe (Zhong et al., 2024), which physically separate the two phases onto different hardware precisely because they exhibit such different bottleneck profiles.

4.5. Implications for capacity planning and stability

The stable VRAM footprint across configurations indicates that memory requirements are relatively stable even as parallelism levels and load characteristics change. However, the remaining available GPU memory capacity is limited for supporting additional processing slots or increased context length. The contrast between chat-load and agentic-load GPU utilization reinforces the finding that the system's limiting factor shifts from queue serialization (queue-bound) to GPU computational capacity (compute-bound) when the prefill process dominates the load.

CPU and RAM were measured during the original --parallel 1 campaign; host memory consumption is independent of inference output length and is reported as a proxy for all configurations. Direct host profiling with --parallel 4 was not performed in this study; the proxy assumption is reasonable for RAM (which is dominated by model loading rather than slot count), but may understate the CPU overhead introduced by the slot-scheduling layer. Dedicated host profiling under --parallel 4 is recommended as future work.

Client timeouts generally cause failures, as widely reported in other studies under extreme load; such conditions were not observed in this study, given a timeout of 600 seconds.

The recommended service capacity for this hardware is up to 3 concurrent users for interactive chat load under --parallel 4, provided both thresholds (TTFT ⩽ 2 s and RT ⩽ 30 s) are satisfied simultaneously. If only the total response time criterion is applied, up to 10 concurrent users remain viable. More broadly, this pattern is consistent with the general observation that system-level scheduling and memory-management optimizations, rather than model-level changes alone, tend to dominate practical LLM serving efficiency gains (Zhou et al., 2024).

5. Conclusion

This study designed and implemented a private AI cloud architecture based on two VMs with an NVIDIA Tesla T4 GPU running llama.cpp, the compressed MoE model Qwen3.6-28B-REAP20-A3B (Q4_K_M), and providing an OpenAI-Compatible API service. Test results reveal three main findings. First, in a single-processing-slot configuration (--parallel 1), throughput reached a plateau of around 38.5 tokens/second. At the same time, latency increased linearly, even though GPU utilization remained around 60%, consistent with a queue-serialization bottleneck. Second, enabling continuous batching with 4 processing slots (--parallel 4) increased throughput by about 1.7× and reduced latency by about 35%. At the same time, GPU utilization remained below maximum capacity, indicating the limiting factor still lies outside GPU computation. Third, agentic-type load with long prompts increased GPU utilization to 97–100% peak (60–63% mean), making GPU computational capacity during prefill the primary limiting factor. All test scenarios achieved a 100% protocol success rate (600 s timeout); practically viable capacity is up to 3 concurrent users under both SLO thresholds simultaneously, or up to 10 if only the response-time criterion is considered.

These results show that adding processing slots effectively increases service capacity for chat load. For agentic-type load, however, service capacity is more strongly influenced by GPU computational capability, so capacity planning needs to account for prefill-heavy characteristics.

Data availability

All raw test data and benchmarking scripts are included with this article. Access: https://gitlab.com/btgdev/data-benchmark-performa-private-ai-cloud.

Declaration of competing interest

The authors declare no known competing financial interests or personal relationships that could have influenced the work reported in this article.

Authors' contributions

MRBJ designed the system architecture and implemented the system; LF designed the testing protocol, conducted benchmarking, and analyzed the data. MRBJ supervised the research. Finally, all authors have reviewed and approved of the final manuscript.

Declaration of generative AI in scientific writing

During the preparation of this work, the authors used ChatGPT to assist with language refinement. After using this tool/service, the authors reviewed and edited the content as needed and took full responsibility for the content of the published article.

Acknowledgements

The authors thank Politeknik Artificial Intelligence Budi Mulia Dua for infrastructure support for this research.

References

Chen, K., Zhou, X., Lin, Y., Feng, S., Shen, L., & Wu, P. (2025). A survey on privacy risks and protection in large language models. In *Journal of King Saud University - Computer and Information Sciences* (Vol. 37, Number 7). Springer International Publishing. https://doi.org/10.1007/s44443-025-00177-1

Chen, Q., Chen, X., & Huang, K. (2025). SlimCaching: Edge Caching of Mixture-of-Experts for Distributed Inference. http://arxiv.org/abs/2507.06567

Crompton, H., Burke, D., Nickel, C., Bozkurt, A., Miao, F., Sharples, M., et al. (2026). Governing generative AI in higher education: A global Delphi study on policy and practice. *International Journal of Educational Technology in Higher Education*, 23(1), 21. https://doi.org/10.1186/s41239-026-00602-z

Fedus, W., Zoph, B., & Shazeer, N. (2022). Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120), 1–39.

Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2022). GPTQ: Accurate post-training quantization for generative pre-trained transformers. arXiv. https://doi.org/10.48550/arXiv.2210.17323

Kakolyris, A. K., Masouros, D., Xydis, S., & Soudris, D. (2024). SLO-Aware GPU DVFS for Energy-Efficient LLM Inference Serving. *IEEE Computer Architecture Letters*, *23*(2), 150–153. https://doi.org/10.1109/LCA.2024.3406038

Khalil, A., Heilles, G., Parraga, M., & Heilles, S. (2025). *Viability and Performance of a Private LLM Server for SMBs: A Benchmark Analysis of Qwen3-30B on Consumer-Grade Hardware*. https://doi.org/10.48550/arXiv.2512.23029

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)* (pp. 611–626). ACM. https://doi.org/10.1145/3600006.3613165

Lasby, M., Lazarevich, I., Sinnadurai, N., Lie, S., Ioannou, Y., & Thangarasa, V. (2025). REAP the Experts: Why Pruning Prevails for One-Shot MoE compression. http://arxiv.org/abs/2510.13999

Malakhov, K. S. (2025). Deploying LLMs on CPU-only Environments with llama.cpp Library Set: MedLocalGPT Project Case. CEUR Workshop Proceedings (repository: https://github.com/knowledge-ukraine/medlocalgpt)

Morell-Mengual, V., Fernández-García, O., Berenguer, C., Ortega-Barón, J., Gil-Llario, M. D., & Estruch-García, V. (2025). Characteristics, motivations and attitudes of students using ChatGPT and other language model-based chatbots in higher education. *Education and Information Technologies*, *30*(15), 22257–22274. https://doi.org/10.1007/s10639-025-13650-1

Nyamsuren, E. (2025). Evaluating quantized Large Language Models for code generation on low-resource language benchmarks. *Journal of Computer Languages*, *84*, 101351. https://doi.org/10.1016/J.COLA.2025.101351

Rakka, M., Fouda, M. E., Khargonekar, P., & Kurdahi, F. (2024). A Review of State-of-the-art Mixed-Precision Neural Network Frameworks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *46*(12), 7793–7812. https://doi.org/10.1109/TPAMI.2024.3394390

Stuhlmann, L., Argerich, M. F., & Fürst, J. (2025). Bench360: Benchmarking Local LLM Inference from 360 Degrees. http://arxiv.org/abs/2511.16682

Wang, J., Du, H., Niyato, D., Kang, J., Xiong, Z., Kim, D. I., & Letaief, K. B. (2025). Toward Scalable Generative Ai via Mixture of Experts in Mobile Edge Networks. *IEEE Wireless Communications*, *32*(1), 142–149. https://doi.org/10.1109/MWC.003.2400046

Wang, Z., Li, S., Zhou, Y., Li, X., Gu, R., Cam-Tu, N., Tian, C., & Zhong, S. (2024). Revisiting SLO and goodput metrics in LLM serving. arXiv. https://doi.org/10.48550/arXiv.2410.14257

Wu, B., Zhong, Y., Zhang, Z., Liu, S., Liu, F., Sun, Y., Huang, G., Liu, X., & Jin, X. (2026). FastServe: Iteration-level preemptive scheduling for large language model inference. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)* (pp. 57–74). USENIX Association.

Xiang, Y., Li, X., Qian, K., Zhang, Y., Yu, W., Zhai, E., Jin, X., & Zhou, J. (2026). ServeGen: Workload characterization and generation of large language model serving in production. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)* (pp. 1845–1859). USENIX Association.

Yan, B., Li, K., Xu, M., Dong, Y., Zhang, Y., Ren, Z., & Cheng, X. (2025). On protecting the data privacy of Large Language Models (LLMs) and LLM agents: A literature review. In *High-Confidence Computing* (Vol. 5, Number 2). Shandong University. https://doi.org/10.1016/j.hcc.2025.100300

Yigci, D., Eryilmaz, M., Yetisen, A. K., Tasoglu, S., & Ozcan, A. (2025). Large Language Model-Based Chatbots in Higher Education. In *Advanced Intelligent Systems* (Vol. 7, Number 3). John Wiley and Sons Inc. https://doi.org/10.1002/aisy.202400429

Yu, G. I., Jeong, J. S., Kim, G. W., Kim, S., & Chun, B. G. (2022). Orca: A distributed serving system for Transformer-based generative models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (pp. 521–538). USENIX Association.

Zhang, Z. J., Shi, J., & Tang, S. (2025). Cloud or On-Premises? A Strategic View of Large Language Model Deployment. https://doi.org/10.2139/ssrn.5296479

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., & Zhang, H. (2024). DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (pp. 193–210). USENIX Association.

Zhou, Z., Ning, X., Hong, K., Fu, T., Xu, J., Li, S., Lou, Y., Wang, L., Yuan, Z., Li, X., Yan, S., Dai, G., Zhang, X.-P., Dong, Y., & Wang, Y. (2024). A survey on efficient inference for large language models. arXiv. https://doi.org/10.48550/arXiv.2404.14294

Zhu, X., Li, J., Liu, Y., Ma, C., & Wang, W. (2024). A Survey on Model Compression for Large Language Models. *Transactions of the Association for Computational Linguistics*, *12*, 1556–1577. https://doi.org/10.1162/tacl_a_00704

Mabrur Roh Bintang Jaya received the Bachelor's degree in informatics engineering from Universitas Islam Negeri Sunan Kalijaga, Yogyakarta, Indonesia, in 2015, and the Master's degree in information technology from Universitas Teknologi Digital Indonesia, Yogyakarta, Indonesia, in 2023.

He is currently a Lecturer at Politeknik Artificial Intelligence Budi Mulia Dua, Yogyakarta, Indonesia. He has professional and academic experience in software engineering, information systems development, and web-based system implementation. His research interests include artificial intelligence, AI infrastructure, software engineering, information systems, data science, and web-based systems.

Lia Farokhah received her Ph.D. degree in Computer Science from Institut Teknologi Sepuluh Nopember (ITS), Surabaya, Indonesia. She is currently a Lecturer and Researcher at the Department of Applied Data Science, Politeknik Artificial Intelligence Budi Mulia Dua, Yogyakarta, Indonesia. Her research interests include data science, neurocomputing, and artificial intelligence.