applied sciences

MDPI

Article

Benchmarking Ollama and vLLM for Concurrent LLM Serving: A Multi-Scenario Evaluation of Performance and Scalability

Betül Ay * and Yunus Emre Demirdağ

Department of Computer Engineering, Fırat University, Elazığ 23200, Türkiye; demirdag.emre.y@gmail.com
* Correspondence: betulay@firat.edu.tr

Abstract

Serving LLMs to many concurrent users gives rise to significant challenges for latency, throughput, and scalability. This paper presents a systematic and reproducible benchmark of two widely used large language model (LLM) serving frameworks, Ollama and vLLM, under concurrent workloads. We conducted a controlled comparison by running both frameworks on a single NVIDIA H100 80 GB GPU with the same model weights (Qwen3-4B) and inference configurations. We then evaluated them using five open benchmark datasets across four scenarios consisting of baseline question answering, complex reasoning, streaming interaction, and stress testing. Each scenario was executed under increasing concurrency levels, ranging from light loads to high-concurrency stress, to measure end-to-end latency, throughput, time-to-first-token (TTFT), success rate, and resource usage. Our experiments show that vLLM clearly outperforms Ollama across all four scenarios, achieving a 100% request success rate. It delivers 20–29 times higher throughput and 8–19 times lower P95 latency, and it completes every request successfully. Additionally, vLLM produced the first token within 0.5–3.5 s and remained stable at up to 100 concurrent users. Ollama, by contrast, required 54–122 s for the first token, hit a concurrency bottleneck near 10 users, and exhibited a timeout-based error rate of 13–30.06% under heavier loads. Notably, both frameworks demonstrated significant memory growth during extended endurance tests, necessitating careful monitoring in long-running deployments. Overall, for a single model on a single GPU, vLLM is highly suitable for high-concurrency serving, while Ollama remains a practical choice for lightweight, local, or developmental workflows.

Keywords: large language model serving; Ollama; vLLM; latency optimization; concurrent request handling; continuous batching; FP8 quantization; PagedAttention; stress testing; performance evaluation; Qwen3; NVIDIA H100

Academic Editors: Yoonsik Choe and Tymoteusz I. Miller

Received: 18 April 2026
Revised: 26 May 2026
Accepted: 27 May 2026
Published: 29 May 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

LLMs have become one of the most important advances in artificial intelligence in recent years. With models such as GPT, LLaMA, and Mistral, LLMs are now used in many areas, including natural language understanding, text generation, question answering, code synthesis, and decision-support systems [1]. Over time, LLMs have moved from being mainly research tools to being practical solutions used in real production systems [2]. This change has made the efficient, scalable, and low-latency deployment of LLMs an important research and engineering problem [3].

Using LLMs in production brings several challenges. At first, these models needed large hardware with high-memory GPUs and expensive clusters, so they were hard to use for organizations with limited resources. To solve this, several techniques were developed to reduce model size, improve parameter efficiency, and speed up inference. For example, methods such as Low-Rank Adaptation (LoRA) [4], model distillation [5], and quantization [6] let LLMs run with fewer resources while keeping a reasonable balance between accuracy and performance.

Even with these methods, latency and scalability under concurrent users are still major problems in real systems [3,7]. For this reason, these problems led to the development of special inference frameworks that aim to improve deployment efficiency. Frameworks such as Hugging Face Transformers [8], Deepspeed-Inference [9], and TensorRT [10] try to solve these issues. However, they often do not give consistent efficiency and scalability for large production workloads.

In this context, two frameworks have recently received much attention: Ollama [11] and vLLM [3]. Ollama is a lightweight framework that focuses on easy integration for developers and efficient execution on local and edge devices. vLLM, on the other hand, introduces techniques such as PagedAttention and optimized scheduling, which give higher throughput and better use of hardware in large-scale serving. To the best of our knowledge, there are still a few peer-reviewed comparisons of Ollama and vLLM under controlled concurrent workloads. Thus, there is a need for systematic, multi-scenario benchmarking of these frameworks under realistic concurrent load.

This study addresses this gap by systematically comparing Ollama and vLLM under controlled, concurrent conditions on a single NVIDIA H100 GPU (NVIDIA Corporation, Santa Clara, CA, USA) rented from the RunPod cloud platform (RunPod, Inc., Dover, DE, USA; https://www.runpod.io/, accessed on 19 May 2026). We design four deployment scenarios, namely baseline performance, complex reasoning, streaming interaction, and stress testing, that simulate production use, and then we measure key metrics such as end-to-end latency, throughput, TTFT, success rate, and resource usage. The results give insights into the strengths and limits of each framework, and they offer practical guidance for choosing a deployment strategy for different applications.

This article is organized as follows. Related work on efficient LLM inference and serving frameworks is reviewed in Section 2. The experimental methodology, including the hardware and software environment, the evaluation metrics, and the four test scenarios, is elaborated in Section 3. The experimental results for all four scenarios, together with the overall comparative analysis, are presented in Section 4. Section 5 presents a discussion of the findings, the resource–performance trade-offs, and the causes of the observed performance gap. Section 6 concludes the paper and outlines directions for future work.

2. Related Works

Recent studies show that the performance of LLMs in production depends strongly on system-level optimizations, not only on model architecture [1–3]. In particular, memory management, batching, and scheduling play a key role in latency and throughput [12,13]. In this section, we want to provide a general summary of the main methods proposed to make LLM inference more efficient. The first group of methods targets the cost of a single inference operation. For example, FlashAttention reduces memory use during attention computation [14], while FlexGen enables high-throughput inference by offloading work across CPU, GPU, and disk [15]. Quantization is also widely used to reduce computation and memory [6,16]. Among modern serving frameworks, vLLM stands out with its PagedAttention mechanism, which improves KV-cache management

and handles concurrent requests efficiently [3]. This approach greatly increases throughput and scalability, especially under high load.

Other high-performance systems, such as Hugging Face TGI and TensorRT-LLM, focus on optimized batching and hardware acceleration. Benchmarking studies show that these systems perform better than general-purpose runtimes under large workloads [17,18]. Ollama, by contrast, is designed for simplicity and easy deployment. It lets developers run LLMs locally with little setup and provides a user-friendly interface [11]. However, its performance under high-concurrency workloads has not been studied much in the literature.

A more recent line of work focuses on the scheduling layer of LLM inference. Yu et al. [19] introduced iteration-level scheduling in ORCA, which lets requests enter and leave a batch dynamically instead of waiting for the whole batch to finish. This idea, now known as continuous batching, is the basis of most modern serving frameworks, including vLLM. Building on this, Agrawal et al. [20] proposed Sarathi-Serve, which combines chunked prefill with stall-free scheduling so that new requests do not interrupt ongoing decodes. The chunked-prefill mechanism is also available in vLLM through the enable-chunked-prefill flag and is part of our setup (Section 3.1.4).

Another direction separates the two phases of inference instead of merging them. DistServe runs prefill and decode on different GPU pools, so each phase can be scaled on its own [7], and Splitwise reports similar gains by splitting the phases across machines [21]. SGLang improves the throughput and latency of complex LM programs through new optimizations [22] and reports large throughput gains over vLLM and TGI on workloads with heavy prefix sharing. Together, these works show that modern serving frameworks differ in basic design choices, which makes direct comparison important.

Some other methods speed up inference at the decoding step. Speculative decoding uses a smaller model to draft several tokens at once, which the target model then checks in a single step. Medusa removes the need for a separate draft model by adding several decoding heads to the base model [23], while EAGLE performs speculation at the feature level instead of the token level and reports higher acceptance rates than earlier methods [24]. These methods reduce per-token latency without changing the model weights, but their real speedup depends on how the serving framework schedules drafted and verified tokens.

A separate line of work studies LLM inference outside the cloud. For example, Husom et al. [25] evaluate 28 quantized models from the Ollama library on a Raspberry Pi 4 and measure energy use, accuracy, and latency for different quantization levels. MELT is a mobile benchmarking framework that runs LLMs on Android, iOS, and Jetson devices and reports end-to-end latency, memory pressure, and energy use [26]. These studies confirm that lightweight runtimes such as Ollama are useful on resource-limited hardware. However, they do not study how the same runtimes behave on server-class hardware under concurrent multi-user workloads, which is the setting we study in this paper.

Even though both frameworks are widely used, direct comparisons between Ollama and vLLM are still limited. Most existing work studies either high-performance serving systems or local deployment tools separately [27,28]. Some prior work benchmarks local LLM runtimes on specific hardware, such as Apple Silicon [28], but does not directly compare local deployment frameworks with high-performance serving systems such as vLLM. Therefore, a unified evaluation that compares these frameworks under the same conditions is needed. We fill this gap with a unified evaluation that compares the two frameworks under identical conditions across several scenarios, namely baseline, reasoning, streaming, and stress, which are largely unexplored in the literature.

3. Methodology

3.1. Experimental Environment

To make the comparison of Ollama and vLLM fair and reproducible, we ran all experiments on a high-performance server. Here we provide the hardware and software details.

3.1.1. Hardware Configuration

The system uses an Intel Xeon Platinum 8470 processor (Intel Corporation, Santa Clara, CA, USA), which is an enterprise-class CPU. During idle benchmarking, the average CPU load was about 9%. The system has 233.8 GiB of DDR5 RAM, and only 3.533 GiB was used at baseline. This confirms that background processes had little effect on the experiments.

For inference, we used a single NVIDIA H100 80 GB HBM3 GPU (NVIDIA Corporation, Santa Clara, CA, USA). The GPU has 80 GiB of HBM3 memory, supports a maximum power draw of 700 W, and runs at the P0 performance state. Figure 1 presents the GPU status, obtained with NVIDIA-SMI. As Figure 1 confirms, the GPU was idle with 0% utilization and almost no memory use at the time of capture, which confirms a clean baseline. The storage consists of a 100 GB system disk (4% used) and a 250 GB data volume (31 GB, 13% used). This is enough space for model weights, Docker images, and experiment outputs.

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.126.09             Driver Version: 580.126.09     CUDA Version: 13.0     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA H100 80GB HBM3          On  |   00000000:9A:00.0 Off |                    0 |
| N/A   29C    P0             69W /  700W |       0MiB /  81559MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
```

Figure 1. GPU information of the test environment, obtained via NVIDIA-SMI.

3.1.2. Software Environment

The server runs Ubuntu 24.04.3 LTS (Noble) with Python 3.11 as the main programming language. We used Docker 29.2.1 for containerized deployment, which ensures reproducibility and isolation between experiments. The GPU software stack includes NVIDIA driver version 580.126.09 and CUDA Toolkit 13.0. We used Ollama version 0.24.0 and vLLM version 0.17.0 as the inference frameworks under evaluation. In addition, we used Hugging Face Transformers 4.57.6 and huggingface_hub 0.36.2 for model and dataset access, and psutil 7.2.2 and GPUtil 1.4.0 for system-resource monitoring.

3.1.3. Preparation of the Test Environment

To make the experiments objective and repeatable, we applied a fixed pre-run procedure before each experiment. First, we prefetched all Hugging Face model weights and datasets into a local cache using a dedicated script. This removed network variability during inference. A preflight script then checked the GPU compute capability and automatically selected the right quantization path for vLLM: FP8 native precision on Hopper-class GPUs (compute capability ≥8.9) and GPTQ on older hardware. On the H100 used here, the FP8 native path was selected. This yields an approximately 8-bit weight-equivalent setup, comparable to Ollama's Q8_0 GGUF format, so the quantization baseline is fair between the two frameworks.

Only one framework was active at any time. Between framework changes, we applied a 60 s cooldown, optionally followed by clearing the Linux page cache. Between scenario changes within the same framework, we applied a 15 s cooldown. Each scenario had a 30 s warm-up phase before measurement and a 30–120 s cooldown after it. We excluded all warm-up and cooldown measurements from the reported results.

We checked the GPU temperature before each scenario with a thermal-check script and logged the readings to a file. If the temperature went above 70 °C, we held for 60 s before starting measurements. During the recorded run, the GPU temperature stayed in the 30–32 °C range in all scenarios, which confirms there was no thermal throttling.

We also collected system resource usage, including CPU load, RAM use, and GPU utilization and memory, about once per second using psutil and GPUtil, together with process-level NVIDIA-smi monitoring. We recorded all configuration parameters, git commit hashes, dataset revision pins, preflight decisions, and tool versions in structured environment files for full reproducibility.

3.1.4. Framework-Specific Deployment Configurations

We configured both frameworks with the same Qwen3-4B model weights [29] and the same sampling parameters, so the comparison is fair. The differences come only from internal design choices, such as batching, KV-cache management, and GPU memory handling. In Table 1, we provide the framework-specific settings.

For Ollama, we used the qwen3:4b-q8_0 GGUF model, served on port 11436. Ollama was run with its default setting, which auto-selects between 1 and 4 parallel requests per model based on available memory (*Ollama_Num_Parallel = 4*) [11]; the context length was set to 4096 tokens, and only one model was loaded at a time *(Ollama_Max_Loaded_Models = 1).* The KV-cache used FP16 precision with the default llama.cpp backend [30].

For vLLM, we used the Qwen/Qwen3-4B model from Hugging Face Transformers, served on port 8002. We set GPU memory utilization to 0.90, the maximum concurrent sequences to *max_num_seqs = 64*, and the maximum model length to 4096 tokens. We applied FP8 native quantization and an FP8 KV-cache on the H100. We set tensor parallelism to 1 *(Tensor_Parallel_Size = 1),* because we used a single GPU in all experiments.

We kept the sampling parameters the same for both frameworks through a shared parameter file: temperature 0.7, top-p 0.9, top-k 50, seed 0, and no repetition penalty. We also aligned token counts when needed using the Hugging Face tokenizer through the *Benchmark_Tokenizer = hf* flag.

A single shell script ran the whole pipeline, including framework startup, health checks, scenario execution, cooldown, and result aggregation. After each full run, the script exported the aggregated results to CSV files and paper-ready tables automatically.

Table 1. Controlled experimental configuration for Ollama and vLLM deployments.

Parameter	Ollama	vLLM
Framework Version	v0.24.0	v0.17.0
Model Weights	qwen3:4b-q8_0 (GGUF)	Qwen/Qwen3-4B (HF Transformers)
Quantization	Q8_0 (GGUF default)	FP8 native (H100/Hopper)
Inference Parameters	Temperature: 0.7, Top-p: 0.9, Top-k: 50, Seed: 0, num_ctx: 4096	Temperature: 0.7, Top-p: 0.9, Top-k: 50, Seed: 0, max_tokens: 4096
Batching Strategy	Internal queue via llama.cpp	Continuous batching
KV-Cache	FP16 (llama.cpp default)	FP8 (H100 native)
GPU Memory Allocation	Automatic	gpu_memory_utilization = 0.90
Max Loaded Models	1	1
Warm-up	30 s (excluded from measurements)	30 s (excluded from measurements)

3.1.5. Reproducibility and Experimental Pipeline

We ran the evaluation on deterministic subsamples drawn from five open datasets on the Hugging Face Hub. We used TriviaQA [31] for the baseline and stress scenarios (S1, S4), GSM8K [32] for mathematical reasoning, HumanEval [33] for code generation, HotpotQA [34] for multi-hop question answering, and MT-Bench [35] for multi-turn streaming dialog (S3). Each dataset was matched to its scenario, as shown in Table 2. We fixed the number of samples per scenario at 100 for S1, S2, and S4, and at 32 dialog turns for S3. We did the subsampling deterministically with *Dataset_Seed = 0* using a shuffle-and-select procedure, so the prompt sets are the same across repeated runs.

To remove network variability during inference, we prefetched all dataset splits and model weights into a local cache before each run. We pinned each Hub dataset to a specific commit SHA through environment variables, so silent upstream changes could not affect the results. The revisions we used are listed in Table 2.

Table 2. Dataset assignments and Hub commit SHA pins used across experimental scenarios.

Scenario	Task	Hub ID	Commit SHA
S1, S4—Baseline/Stress	Short-form QA	mandarjoshi/trivia_qa	0f7faf33
S2—Reasoning	Mathematics	openai/gsm8k	740312ad
S2—Reasoning	Code generation	openai/openai_humaneval	7dce6050
S2—Reasoning	Multi-hop QA	hotpot_qa	1908d6af
S3—Streaming	Multi-turn dialog	HuggingFaceH4/mt_bench_prompts	e3a795c5

We evaluated both frameworks one after the other on the same prompt sets, using the same sampling parameters from a shared parity file (temperature 0.7, top-p 0.9, top-k 50, seed 0). Only one inference server was active at any time. A single shell script ran the whole pipeline, covering dataset prefetch, environment logging, framework startup, health checks, scenario execution, cooldown, and result aggregation. We exported the aggregated metrics to CSV files and paper-ready tables with dedicated post-processing scripts.

We recorded all configuration parameters, software versions, dataset revision pins, preflight quantization decisions, and git commit hashes in a structured environment file in each run directory. An independent researcher can reproduce the same prompt set and conditions using the same repository commit and dataset pins. The complete benchmarking framework, including all scripts, Docker configurations, and dataset pin files, is publicly available at: https://github.com/demirdagyunus/serving-compare-tests (accessed on 24 May 2026).

3.2. Evaluation Metrics

Here we provide one common set of metrics collected across all four scenarios to evaluate Ollama and vLLM under different conditions. We measured all metrics from the client side using a shared HTTP streaming driver, so the measurements are consistent and comparable between the two frameworks. The metrics are summarized in Table 3. A wide range of metrics was collected to evaluate Ollama and vLLM across different deployment scenarios. These metrics were grouped into categories reflecting fundamental performance, streaming, token usage, reasoning quality, system resources and scalability.

Core performance metrics capture the basic behavior of the system. They include end-to-end latency, throughput, TTFT, token generation rate, and error rate. End-to-end latency is the wall-clock time from the moment the HTTP request is sent to the moment the full response is received. It includes client overhead, network round-trip time, server-side queuing, prefill, and decode. We do not instrument these parts separately, so all

latency values are client-observed end-to-end measurements. TTFT is the time from the request to the first non-empty token in streaming mode, and we report it for all four scenarios. We compute latency and TTFT only from successful requests. We exclude failed or timed-out requests from the average and percentile values, but we count them in the error rate.

Reasoning and task-specific metrics are reported for Scenario 2. They include heuristic accuracy scores for mathematical reasoning, code generation, and multi-hop question answering. These scores use lightweight proxy methods. They do not use human annotation or LLM-as-judge evaluation, so they are only approximate indicators of output quality under load.

Streaming and conversational metrics are reported for Scenario 3. They include TTFT, tokens per second, conversation completion rate, and bandwidth per request. Coherence and context retention scores use heuristic methods and should be read as approximate indicators, not as exact quality measurements.

System resource metrics measure hardware use during inference. They include CPU load, RAM use, GPU utilization, and GPU memory use. We collected these about once per second using psutil and GPUtil, together with process-level NVIDIA-smi monitoring.

Scalability and stress metrics are reported for Scenario 4. They include the breaking point, recovery behavior, throughput drop under ramp and spike loads, and long-term memory growth during endurance testing. The measurement harness does not export server-internal queue depth or cache hit rates, so we do not report them. Latency under overload reflects both queuing delay and inference time, as seen by the client. We recorded all timings in seconds and converted them to milliseconds in tables and figures unless stated otherwise.

Table 3. Summary of evaluation metrics grouped by category.

Category	Metric	Unit	Scope
Core Performance	End-to-end latency (avg, P95, P99)	ms	All Scenarios
Core Performance	TTFT	ms	All Scenarios
Core Performance	Throughput	req/s or task/s	All Scenarios
Core Performance	Token generation rate	tok/s	All Scenarios
Core Performance	Error Rate	%	All Scenarios
Task Quality	Reasoning accuracy (math, code, QA)	0–1 (heuristic)	S2
Streaming	Conversation completion rate	%	S3
Streaming	Bandwidth per request	KB	S3
Resource	CPU/RAM/GPU utilization	%/MB	All Scenarios
Scalability	Breaking point, recovery time	concurrent users/s	S4
Scalability	Memory growth rate	MB/hr	S4

Table 4 provides the definitions and formulas for the performance metrics. We recorded timing in seconds and converted it to milliseconds in tables and figures unless stated otherwise. We computed latency and TTFT only from successful requests. We excluded failed requests from the averages but counted them in the error rate.

Table 4. Metric formulas and definitions used across all experimental scenarios.

Metric	Description	Formula
Response Latency	End-to-end wall-clock duration from request initiation to final token reception.	$Latency^i = t^i_{last_token} - t^i_{request}$
Average Latency	Mean latency over all successfully completed requests.	$Latency_{avg} = \frac{1}{N}\sum_{i=1}^{N} Latency^i$

Percentile Latency	Latency below which a given percentage of requests are completed.	$Latency_{pX} = Percentile_X(\{Latency_1, \ldots, Latency_N\}), X \in \{50, 75, 90, 95, 99, 99.9\}$
Throughput	Number of successful requests or generated tokens per unit time, NN; N is the number of requests; TT T is total test duration.	$Throughout_{req} = \frac{SuccessfullRequests}{T}$ $Throughput_{tokens} = \frac{Total\ Tokens}{T}$
Error Rate	Ratio of failed requests to total requests. Ratio of failed requests.	$ErrorRate = \frac{FailedRequests}{TotalRequests}\ x\ 100\%$
Time-to-First-Token (TTFT)	Duration from request initiation to reception of the first non-empty token delta. Includes network overhead and represents client-observed responsiveness.	$TTFT^i = t^i_{first_token} - t^i_{request}$
Resource Utilization	CPU, GPU, and memory usage during inference.	Measured via psutil and GPUtil at ~1 s intervals; GPU metrics supplemented by NVIDIA-smi process monitoring.
Performance Degradation	Percentage change in a metric relative to a baseline concurrency level.	Latency-based: increase indicates degradation. Throughput-based: decrease indicates degradation.

Metric Definitions and Measurement Methodology

To keep the metrics clear and consistent, we define all of them from one client-side point of view. Here, the life cycle of a request is shown in Figure 2.

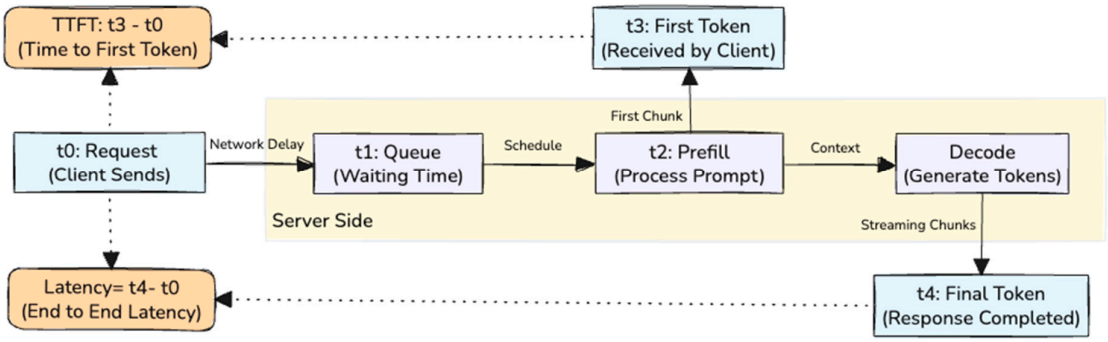


Figure 2. Conceptual request timeline used for metric definitions.

In this timeline, t0 is the moment the client sends the streaming HTTP request, t3 is the moment the first non-empty token is received, and t4 is the moment the full streamed response is finished. The middle phases (network transmission, server-side queuing, and inference, that is, prefill and decode) are shown only conceptually. The measurement harness does not instrument these phases separately. For this reason, all reported latency and TTFT values are client-observed end-to-end measurements that include all of these parts.

End-to-end latency is measured as t4 − t0: the wall-clock time from the request to the final token. In Scenario 1, we report this as latency. In Scenarios 2, 3, and 4, we report it as completion time. Both refer to the same client-observed measurement.

Time-to-first-token (TTFT) is measured as t3−t0: the time from the request to the first non-empty token. All four scenarios use streaming, so we report TTFT for all of them. TTFT includes network overhead, server-side scheduling delay, and prefill. It reflects client-perceived responsiveness, not isolated model inference time.

We send all requests over HTTP streaming. The timer origin (t0) is recorded just before the streaming request is sent in the client driver. We compute latency and TTFT averages only from successful requests. We count failed and timed-out requests in the error rate but exclude them from latency, TTFT, and token-rate values.

Throughput is the number of successful requests divided by the total test duration, and it should be read together with the error rate. The token generation rate is the number of completion tokens divided by the end-to-end time per request.

Quality-oriented metrics (reasoning quality in Scenario 2, and response coherence, response quality, and context retention in Scenarios 3 and 4) use lightweight heuristic scoring. These scores do not use human annotation or LLM-as-judge evaluation. They should be read as approximate proxy indicators of output quality under load, not as validated quality measurements.

The measurement harness does not separately instrument server-side queue waiting time and pure inference time. Therefore, we make no claims about how the observed latency splits into queuing and inference. All reported values are the combined client-observed measurement.

3.3. Statistical Analysis

We ran each scenario as a single continuous run at each concurrency level and aggregated the per-request statistics over the full test duration. In addition to average values, we also report percentile distributions (P50, P75, P90, P95, and P99), minimum and maximum values, and variability-aware plots to make the results easier to read.

We analyzed latency and TTFT with percentile-based statistics, because inference latency under concurrent load is not Gaussian. This follows common practice in tail-latency analysis of large-scale serving systems [36]. We also use error bars and range plots where useful to show request-level variability.

We exclude failed and timed-out requests from the latency and TTFT averages but count them in the error rate. The harness does not instrument server-side queue time and inference time separately, so all reported latency values are combined client-observed end-to-end measurements, as defined in the Metric Definitions and Measurement Methodology.

Several distributions are heavily skewed and unstable under overload, especially for Ollama at high concurrency. For this reason, our analysis is mainly descriptive rather than based on formal parametric significance tests.

3.4. Test Scenarios

To study the behavior of Ollama and vLLM under different conditions, we designed four scenarios. Each one focuses on a different aspect of deployment: baseline performance, complex reasoning, streaming quality, and stress resilience. All scenarios use the same HTTP streaming client path, the same sampling parameters, and a single GPU. Here, the only difference between the frameworks is the inference backend itself.

3.4.1. Scenario 1: Baseline Performance (Simple Q&A)

In this scenario, we measure the basic performance of both frameworks on short factual question-answering tasks from the TriviaQA dataset [31]. It focuses on latency, throughput, and error rates under different concurrency levels.

- Model: Qwen3-4B.
- Concurrency: 10, 50, 100 requests.
- Duration: 180 s per level.
- Warm-up: 30 s (excluded from measurements).

- Metrics: End-to-end latency (avg, P95, P99), TTFT, throughput (req/s), token generation rate, error rate, and system resource use.

3.4.2. Scenario 2: Complex Reasoning

In this scenario, we evaluate three types of multi-step reasoning tasks, namely mathematical problem solving [37] (GSM8K), code generation [38] (HumanEval), and multi-hop question answering [39] (HotpotQA). It focuses on task completion time, heuristic reasoning accuracy, and resource use under moderate concurrency.

- Model: Qwen3-4B.
- Concurrency: 20, 50, 100 tasks.
- Duration: 180 s per level.
- Warm-up: 30 s; cooldown: 60 s.
- Metrics: End-to-end completion time, TTFT, heuristic accuracy (math, code, QA), token generation rate, error rate, and memory use.

3.4.3. Scenario 3: Streaming and Conversational

In this scenario, we study streaming responsiveness and conversation quality. It uses multi-turn dialogs from the MT-Bench [35] prompt set. We evaluate both frameworks on their ability to deliver tokens in real time and to keep context across conversation turns.

- Model: Qwen3-4B.
- Concurrency: 50–150 conversations.
- Duration: 900 s per level.
- Warm-up: 30 s; cooldown: 60 s.
- Metrics: TTFT, tokens per second, conversation completion rate, heuristic coherence and response quality scores, context retention, and bandwidth per request.

3.4.4. Scenario 4: Stress and Scalability

In this scenario, we find breaking points, scalability limits, and long-term stability. It uses three steps, namely a gradual ramp test, a spike test, and a long endurance phase. We push both frameworks under rising load until performance drops or fails. The experiments have been conducted on the TriviaQA dataset.

- Model: Qwen3-4B.
- Concurrency: 10–500 (gradual ramp); fixed load during the endurance phase.
- Duration: 300 s per ramp level; 3600 s for endurance.
- Warm-up: 30 s; cooldown: 120 s.
- Metrics: Breaking point (concurrent users), recovery time, throughput drop, memory growth rate (MB/hr), and error rate.

Each scenario started with a 30 s warm-up phase that we excluded from the results. Scenarios 1 and 2 ran for 180 s per level, while Scenario 3 ran for 900 s per level to capture conversation behavior under sustained load. Scenario 4 used 300 s per ramp level and then a 3600 s endurance phase. We applied cooldown periods of 30, 60, 60, and 120 s after Scenarios 1, 2, 3, and 4, respectively.

4. Experimental Results

We conducted four sets of experiments to demonstrate how Ollama and vLLM behave under increasing concurrency. In this section, we present the results of the four scenarios described in Section 3.4. The goal is to compare Ollama and vLLM under different conditions, from baseline performance to complex reasoning, streaming, and stress testing, at the concurrency levels listed in Table 5.

Table 5. Scenario summary.

Scenario	Objective	Dataset	Concurrency	Key Metrics
1. Baseline Performance	Evaluate latency, throughput, and error rates under simple Q&A tasks	TriviaQA	10, 50, 100	Latency (avg, P95, P99), TTFT, throughput, error rate, resource usage
2. Complex Reasoning	Assess reasoning accuracy and resource utilization in multi-step tasks	GSM8K, HumanEval, HotpotQA	20, 50, 100	Completion time, TTFT, accuracy (math, code, QA), memory usage
3. Streaming & Conversational	Measure streaming responsiveness and dialog quality	MT-Bench	50, 150	TTFT, tokens/sec, coherence, conversation completion, bandwidth
4. Stress & Scalability	Identify breaking points, recovery behavior, and long-term stability	TriviaQA	10–500 (gradual), endurance	Breaking point, recovery time, memory growth, throughput degradation

For each scenario, we present both performance metrics (latency, throughput, error rate, TTFT, and accuracy) and system-level measurements (CPU, GPU, and memory use). We also give statistical summaries with mean, standard deviation, and percentile values (P95, P99) to show variability. We ran each scenario as a single continuous run at each concurrency level and aggregated the per-request statistics over the full test duration. We use line charts and bar plots to show trends, and tables to show detailed values. This lets us compare the two frameworks at both a fine and a high level.

4.1. Scenario 1: Baseline Performance (Simple Q&A)

In the first set of experiments, we evaluated the baseline performance of Ollama and vLLM on simple Q&A tasks from the TriviaQA dataset, using the Qwen3-4B model. We ran each concurrency level (10, 50, 100 requests) for 180 s, with a 30 s warm-up excluded from the results. We set the maximum output length to 256 tokens. The metrics were end-to-end latency, throughput, TTFT, token generation rate, error rate, and system resource use.

Table 6 summarizes the baseline results for the three concurrency levels. The values include average and percentile latencies, throughput, token generation rate, TTFT, error rate, and resource use. All values are aggregated over the 180 s runs. As Table 6 indicates, vLLM substantially outperforms Ollama in latency, throughput, and token generation, while Ollama degrades quickly and starts to fail beyond moderate concurrency.

Table 6. Baseline performance test summary for Ollama and vLLM across concurrency levels.

Concurrency	Platform	Avg Latency (s)	P95 Latency (s)	P99 Latency (s)	Throughput (req/s)	Token/sec	TTFT Avg (s)	Error Rate	CPU (%)	Memory (%)
10	Ollama	20.92	26.65	26.89	0.45	1.53	20.75	0.0	9.3	17.0
10	vLLM	3.71	3.74	4.11	2.61	499.33	0.07	0.0	3.9	35.5
50	Ollama	89.72	114.03	114.58	0.46	0.16	89.70	0.0	10.1	17.2
50	vLLM	4.02	4.15	4.26	11.93	2266.30	0.11	0.0	4.7	40.0
100	Ollama	100.48	126.09	128.81	0.45	0.65	100.40	39.29	10.3	17.1
100	vLLM	6.37	7.94	8.50	15.20	2891.13	2.25	0.0	6.0	40.5

Figure 3a presents the end-to-end latency of both frameworks on a logarithmic scale, which makes the order-of-magnitude gap clear. vLLM keeps its latency below 10 s even at 100 concurrent requests, while Ollama degrades sharply: its average latency rises from about 21 s at 10 users to over 100 s at 100 users, and its P99 latency passes 128 s. At every

concurrency level, vLLM's average latency is roughly 5–25 times lower than Ollama's, and the gap grows with concurrency. The log scale also shows that vLLM's latency grows slowly, while Ollama's rises sharply between 10 and 50 users. Figure 3b reveals that vLLM sustains far higher throughput than Ollama at all levels. vLLM scales from 2.61 req/s at 10 users to 15.20 req/s at 100 users, while Ollama plateaus at about 0.45 req/s. This indicates that Ollama's internal queue cannot handle more requests in parallel beyond its set limit, while vLLM's continuous batching scales throughput with concurrency over the tested range.

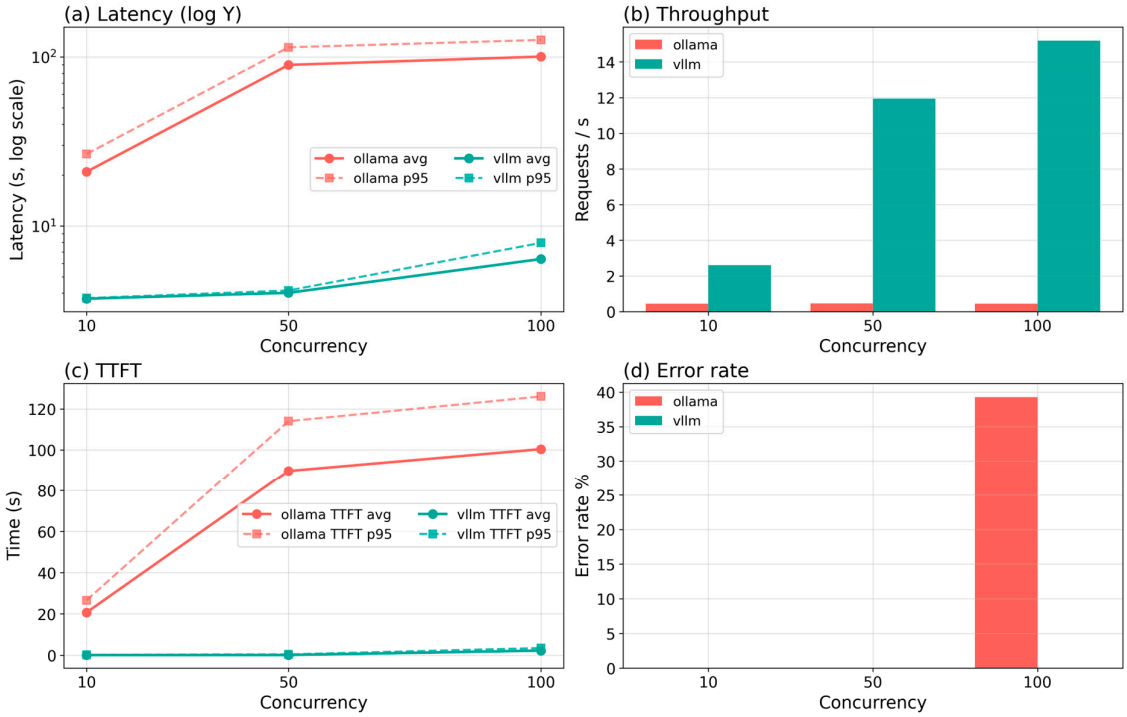


Figure 3. (**a**) End-to-end latency (log scale), (**b**) throughput, (**c**) time-to-first-token (TTFT), and (**d**) error rate of Ollama and vLLM across concurrency levels.

It can be seen from Table 6, which reports the token generation rate, that vLLM reaches 499 to 2891 tokens per second and scales with concurrency, while Ollama stays below 2 tokens per second at all levels. At peak concurrency, this is about a three-orders-of-magnitude difference. We observed that vLLM's PagedAttention and continuous batching turn concurrent demand into real generation throughput much better than Ollama's llama.cpp backend.

Figure 3c illustrates the TTFT difference. vLLM keeps TTFT below 3 s even at 100 users, while Ollama's TTFT grows with concurrency and reaches over 100 s at the highest level. This matters for interactive use, where TTFT decides how fast the system feels. The results indicate that Ollama cannot start generating tokens quickly under load, because requests pile up in its internal queue before prefill can start.

Figure 3d reports the error rate at each level. vLLM had a 0% error rate at all levels and completed all 5498 requests. Ollama had no errors at 10 and 50 users, but at 100 users, it had a 39.29% timeout-based failure rate, with 88 of 224 requests failing to finish within the timeout. As Figure 3d indicates, Ollama's effective concurrency limit for baseline Q&A on this hardware is between 50 and 100 users.

Figure 4 summarizes the resource use of both frameworks. Ollama uses slightly more CPU (about 9–10%) than vLLM (4–6%), but it produces much less output for it. GPU use

is much higher for vLLM (35–41%) than for Ollama (17%), which indicates that vLLM uses the GPU better through continuous batching and FP8 native execution on the H100. Memory use is similar for both (about 6%). Hence, the main difference lies in computational efficiency, not memory use. Together, these results demonstrate that vLLM [illegible] ing the

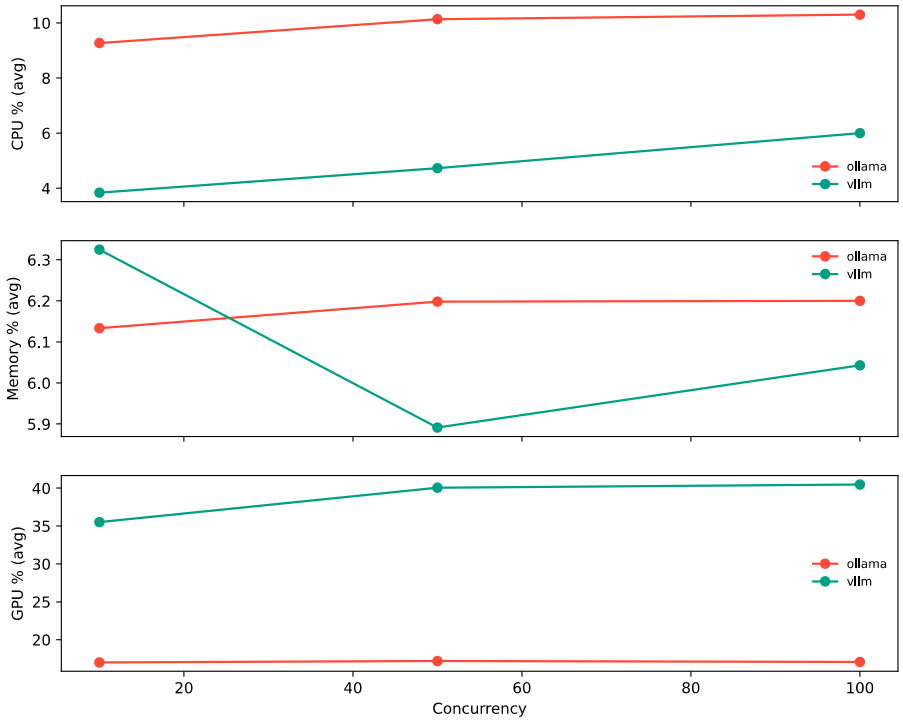


Figure 4. System-level resource utilization of Ollama and vLLM across concurrency levels.

The baseline experiments confirm that vLLM delivers much higher throughput (15.20 vs. 0.45 req/s at 100 users), lower latency (below 10 s vs. over 100 s average), and a 0% error rate, while Ollama fails 39.29% of requests. For this reason, vLLM is the more efficient framework for concurrent Q&A under the tested conditions.

We also looked at variability to better describe the behavior at each level. The percentile statistics (P50–P99) and min–max ranges show that vLLM stayed very stable as concurrency rose, with standard deviations below 1 s at all levels. We further observe that Ollama exhibited much larger variance and wider ranges, with a standard deviation of nearly 16 s at 100 users. We excluded failed requests from the latency values but counted them in the error rate. Overall, vLLM delivers more predictable and scalable performance, while Ollama becomes unstable beyond moderate concurrency. Table 7 summarizes the latency variability for Scenario 1. For each concurrency level, it reports the average latency and its spread for both frameworks. The standard deviations (SD) in the summary tables are approximate values estimated from the P95 latency. They show only a rough spread, while the percentiles (P50–P99) give a more reliable measure of variability.

Table 7. Additional statistical summary for Scenario 1.

Concurrency	Platform	Avg Latency (s)	SD (s)
10	Ollama	20.92	3.48
10	vLLM	3.71	0.02
50	Ollama	89.72	14.78

50	vLLM	4.02	0.08
100	Ollama	100.48	15.57
100	vLLM	6.37	0.96

4.2. Scenario 2: Complex Reasoning

In the second set of experiments, we evaluated Ollama and vLLM on multi-step reasoning tasks from three Hugging Face datasets: GSM8K for mathematical problem solving, HumanEval for code generation, and HotpotQA for multi-hop question answering. We ran each concurrency level (20, 50, 100 tasks) for 180 s, with a 30 s warm-up excluded. The metrics were end-to-end completion time, TTFT, heuristic accuracy by task type, throughput, token generation rate, error rate, and resource use.

Table 8 summarizes the reasoning results for the three concurrency levels. The values include average and percentile completion times, throughput, token generation rate, overall accuracy, average reasoning steps, error rate, and GPU use. All values are aggregated over the 180 s runs. As Table 8 shows, vLLM outperforms Ollama on every metric, and the gap grows with concurrency.

Table 8. Complex reasoning performance summary for Ollama and vLLM across concurrency levels.

Concurrency	Platform	Avg Completion (s)	P95 (s)	P99 (s)	Throughput (Task/s)	Token/sec	Overall Accuracy	Reasoning Steps	Error Rate (%)	GPU (%)
20	Ollama	41.87	46.60	46.83	0.44	32.29	0.00	0.00	0.00	16.6
20	vLLM	3.81	3.86	3.93	5.02	1535.80	0.46	15.11	0.0	39.1
50	Ollama	93.75	119.03	120.17	0.43	27.62	0.00	0.00	0.0	16.8
50	vLLM	4.02	4.13	4.56	12.03	3648.36	0.44	15.07	0.0	42.2
100	Ollama	101.75	126.30	128.38	0.44	29.61	0.00	0.00	40.0	16.8
100	vLLM	6.19	7.57	7.92	15.55	4692.32	0.45	15.09	0.0	43.7

Figure 5a illustrates the completion-time difference. vLLM keeps the completion time below 7 s even at 100 tasks, while Ollama rises from about 42 s at 20 tasks to over 101 s at 100 tasks. Notably, vLLM parallelizes reasoning work through continuous batching, while Ollama runs tasks one after another beyond its set limit. Figure 5b presents throughput and token rate together. vLLM scales from 5.02 task/s and 1536 tok/s at 20 tasks to 15.55 task/s and 4692 tok/s at 100 tasks, which is nearly linear scaling. Ollama plateaus at about 0.44 task/s and 27–32 tok/s. This indicates that Ollama's scheduling cannot turn extra demand into reasoning throughput.

Figure 5c breaks down the heuristic accuracy by task type. vLLM achieves consistent accuracy at all levels. The overall accuracy is 0.44–0.46, with multi-hop QA being the strongest (0.57–0.61), code being moderate (0.43–0.45), and mathematics being the weakest (0.32). Ollama reports 0.00 accuracy for all task types and levels. This most likely comes from heavy response truncation under load and timeout-induced incomplete outputs, not from purely wrong answers. This result should be read together with the high completion times and the 40% error rate at 100 tasks.

Figure 5d reports the average number of reasoning steps per task. vLLM produces about 15 steps per task at all levels, which reflects stable chain-of-thought behavior regardless of load. Ollama shows 0 reasoning steps, which matches its accuracy results: under high latency and timeout pressure, its outputs were either incomplete or did not contain detectable reasoning.

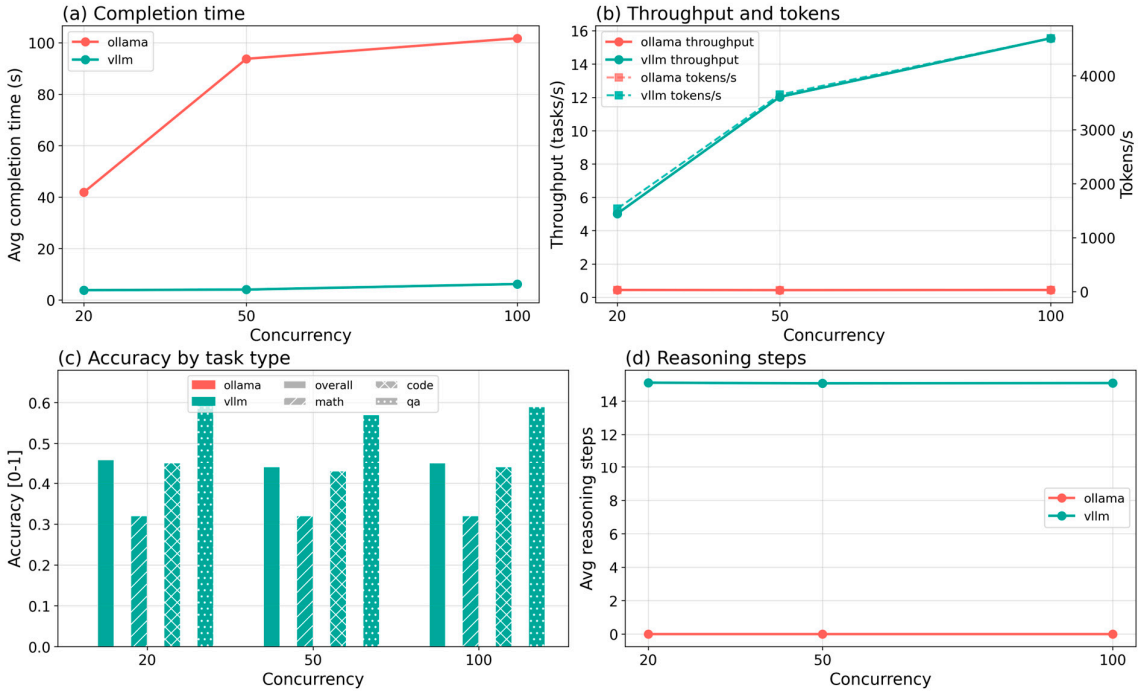


Figure 5. (**a**) Average task completion time, (**b**) throughput and token generation rate (dual-axis), (**c**) accuracy breakdown by task type (overall, mathematics, code, multi-hop QA), and (**d**) average reasoning steps per task, for Ollama and vLLM across concurrency levels.

Figure 6 reports the total number of tasks processed by each framework by task type. The large gap, with vLLM processing 6044 tasks and Ollama only 442, comes from the throughput gap, not from sampling, because both frameworks drew from the same prompt pool with the same task-type mix. At 100 tasks, Ollama completed only 132 of 220 attempted tasks because of timeouts, while vLLM completed all 2916 attempted tasks.

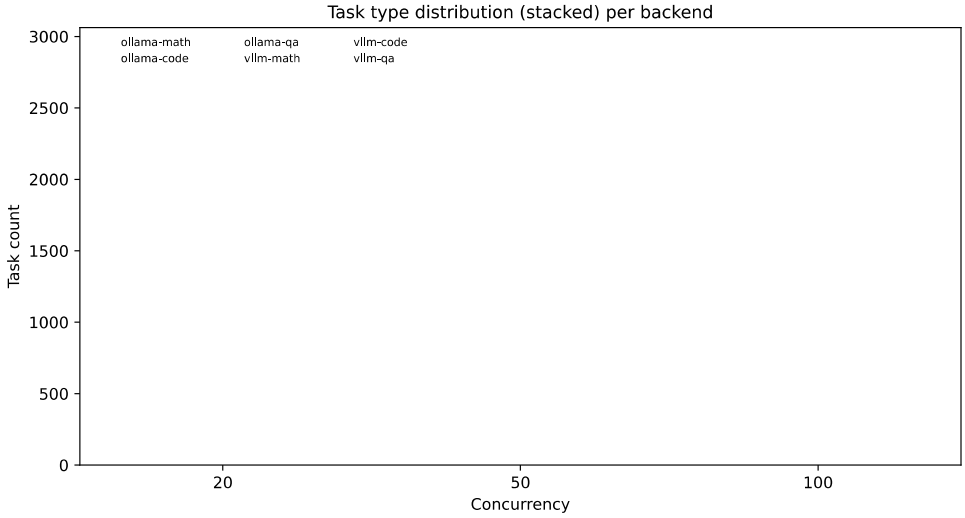


Figure 6. Task type distribution (math, code, QA) processed by each framework across concurrency levels.

The reasoning experiments confirm that vLLM delivers an order-of-magnitude higher throughput (15.55 vs. 0.44 task/s at 100 tasks), 16 times faster completion time (6.19 s vs. 101.75 s), and a 0% error rate, while Ollama fails 40% of tasks by timeout. We also

https://doi.org/10.3390/app16115435

find that vLLM kept measurable accuracy for all task types, while Ollama's outputs did not meet the heuristic correctness criteria under the tested conditions.

Table 9 summarizes the latency variability for Scenario 1.The reasoning workload produced very skewed completion-time distributions, especially for Ollama under overload. Therefore, we examined percentiles and variability as well as the averages. Ollama showed a large variance in completion time because of timeouts and scheduling instability at 100 tasks, while vLLM stayed stable at all levels, with standard deviations below 1 s. We excluded failed and timed-out requests from the completion time values but counted them in the error rate. Overall, vLLM delivers more consistent reasoning performance under sustained load.

Table 9. Additional statistical summary for Scenario 2.

Concurrency	Platform	Avg Completion (s)	SD (s)
20	Ollama	41.87	2.87
20	vLLM	3.81	0.03
50	Ollama	93.75	15.37
50	vLLM	4.02	0.07
100	Ollama	101.75	14.93
100	vLLM	6.19	0.84

4.3. Scenario 3: Streaming and Conversational

In the third set of experiments, we evaluated how Ollama and vLLM handle real-time streaming using multi-turn dialogs from the MT-Bench prompt set. We ran each concurrency level (50 and 150 conversations) for 900 s with a 30 s warm-up excluded. We set the conversation length to five turns with a maximum output of 256 tokens per response. The metrics were TTFT, end-to-end completion time, token generation rate, request and conversation throughput, bandwidth use, streaming chunk counts, response coherence, and context retention. Streaming matters in production systems such as chatbots, voice assistants, and customer support, where latency affects user experience. Multi-turn dialog is itself an active research direction, with recent surveys reviewing advances in LLM-based dialog systems and the challenge of maintaining responsiveness and context across conversation turns [40].

Table 10 summarizes the streaming results. The values include average and percentile completion times, TTFT, throughput, token rate, conversation completion, error rate, and heuristic quality scores. As Table 10 shows, vLLM substantially outperforms Ollama on almost every metric, especially TTFT, token rate, and reliability.

Table 10. Streaming and conversational test summary for Ollama and vLLM across concurrency levels.

Platform	Avg Completion (s)	TTFT Avg (s)	P95 (s)	Throughput (Req/s)	Throughput (Conv/s)	Avg Tokens/s	Bandwidth/Req (KB)	Coherence	Context Retention	Error Rate (%)
Ollama @50	101.92	101.31	116.86	0.47	0.06	0.58	0.26	0.74	0.07	0.00
vLLM @50	4.29	0.13	0.22	11.36	0.55	57.56	1.37	0.78	1.00	0.0
Ollama @150	118.17	117.54	128.27	0.46	0.14	0.47	0.30	0.76	0.07	61.29
vLLM @150	9.64	5.51	6.77	15.29	1.57	26.56	1.35	0.81	1.00	0.0

Figure 7a presents the completion time and TTFT for both frameworks. vLLM keeps the completion time below 10 s and TTFT below 6 s at 150 conversations. Ollama's completion times are over 100 s, and its TTFT is almost the same as its completion time. Notably, Ollama produces its output as a single block rather than streaming it. This fits

with Ollama's queue filling up before token streaming can start, so most of the wall-clock time is spent waiting in the queue rather than delivering tokens.

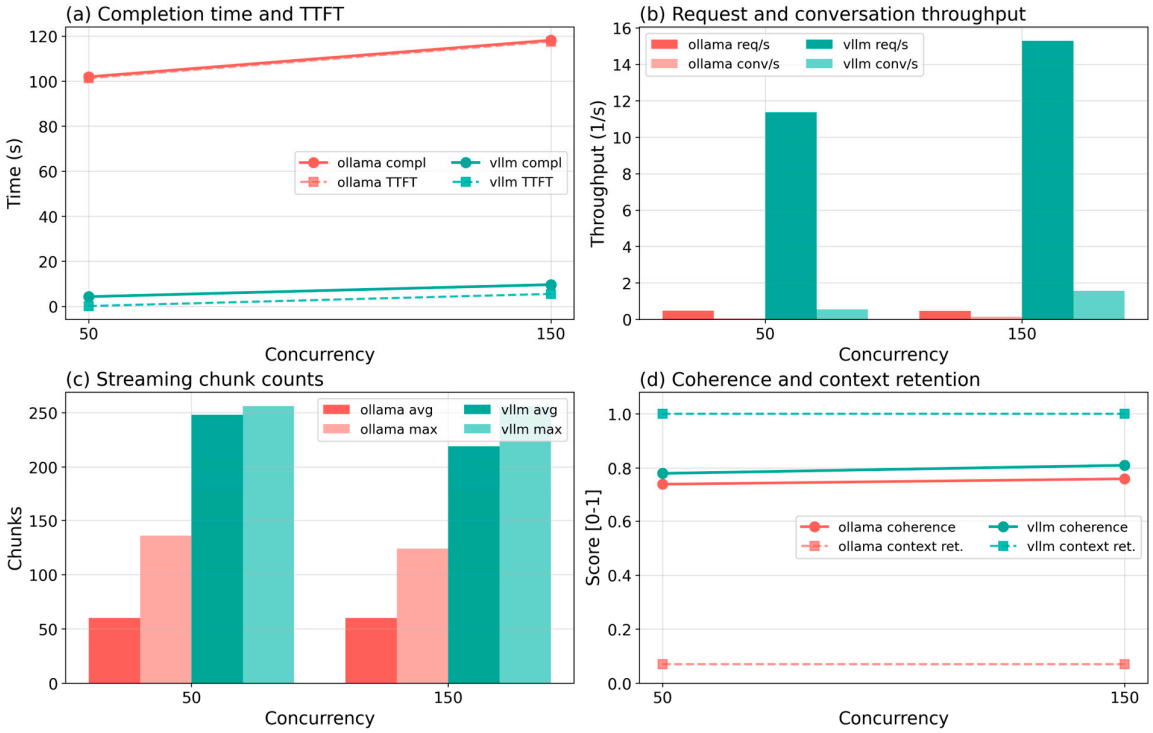


Figure 7. (**a**) Completion time and time-to-first-token (TTFT), (**b**) request and conversation throughput, (**c**) streaming chunk counts (average and maximum), and (**d**) heuristic response coherence and context retention scores of Ollama and vLLM across streaming concurrency levels.

The token-rate distribution shows a similar disparity. vLLM reaches 57.56 tokens per second at 50 conversations and 26.56 at 150 conversations, with P99 values of 33.49 to 69.82 tok/s. Ollama stays below 1 token per second at both levels, with P99 values of 0.91–1.02 tok/s. This 50–60 times difference shows the large gap in real-time generation between the two frameworks.

Figure 7b presents the request and conversation throughput together. vLLM scales from 11.36 req/s and 0.55 conv/s at 50 conversations to 15.29 req/s and 1.57 conv/s at 150 conversations. Ollama stays flat at about 0.46 req/s, with a conversation throughput of 0.06 to 0.14 conv/s. This large gap confirms that Ollama cannot turn concurrent demand into streaming throughput. Bandwidth per request also differs between the two frameworks: vLLM sends about 1.35–1.37 KB per request, while Ollama sends 0.26–0.30 KB per request. This difference comes from vLLM's longer and more complete responses: vLLM produced 16.3–18.1 million tokens during the test windows, compared with 30,000–34,000 tokens for Ollama. Hence, Ollama's lower bandwidth is not an advantage; it is the result of producing far less output per request.

Figure 7c reports the number of streaming chunks per request. vLLM produces 219–248 chunks per request on average, with a maximum of 256, which reflects fine-grained streaming consistent with the streaming_chunk_size = 1 setting. Ollama produces about 60 chunks on average, with a maximum of 124–136, which reflects both shorter responses and coarser streaming. This matches the TTFT result: Ollama's streaming under load is hurt by upstream queue delays.

Figure 7d presents the heuristic coherence and context-retention scores. The coherence scores are similar for both frameworks (Ollama: 0.74–0.76; vLLM: 0.78–0.81),

which means that on the requests Ollama did finish, the text quality was reasonable. However, the context retention scores show a large gap: vLLM reaches 1.00 at both levels, while Ollama scores only 0.07. This is because Ollama completed only a small share of the multi-turn conversations (57 of 239 at 50 concurrent, 141 of 673 at 150 concurrent), so it could not build up context across turns.

The streaming experiments show a clear gap in reliability and responsiveness. At 150 conversations, Ollama had a 61.29% timeout-based error rate (741 of 1209 requests failed), while vLLM had a 0% error rate across all 24,187 streaming requests. The TTFT gap is large: vLLM delivers the first token in 0.13–5.51 s, while Ollama needs 101–118 s, a 20–800 times gap depending on concurrency. For this reason, Ollama is not suitable for real-time streaming at the tested levels, while vLLM keeps production-grade streaming throughout.

The streaming workloads produced non-uniform latency distributions as concurrency rose. Therefore, we used percentile statistics (P50–P99) as well as the averages for TTFT and streaming latency. vLLM kept much lower TTFT variance and more stable streaming, especially at high concurrency. At 150 conversations, Ollama's TTFT had a standard deviation of over 6 s even though its mean was held near the 120 s timeout, while vLLM kept sub-second TTFT variance. The min–max ranges also show that Ollama had large response-time instability during streaming, which hurts the user experience. Table 11 summarizes TTFT statistics for the streaming scenario. It reports the average TTFT and its standard deviation for both frameworks at two concurrency levels. The values show how stable the TTFT was during streaming.

Table 11. Additional statistical summary for Scenario 3.

Platform	AVG TTFT (s)	SD (s)
Ollama @50	101.31	9.46
Ollama @150	117.54	6.53
vLLM @50	0.13	0.06
vLLM @150	5.51	0.77

4.4. Scenario 4: Stress Testing and Scalability

In the fourth set of experiments, we evaluated the resilience and scalability of both frameworks under high concurrent load. We used three steps: a gradual ramp test that raised concurrency from 10 to 500 users, a spike test that simulated sudden load surges, and a 3600 s endurance phase at a steady moderate load. Together, these steps show both short-term breaking points under rising load and long-term stability under steady use. Unlike the previous version of this study, both Ollama and vLLM completed all stress phases, so we can compare them directly across the full pipeline.

Table 12 summarizes the gradual load results. The values include total requests, successful requests, throughput, average and P95 latency, error rate, and whether the breaking point criterion was triggered at each level. Ollama reached its breaking point at the first level (10 users), while vLLM stayed stable up to 200 users before its performance dropped.

Table 12. Gradual load test results for Ollama and vLLM across concurrency levels.

Platform	Load	Requests	Successful	Throughput (Req/s)	Avg Latency (s)	P95 Latency (s)	Error Rate	Breaking Point
Ollama	10	146	146	0.46	21.13	26.21	0.0	True
vLLM	10	780	780	2.57	3.74	3.80	0.0	False
vLLM	25	1882	1882	6.18	3.86	3.94	0.0	False
vLLM	50	3561	3561	11.70	4.06	4.17	0.0	False
vLLM	100	4819	4819	15.67	5.97	7.24	0.0	False
vLLM	200	4872	4872	15.46	12.23	14.34	0.0	True

As Figure 8 shows, vLLM scales well: throughput rises from 2.57 req/s at 10 users to a peak of 15.67 req/s at 100 users, then levels off near 15.46 req/s at 200 users while latency rises from about 6 s to over 12 s. This indicates that vLLM's continuous batching uses the GPU efficiently up to about 100 users, after which extra load builds up in the queue instead of adding throughput. Ollama, by contrast, reaches its breaking point already at 10 users, with average latency over 21 s and throughput of only 0.46 req/s. This confirms that Ollama cannot handle even modest concurrent serving on this hardware.

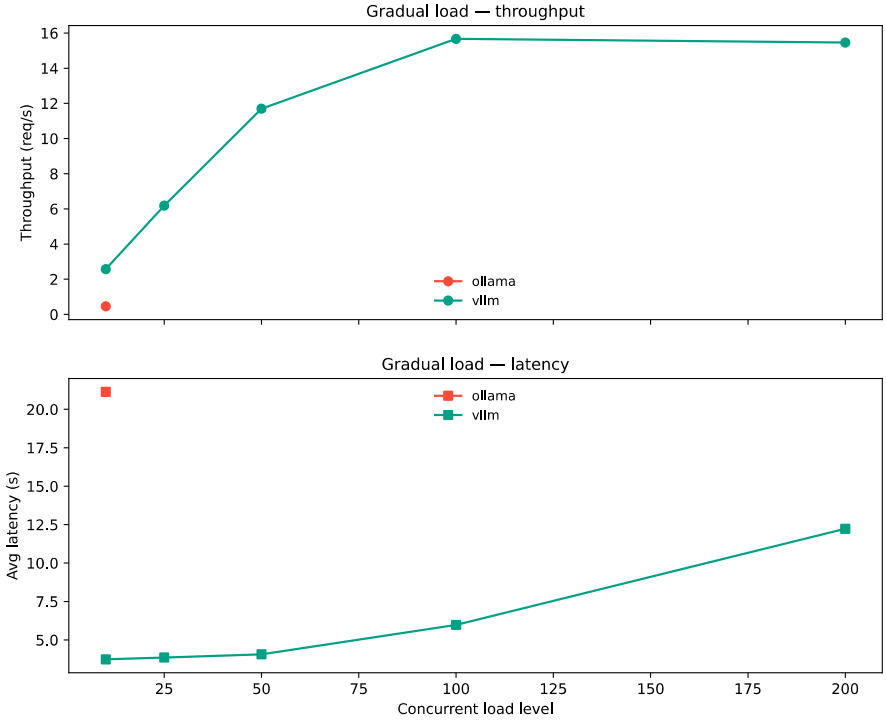


Figure 8. Gradual load test results showing throughput (**top**) and average latency (**bottom**) of Ollama and vLLM as a function of concurrent load level. Dashed vertical lines indicate the breaking point load for each framework.

Figure 9 presents the spike test results. vLLM recovers in about 63–64 s after each spike at all tested peak loads (10–200 users), which indicates predictable recovery. Ollama needed about 164 s to recover from a spike at 10 users (over 2.5 times longer than vLLM) and could not be tested at higher spike loads, because it cannot sustain even baseline concurrency. The peak error rate stayed at 0% for both frameworks during the spike phases that finished.

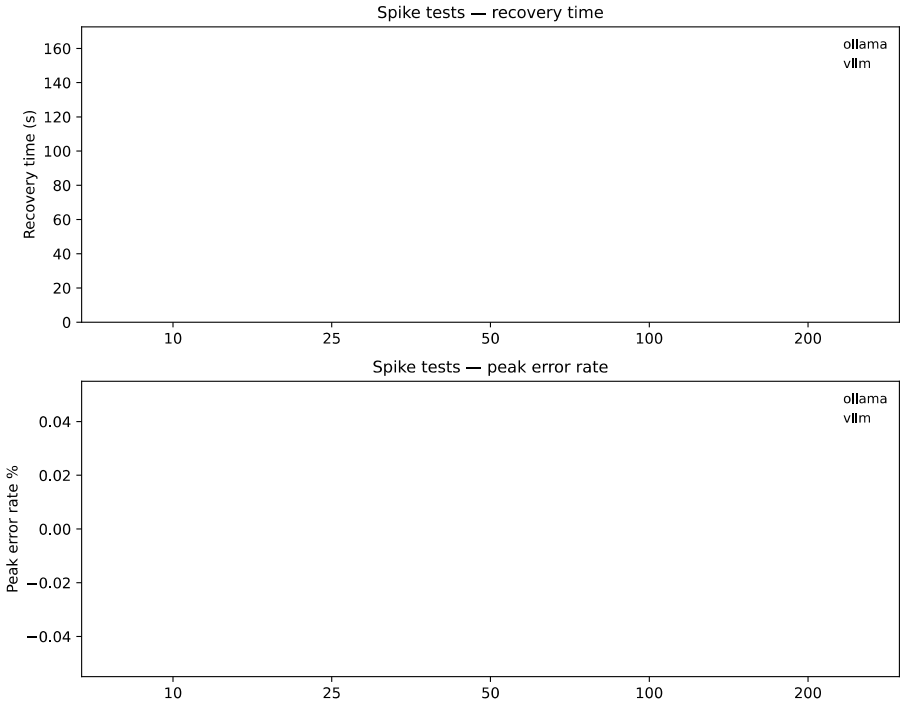


Figure 9. Spike test results: recovery time after sudden load surges (**top**) and peak error rate during spikes (**bottom**) for Ollama and vLLM.

As Figure 10 shows, the endurance test reveals different resource patterns over the 60 min window. Ollama's CPU use is around 10–12%, with frequent spikes above 25%, while vLLM runs at 4–6% with occasional peaks. GPU use is the opposite: vLLM stays at about 35–45% throughout, which reflects good continuous batching, while Ollama stays around 15–20%, which indicates that it underuses the GPU. Memory use for both frameworks rises gradually rather than steadily: vLLM shows step-wise changes between about 5–7%, while Ollama stays near 6.5–6.8% with small drift.

Total memory growth during the endurance phase differs in magnitude (see Figure 10); vLLM grew by about 14,219 MB over the 3600 s window, while Ollama grew by about 5602 MB. Overall, both frameworks exhibit notable memory growth during sustained use, which should be monitored in production. We note that this is the cumulative increase seen by the resource monitor, not a steady-state leak rate. For vLLM, the higher growth is partly explained by KV-cache and batching buffers expanding under much higher throughput than Ollama.

During the gradual load phase, vLLM keeps consistency scores between 0.81 and 0.98 at all loads, with the lowest value (0.81) at 100 users, which is the throughput-peak point. Its quality scores stay in the 0.64–0.68 range. Ollama, tested only at 10 users, scored 0.75 and 0.82 for quality and consistency, respectively. Hence, performance degradation under stress mainly affects throughput and latency, not the quality of completed requests.

In the endurance phase, vLLM kept about 15.7 req/s with an average completion time of 6 s across the window and completed 56,345 requests. Ollama reached about 0.5 req/s with an average completion time of 20.7 s and completed 1722 requests. This 30 times throughput gap during sustained use confirms vLLM's advantage for long-running workloads.

The stress results show that the two frameworks degrade very differently under rising concurrency. We observe that vLLM scaled up to 200 users before its performance dropped, and its main degradation was rising latency and queue buildup, not failure.

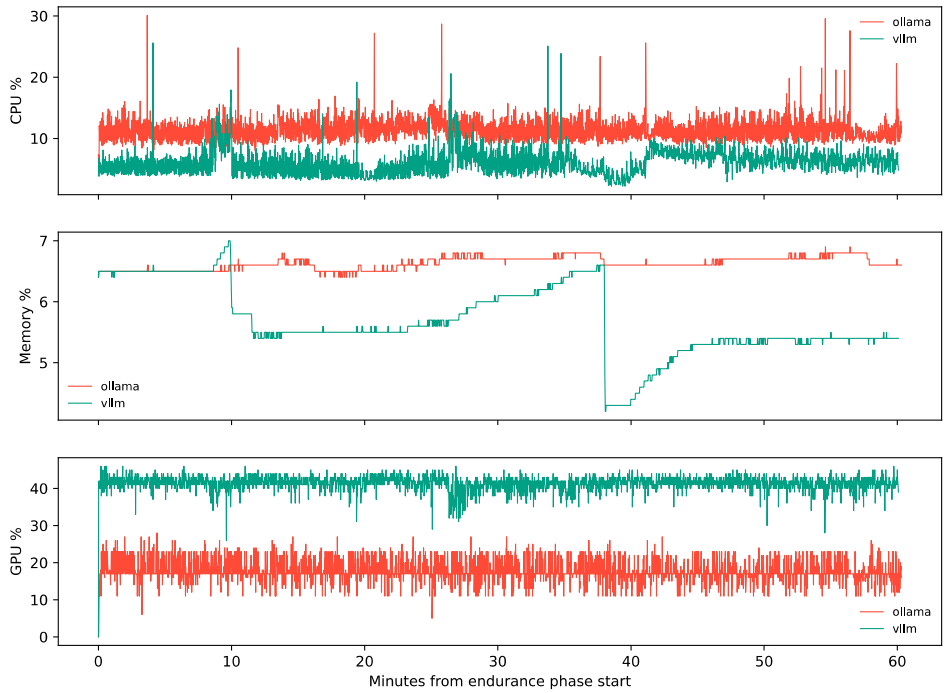


Figure 10. Endurance phase resource timeline (CPU, memory, GPU utilization) of Ollama and vLLM over 60 min of sustained load.

As Table 13 shows, the two frameworks differ by about an order of magnitude in stable operating concurrency. vLLM completed all gradual, spike, and endurance tests with a 0% error rate across 95,108 tracked requests, while Ollama completed only 2162 tracked requests under the same conditions. The main concerns for vLLM in long-running use are the observed memory growth and the latency increase beyond 100 users, both of which should be monitored in production.

Table 13. Observed failure thresholds and breaking points during stress testing.

Service	Concurrency Range	Degradation Zone	Breaking Point	Dominant Failure Behavior
Ollama	≤10 users	10–25 users	≥10 users	Throughput stagnation, queue saturation, and extended recovery after spikes
vLLM	≤100 users	100–200 users	≥200 users	Gradual latency increase and queue buildup; no request failures

The stress experiments show clear differences in stability and scalability. Both frameworks finished all gradual, spike, and endurance phases without request failures, but at very different operating points: Ollama held about 0.5 req/s, while vLLM held 15–16 req/s. Under sustained load, both frameworks showed resource growth that should be monitored in production. The analysis also shows that vLLM's degradation is mainly gradual latency increase and queue buildup, not sudden failure, while Ollama's main limit is a hard ceiling on concurrent serving capacity set at the lowest tested load. Table 14 reports the combined Scenario 4 results, including total requests, average completion time, throughput, memory growth, and error rate for both frameworks.

Table 14. Additional statistical summary for Scenario 4.

Service	Total Requests	Avg Completion (s)	Throughput (Req/s)	Memory Growth (MB)	Error Rate (%)
Ollama	1722	20.70	0.48	5602	0.0
vLLM	56,345	6.07	15.65	14,219	0.0

4.5. Overall Comparative Analysis of Scenarios

Across the four scenarios, there is a clear and large gap between Ollama and vLLM. Here, Table 15 brings together the main metrics across all scenarios and provides a single view of throughput, latency, token rate, success rate, and error rate.

Table 15. Comprehensive performance comparison of Ollama and vLLM across all experimental scenarios.

Scenario	Service	Success Rate (%)	Avg Latency (s)	Throughput (req/s)	Tokens/s	Error Rate (%)
Baseline Performance	Ollama	80.2	70.37	0.45	0.78	13.1
	vLLM	100.0	4.70	9.91	1885.59	0.0
Complex Reasoning	Ollama	80.1	79.12	0.44	29.84	13.3
	vLLM	100.0	4.67	10.87	3292.16	0.0
Streaming	Ollama	55.8	110.05	0.46	0.52	30.06
	vLLM	100.0	6.97	13.33	42.06	0.0
Stress Testing	Ollama	100.0	20.70	0.46	1.85	0.0
	vLLM	100.0	5.97	11.04	49.59	0.0

In Scenario 1 (baseline), vLLM achieved about 22 times higher throughput (9.91 vs. 0.45 req/s) and about 17 times lower P95 latency (5.28 s vs. 88.92 s) than Ollama, with no failed requests. In Scenario 2 (reasoning), vLLM achieved about 25 times higher throughput (10.87 vs. 0.44 task/s) and about 19 times lower P95 completion time (5.19 s vs. 97.31 s). Ollama showed 0.00 heuristic accuracy for all task types because of truncation and timeouts, while vLLM kept measurable accuracy (0.45 overall) and a steady number of reasoning steps (about 15 per task).

Scenario 3 (streaming) showed the largest reliability gap. vLLM kept a 100% success rate, while Ollama dropped to 55.8% on average and to 38.7% at 150 conversations because of read-timeout failures. The TTFT gap was very large: vLLM produced the first token in 0.13–5.51 s, while Ollama needed 101–118 s, a P95 TTFT advantage of about 35 times for vLLM. In Scenario 4 (stress and endurance), both frameworks finished all phases without failures, but at very different operating points: Ollama's limit was reached at 10 users, while vLLM scaled to 200 users before its performance dropped.

As Figure 11 shows, vLLM outperforms Ollama on all four core metrics. Its throughput is about an order of magnitude higher in every scenario, its P95 latency and P95 TTFT are about two orders of magnitude lower, and its error rate stays at 0% while Ollama's error rate ranges from 13.1% (baseline) to 30.06% (streaming). Figure 11a compares throughput across all scenarios. vLLM holds 9.91–13.33 req/s depending on the workload, while Ollama plateaus at 0.45–0.51 req/s in every scenario. Ollama's throughput ceiling seems to be set by its queue configuration, not by the hardware, which matches the GPU use results (Ollama: about 17%; vLLM: 35–69%).

Figure 11b,c show latency behavior. Ollama's P95 latency ranges from 54.5 s (stress) to 122.5 s (streaming), while vLLM's stays in the 5.0–7.9 s range in all scenarios. The TTFT P95 gap is larger: Ollama produces the first token in 54–122 s, while vLLM does it in 0.5–3.5 s, a 20–75 times advantage.

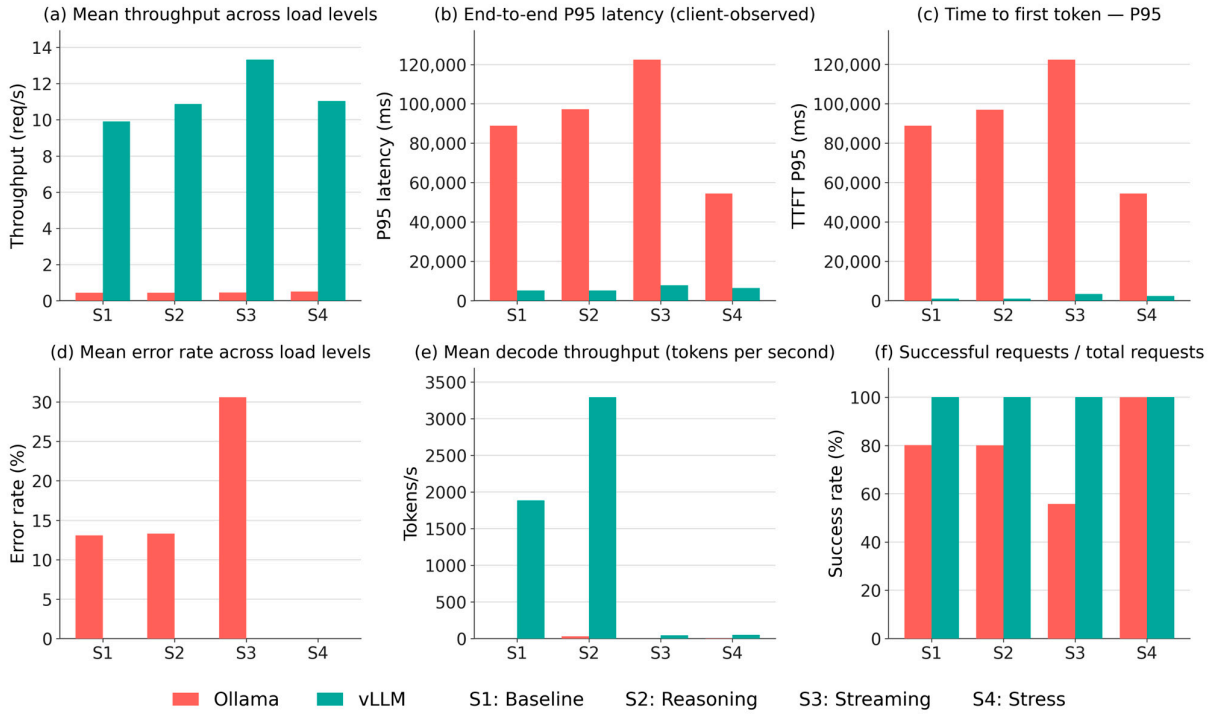


Figure 11. Cross-scenario performance comparison of Ollama and vLLM across all four test scenarios, showing (**a**) mean throughput, (**b**) end-to-end P95 latency, (**c**) P95 time-to-first-token, (**d**) mean error rate, (**e**) mean decode throughput, and (**f**) success rate (successful/total requests).

Figure 11d demonstrates the mean error rate. vLLM kept a 0% error rate in all scenarios and completed all 99,802 evaluated requests across the four scenarios. Ollama's error rate ranges from 0% at low stress test concurrency to 30.06% in streaming, which indicates that its errors depend on the workload and get worse under multi-turn streaming with long connections. Figure 11e gives the mean decode throughput. vLLM reaches 1886 tok/s in baseline, 3292 tok/s in reasoning, and 42 tok/s in streaming, while Ollama stays below 30 tok/s in all scenarios and below 2 tok/s in baseline and streaming. The reasoning scenario yields the highest decode throughput for vLLM, reflecting effective batching of long-output tasks. Figure 11f reports the success rate. vLLM reaches 100% in all four scenarios, while Ollama's reliability varies: 80% in baseline and reasoning, 56% in streaming, and 100% in stress testing. Because the stress test was run at Ollama's effective limit of 10 users, not at its failure point.

Figure 12a compares the heuristic quality proxy scores. Where both frameworks produced enough output to score (Scenarios 3 and 4), Ollama and vLLM are similar, with vLLM slightly ahead in both coherence and response quality (0.79 vs. 0.75 and 0.65 vs. 0.63, respectively). In Scenario 2, Ollama produced no scorable reasoning output because of truncated responses under load, while vLLM reached a reasoning quality score of 0.47. Hence, the gap between the two frameworks stems mainly from throughput, latency, and reliability, not from the quality of the output on completed requests.

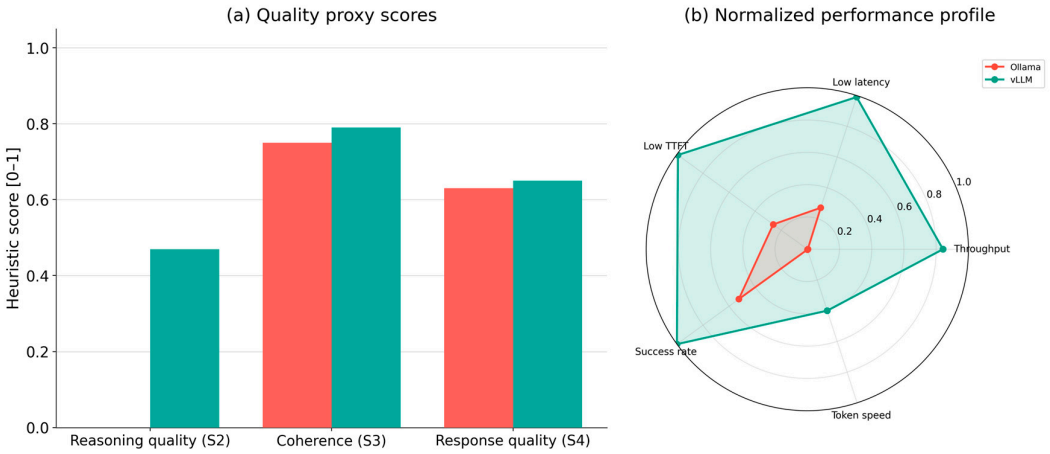


Figure 12. (**a**) Heuristic quality proxy scores across scenarios for Ollama and vLLM (reasoning quality in S2, response coherence in S3, response quality in S4); (**b**) normalized performance profile of the two frameworks (min–max scaling across scenarios). In both panels, red denotes Ollama and teal denotes vLLM.

Figure 12b presents a radar plot of the normalized performance profile of both frameworks. As Figure 12b shows, vLLM reaches or approaches the upper bound on every axis, while Ollama stays near the center on all of them. We observe that vLLM does not trade one dimension for another; it keeps a balanced advantage in throughput, latency, responsiveness, reliability, and token rate.

Figure 13 presents a heatmap of vLLM's advantage ratios. The throughput advantage is 20.3 to 28.9 times across scenarios, the P95 latency advantage is 8.0 to 18.8 times, the P95 TTFT advantage is 19.8 to 74.5 times, and the decode-throughput advantage is 38 times to over 2400 times (in Scenario 1, where Ollama's token rate fell below 1 tok/s while vLLM passed 1800 tok/s). The smallest ratios are in Scenario 4, where Ollama was run at its limit rather than under stress, which narrows the comparison to the only range where Ollama still works. Thus, vLLM is not only faster on average but also more predictable in tail latency.

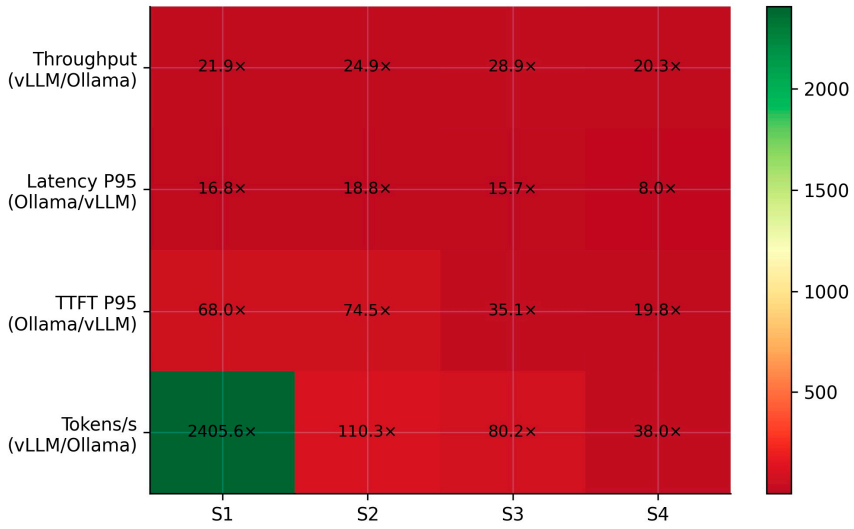


Figure 13. Heatmap of vLLM advantage ratios across scenarios and metrics, showing throughput, P95 latency, P95 TTFT, and decode throughput.

Taken together, these results demonstrate that vLLM is more scalable, more resource-efficient, and more reliable for high-concurrency LLM serving under the tested

workloads, with 20–29 times higher throughput and 8–19 times lower latency than Ollama in all four scenarios. Ollama is simpler to deploy and provides similar output quality on completed requests, but it cannot keep throughput, stability, and latency under concurrent load beyond lightweight settings. Overall, from a systems perspective, vLLM is the more robust and scalable choice for production LLM serving in the tested ranges, while Ollama is still a practical option for lightweight, local, or developer use where concurrent serving is not a main requirement. Detailed logs, raw outputs, benchmark scripts, and configuration files are in the supplementary repository to support reproducibility and result checking.

5. Discussion

5.1. Which Framework Is More Suitable in Different Scenarios?

The results demonstrate that vLLM delivers better scalability and reliability in all four scenarios. For this reason, it is the better choice for production systems with concurrent serving, streaming, or stress-heavy demands. Its throughput of 9.9–13.3 req/s across scenarios and its predictable latency below 10 s at up to 100 users give a stable user experience in real-time systems such as chatbots, customer support assistants, and interactive reasoning agents. Its 100% success rate in all scenarios also makes it suitable for production, where request reliability is important.

Ollama, by contrast, is not competitive under concurrent load, but it is still useful for lightweight, local, or single-user deployments. On the test hardware, its effective limit is about 10 users, beyond which throughput stagnates and errors rise. Its simple setup, GGUF model format, and low operational overhead make it suitable for personal assistants, developer prototyping, edge deployment, and settings with little operational complexity. The heuristic quality scores on completed requests (coherence 0.74–0.76, response quality 0.63) indicate that Ollama's output quality on finished requests is similar to vLLM's. This means that its limits come from the serving architecture, not from the model. Hence, Ollama fits small-scale or offline use, while vLLM fits high-concurrency serving much better.

5.2. Resource vs. Performance Trade-Off

The two frameworks turn hardware into serving capacity in different ways. Ollama used slightly more CPU (9–10%) than vLLM (4–6%) across scenarios, but it used less GPU (15–17% vs. 35–69%). This indicates that Ollama underuses the H100, while vLLM gets much more inference work per GPU cycle through continuous batching, PagedAttention, and FP8 native execution. FP8 generation on Hopper-class GPUs has independently been reported to deliver substantial decode-throughput gains over BF16 at comparable accuracy, which is consistent with the higher GPU efficiency we observe for vLLM [41]. This pattern is not specific to our framework or hardware: independent studies evaluating vLLM against a different competitor (Hugging Face TGI), a different model family (LLaMA-2), and different hardware (A100) likewise report that vLLM attains substantially higher throughput and GPU utilization through continuous batching and PagedAttention, while the weaker framework saturates at lower concurrency [17]. This is consistent with our observations relative to Ollama. As a result, vLLM delivers 20–29 times higher throughput and 8–19 times lower P95 latency while using similar or less CPU.

One notable point is the memory growth during endurance testing. Both frameworks grew over the 3600 s window (Ollama: about 5602 MB; vLLM: about 14,219 MB), but vLLM's higher growth happened while it processed about 33 times more requests. Per request, vLLM's memory footprint during endurance testing is much lower, which fits its

FP8 KV-cache strategy. Still, the absolute growth in vLLM should be monitored in long-running use. This aligns with prior work showing that memory pressure and fragmentation under sustained, high-concurrency serving can degrade scalability, motivating dynamic scheduling and migration techniques [42]. This trade-off shows that the choice of framework should match the hardware and the workload: Ollama for resource-limited but low-demand settings, and vLLM for performance-driven production systems with enough GPU memory and monitoring.

5.3. Implications for Deployment Scenarios

From a deployment point of view, the results are clear. vLLM keeps a 100% success rate and predictable latency under concurrent loads up to 100 users, so it is a strong choice for high-concurrency serving under the tested conditions. It supports auto-scaling, load balancing, and fault-tolerant deployment, which are important for large-scale reliability. Its degradation beyond 100 users is gradual latency increase rather than failure, which gives operators a clear warning sign for capacity planning. Memory use should be monitored to handle long-term growth.

Ollama's limits in scalability and stability, in particular its 30.06% error rate in streaming and 13–13.3% error rates in baseline and reasoning at 100 users, restrict it to lightweight or single-user settings. Its simplicity, GGUF model format, and local-first design make it useful where easy integration and developer access matter more than raw performance, such as on-device inference, developer prototyping, or constrained-resource settings. In the end, the choice between Ollama and vLLM should depend on the target environment: vLLM for high-throughput, latency-sensitive serving with concurrent users, and Ollama for lightweight, resource-limited, or edge settings where concurrent serving is not a main need.

5.4. Diagnostic Analysis of Performance Degradation

We observe that the strong degradation of Ollama under concurrent load seems to come from internal queue saturation, limited parallel request handling, and the lack of continuous batching at the inference level. Its effective concurrency limit (10 users in the stress test) is close to its set number of parallel slots, which suggests that the bottleneck is structural, not hardware-bound. Matching such parallelism configurations across frameworks is a recognized baseline-fairness concern when evaluating inference serving systems [43]. As concurrency rose past this limit, request queues filled quickly, response times became dominated by queue waiting, and timeout failures appeared at 100 users in baseline and reasoning (39–40% error rate) and at 150 conversations in streaming (61% error rate).

vLLM, by contrast, kept stable scheduling and continuous batching under the same load, with a 100% success rate at all levels. This strongly suggests that batching strategy, KV-cache management, and request scheduling are the main factors for scalable inference, which agrees with prior work on iteration-level scheduling [19] and chunked-prefill scheduling [20]. The much higher TTFT for Ollama (54–122 s P95 across scenarios, vs. 0.5–3.5 s for vLLM) shows that requests waited a long time before token generation started. Since we measured latency from the client side, these values include queue buildup and serialized inference, not just model compute time.

Ollama's degradation followed a structural overload pattern, not a sudden failure. As concurrency rose, request queues filled quickly, average completion time rose sharply (from 21 s at 10 users to 100 s at 100 users in baseline), and some requests began to time out. However, throughput stayed flat at about 0.45 req/s at all levels, which shows that the framework could not turn extra load into serving capacity beyond its parallel-slot limit.

5.5. Limitations

This study has some limitations to keep in mind. First, we used a single model family (Qwen3-4B) on one hardware setup with a single NVIDIA H100 80 GB GPU. Therefore, the results may not transfer directly to other model architectures, parameter sizes, quantization formats, or hardware. In particular, the FP8 native path for vLLM is available only on Hopper-class and newer GPUs. On older hardware, vLLM would fall back to GPTQ or FP16 execution, which may narrow the gap. Ollama's relative performance may also differ on hardware where its GGUF path is more competitive.

Second, we drew the scenarios from five open Hugging Face datasets (TriviaQA, GSM8K, HumanEval, HotpotQA, MT-Bench) under controlled concurrency, not from fully mixed real-world traffic. The scenarios cover question answering, multi-step reasoning, streaming dialog, and stress workloads, but real systems may have more complex request mixes, more workload variety, and multi-tenant behavior than we captured.

Third, several quality metrics (reasoning accuracy, response coherence, response quality, and context retention) use lightweight heuristic methods, rather than human annotation, embedding-based metrics such as BERTScore [44], or LLM-as-judge evaluation. The 0.00 accuracy for Ollama in Scenario 2 comes from timeout-induced truncation and heuristic-scoring limits, not from purely wrong answers. Therefore, this metric should be read as a proxy for the quality of completed output under load, not as an exact quality measure.

Fourth, the measurement harness does not separately instrument server-side queue waiting time and pure inference time. All latency and TTFT values are combined client-observed measurements that include network overhead, queuing, prefill, and decode. This keeps the two frameworks consistent, but it limits how finely we can attribute latency. Finer internal traces would be needed to fully split queuing from inference under overload.

Finally, we focused on single-node, single-GPU deployments. We did not study distributed inference, multi-node scaling, energy efficiency, or cost–performance trade-offs. We reported the endurance memory growth as cumulative resource-monitor readings, not steady-state leak rates, and a longer multi-day test would be needed to confirm long-term stability. Future work should extend the study to broader settings, larger models, and longer endurance windows to check how well these findings generalize.

6. Conclusions and Future Work

This paper presents a systematic benchmark of Ollama and vLLM across four scenarios: baseline performance, complex reasoning, streaming, and stress testing. We construct the comparison on a single NVIDIA H100 80 GB GPU using the same Qwen3-4B model weights, the same sampling parameters, and deterministic subsamples from five open benchmark datasets (TriviaQA, GSM8K, HumanEval, HotpotQA, MT-Bench). This setup ensures that the only differences come from the internal design of the two frameworks.

Our experiments demonstrate that vLLM delivers higher throughput (20–29 times higher across scenarios), much lower latency (8–19 times lower P95 latency), and near-perfect reliability (a 100% success rate in all scenarios) under concurrent load. We demonstrate that it remains stable up to about 100 users before its degradation zone at 200 users, while still completing all requests. Its FP8 native execution on Hopper hardware, combined with continuous batching and PagedAttention-based KV-cache management, yields a decode throughput of over 1800 tokens per second in baseline and over 3200 tokens per second in reasoning.

https://doi.org/10.3390/app16115435

Ollama, by contrast, reaches a structural concurrency limit at about 10 users on the test hardware, with throughput flat at 0.45 req/s regardless of concurrency. Beyond this limit, queue saturation causes request timeouts that range from 13% in baseline to over 30% in streaming. We find that the heuristic quality scores on completed Ollama requests (coherence 0.74–0.76, response quality 0.63) are close to those of vLLM, which indicates that the gap comes mainly from the serving architecture rather than from generation quality.

During endurance testing, both frameworks exhibit notable resource growth over the 3600 s window (about 5.6 GB for Ollama and about 14.2 GB for vLLM), although vLLM processes roughly 33 times more requests in the same window. Per request, vLLM's memory footprint is therefore much lower, but the absolute growth should still be monitored in long-running use.

Beyond the head-to-head comparison, this work contributes a unified four-scenario pipeline (baseline, reasoning, streaming, and stress testing) and an open benchmarking framework that lets others reproduce the same comparison with other models, frameworks, and hardware. The pipeline provides deterministic dataset pinning through Hugging Face commit SHAs, structured environment logging, unified streaming HTTP measurement, and modular post-processing for paper-ready tables and charts. With this work, we highlight that vLLM is the more robust choice for scalable LLM inference under concurrent load on Hopper-class hardware, while Ollama remains useful for lightweight, local, or developer deployments.

There is still room for future work in this direction. In particular, the comparison could be extended to larger models and multi-node deployments, and the memory growth observed during endurance testing could be studied over longer runs to confirm long-term stability. Another useful direction is to add energy and cost-performance analysis. In this work, we measured only performance and resource usage; future studies could additionally report metrics such as tokens per watt, energy use per request, and cost per request, thereby providing a more complete basis for deployment decisions. Finally, the workloads here were controlled and separated into distinct scenarios. Real systems often serve mixed and bursty traffic with many request types at once, as well as multi-tenant usage. Future work could therefore test both frameworks under mixed, realistic workloads that better reflect production traffic.

Author Contributions: Conceptualization, B.A. and Y.E.D.; methodology, B.A.; software, B.A. and Y.E.D.; validation, B.A.; formal analysis, B.A. and Y.E.D.; investigation, Y.E.D.; resources, B.A.; data curation, B.A.; writing—original draft preparation, B.A.; writing—review and editing, B.A. and Y.E.D.; visualization, Y.E.D.; supervision, B.A.; project administration, B.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All experimental scripts, prompt sets, Docker configurations, dataset revision pins, and measurement tools used in this study are publicly available on GitHub at https://github.com/demirdagyunus/serving-compare-tests. The aggregated results are reported within the article. Further inquiries can be directed to the corresponding author.

Acknowledgments: During the preparation of this manuscript, the authors used Claude Opus 4.7 (Anthropic) for language editing, proofreading and graphical abstract. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

https://doi.org/10.3390/app16115435

References

1. Pan, J.; Li, G. A survey of LLM inference systems. *arXiv* **2025**, arXiv:2506.21901.
2. Zhou, Z.; Ning, X.; Hong, K.; Fu, T.; Xu, J.; Li, S.; Lou, Y.; Wang, L.; Yuan, Z.; Li, X.; et al. A survey on efficient inference for large language models. *arXiv* **2024**, arXiv:2404.14294.
3. Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C.H.; Gonzalez, J.; Zhang, H.; Stoica, I. Efficient memory management for large language model serving with PagedAttention. In Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23), Koblenz, Germany, 23–26 October 2023; pp. 611–626.
4. Hu, E.J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; Chen, W. LoRA: Low-rank adaptation of large language models. In Proceedings of the International Conference on Learning Representations (ICLR), Virtual, 25–29 April 2022.
5. Xu, X.; Li, M.; Tao, C.; Shen, T.; Cheng, R.; Li, J.; Xu, C.; Tao, D.; Zhou, T. A survey on knowledge distillation of large language models. *arXiv* **2024**, arXiv:2402.13116.
6. Dettmers, T.; Holtzman, A.; Pagnoni, A.; Zettlemoyer, L. QLoRA: Efficient finetuning of quantized LLMs. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 10088–10115.
7. Zhong, Y.; Liu, S.; Chen, J.; Hu, J.; Zhu, Y.; Liu, X.; Jin, X.; Zhang, H. 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), Santa Clara, CA, USA, 10–12 July 2024.
8. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. Transformers: State-of-the-art natural language processing. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP), Online, 16–20 November 2020; pp. 38–45.
9. Aminabadi, R.Y.; Rajbhandari, S.; Awan, A.A.; Li, C.; Li, D.; Zheng, E.; Ruwase, O.; Smith, S.; Zhang, M.; Rasley, J.; et al. DeepSpeed-Inference: Enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, 13–18 November 2022*; IEEE: Piscataway, NJ, USA, 2022.
10. Zhou, Y.; Yang, K. Exploring TensorRT to improve real-time inference for deep learning. In *Proceedings of the 2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*; IEEE: Piscataway, NJ, USA, 2022.
11. Ollama Contributors. Ollama: Local Large Language Models. Available online: https://github.com/ollama (accessed on 10 August 2025).
12. Li, B.; Jiang, Y.; Gadepally, V.; Tiwari, D. LLM inference serving: Survey of recent advances and opportunities. In *Proceedings of the 2024 IEEE High Performance Extreme Computing Conference (HPEC)*; IEEE: Piscataway, NJ, USA, 2024.
13. Oh, H.; Kim, K.; Kim, J.; Kim, S.; Lee, J.; Chang, D.-S.; Seo, J. ExeGPT: Constraint-aware resource scheduling for LLM inference. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24), La Jolla, CA, USA, 27 April–1 May 2024; Volume 2.
14. Dao, T.; Ermon, S.; Fu, D.; Ré, C.; Rudra, A. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 16344–16359.
15. Sheng, Y.; Zheng, L.; Yuan, B.; Li, Z.; Ryabinin, M.; Chen, B.; Liang, P.; Re, C.; Stoica, I.; Zhang, C. FlexGen: High-throughput generative inference of large language models with a single GPU. In Proceedings of the 40th International Conference on Machine Learning (ICML), Honolulu, HI, USA, 23–29 July 2023.
16. Lang, J.; Guo, Z.; Huang, S. A comprehensive study on quantization techniques for large language models. In *Proceedings of the 2024 4th International Conference on Artificial Intelligence, Robotics, and Communication (ICAIRC)*; IEEE: Piscataway, NJ, USA, 2024.
17. Kolluru, S. Comparative analysis of large language model inference serving systems: A performance study of vLLM and HuggingFace TGI. *arXiv* **2025**, arXiv:2511.17593.
18. Chitty-Venkata, K.T.; Raskar, S.; Kale, B.; Ferdaus, F.; Tanikanti, A.; Raffenetti, K.; Taylor, V.; Emani, M.; Vishwanath, V. LLM-inference-bench: Inference benchmarking of large language models on AI accelerators. In *Proceedings of SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 17–22 November 2024*; IEEE: Piscataway, NJ, USA, 2024.
19. Yu, G.I.; Jeong, J.S.; Kim, G.W.; Kim, S.; Chun, B.G. 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), Carlsbad, CA, USA, 11–13 July 2022.

20. Agrawal, A.; Kedia, N.; Panwar, A.; Mohan, J.; Kwatra, N.; Gulavani, B.; Tumanov, A.; Ramjee, R. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24), Santa Clara, CA, USA, 10–12 July 2024.
21. Patel, P.; Choukse, E.; Zhang, C.; Shah, A.; Goiri, Í.; Maleki, S.; Bianchini, R. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), Buenos Aires, Argentina, 29 June–3 July 2024*; IEEE: Piscataway, NJ, USA, 2024.
22. Zheng, L.; Yin, L.; Xie, Z.; Sun, C.; Huang, J.; Yu, C.H.; Cao, S.; Kozyrakis, C.; Stoica, I.; Gonzalez, J.E.; et al. SGLang: Efficient execution of structured language model programs. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 62557–62583.
23. Cai, T.; Li, Y.; Geng, Z.; Peng, H.; Lee, J.D.; Chen, D.; Dao, T. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. *arXiv* **2024**, arXiv:2401.10774.
24. Li, Y.; Wei, F.; Zhang, C.; Zhang, H. EAGLE: Speculative sampling requires rethinking feature uncertainty. *arXiv* **2024**, arXiv:2401.15077.
25. Husom, E.J.; Goknil, A.; Astekin, M.; Shar, L.K.; Kã¥Sen, A.; Sen, S.; Mithassel, B.A.; Soylu, A. Sustainable LLM inference for edge AI: Evaluating quantized LLMs for energy efficiency, output accuracy, and inference latency. *ACM Trans. Internet Things* **2025**, *6*, 1–35.
26. Laskaridis, S.; Katevas, K.; Minto, L.; Haddadi, H. Melting point: Mobile evaluation of language transformers. In Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24), Washington, DC, USA, 18–22 November 2024.
27. Stuhlmann, L.; Fadel Argerich, M.; Fürst, J. Bench360: Benchmarking local LLM inference from 360°. *arXiv* **2025**, arXiv:2511.16682.
28. Rajesh, V.; Jodhpurkar, O.; Anbuselvan, P.; Singh, M.; Jallepali, A.; Godbole, S.; Sharma, P.K.; Shrivastava, H. Production-grade local LLM inference on Apple Silicon: A comparative study of MLX, MLC-LLM, Ollama, llama.cpp, and PyTorch MPS. *arXiv* **2025**, arXiv:2511.05502.
29. Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; et al. Qwen3 technical report. *arXiv* **2025**, arXiv:2505.09388.
30. ggml-org. llama.cpp: LLM Inference in C/C++. Available online: https://github.com/ggml-org/llama.cpp (accessed on 10 August 2025).
31. Joshi, M.; Choi, E.; Weld, D.S.; Zettlemoyer, L. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Vancouver, BC, Canada, 30 July–4 August 2017; pp. 1601–1611.
32. Cobbe, K.; Kosaraju, V.; Bavarian, M.; Chen, M.; Jun, H.; Kaiser, L.; Plappert, M.; Tworek, J.; Hilton, J.; Nakano, R.; et al. Training verifiers to solve math word problems. *arXiv* **2021**, arXiv:2110.14168.
33. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374.
34. Yang, Z.; Qi, P.; Zhang, S.; Bengio, Y.; Cohen, W.; Salakhutdinov, R.; Manning, C.D. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP), Brussels, Belgium, 31 October–4 November 2018; pp. 2369–2380.
35. Zheng, L.; Chiang, W.L.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E.; et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 46595–46623.
36. Dean, J.; Barroso, L.A. The tail at scale. *Commun. ACM* **2013**, *56*, 74–80.
37. Yan, Y.; Su, J.; He, J.; Fu, F.; Zheng, X.; Lyu, Y.; Wang, K.; Wang, S.; Wen, Q.; Hu, X. A survey of mathematical reasoning in the era of multimodal large language model: Benchmark, method & challenges. In Proceedings of the Findings of the Association for Computational Linguistics: ACL 2025, Vienna, Austria, 27 July–1 August 2025.
38. Jiang, J.; Wang, F.; Shen, J.; Kim, S.; Kim, S. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 1–72.
39. Mavi, V.; Jangra, A.; Jatowt, A. Multi-hop question answering. Found. *Trends Inf. Retr.* **2024**, *17*, 457–586.
40. Yi, Z.; Ouyang, J.; Xu, Z.; Liu, Y.; Liao, T.; Luo, H.; Shen, Y. A survey on recent advances in LLM-based multi-turn dialogue systems. *ACM Comput. Surv.* **2025**, *58*, 1–38.
41. NVIDIA. Llama-Nemotron: Efficient reasoning models. *arXiv* **2025**, arXiv:2505.00949.

42. Sun, B.; Huang, Z.; Zhao, H.; Xiao, W.; Zhang, X.; Li, Y.; Lin, W. Llumnix: Dynamic scheduling for large language model serving. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24), Santa Clara, CA, USA, 10–12 July 2024.
43. Agrawal, A.; Kedia, N.; Agarwal, A.; Mohan, J.; Kwatra, N.; Kundu, S.; Ramjee, R.; Tumanov, A. On evaluating performance of LLM inference serving systems. *arXiv* **2025**, arXiv:2507.09019.
44. Zhang, T.; Kishore, V.; Wu, F.; Weinberger, K.Q.; Artzi, Y. BERTScore: Evaluating text generation with BERT. *arXiv* **2019**, arXiv:1904.09675.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.