GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Bachelor's Thesis

submitted in partial fulfillment of the
requirements for the course "Applied Computer Science"

Performance Evaluation of LLM Inference Engines

Yining Liu
MatrNr: 1904700

First Supervisor: Prof.Dr.Julian Kunkel
Second Supervisor: Dr. Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science
ISSN: 1612-6793

August 18, 2026

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

Abstract

In recent years, Large Language Models have been increasingly deployed in production services, and the inference engine largely determines the latency, throughput, and hardware cost of these services. This thesis presents a performance evaluation of three widely used open-source inference engines: vLLM, SGLang, and llama.cpp. All experiments were conducted under controlled conditions on a Slurm-managed HPC cluster using an NVIDIA A100 GPU, with the three engines deployed in mutually isolated Apptainer containers. The experiments evaluated six models of different sizes and architectures under three workloads of different scales and three concurrency levels, measuring TTFT, ITL, throughput, and peak GPU memory, complemented by stress tests without a concurrency limit. The experimental results show that vLLM achieves the lowest TTFT in the vast majority of medium- and high-concurrency configurations. SGLang delivers the highest throughput under long-input, high-load conditions. llama.cpp, benefiting from 4-bit quantization, generally performs best under single-request small and medium workloads and uses substantially less GPU memory, but falls clearly behind vLLM and SGLang under high-concurrency conditions. Concurrency, prompt length, and model architecture jointly affect the performance of inference engines, and therefore no single engine performs optimally under all conditions. This thesis summarizes these experimental findings into selection recommendations for practical deployment scenarios, providing a reference for developers and researchers in choosing an appropriate inference engine.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☑ To write individual passages, altogether to the extent of 5-10% of the entire text
- ☐ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Acknowledgements

I would like to express my gratitude to my supervisors, Professor Julian Kunkel and Dr. Jonathan Decker, for their guidance and valuable insights regarding this thesis.

I also wish to thank Lauritz Rasbach for his guidance, patience, and support throughout the research process. He remained patient during our weekly meetings and encouraged me to persevere in completing this thesis. The feedback and suggestions he provided during the experiments and the writing phase were immensely helpful.

Furthermore, I would like to thank GWDG for providing access to the HPC cluster required for this research.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

API Application Programming Interface

BF16 Brain Floating Point 16-bit

CUDA Compute Unified Device Architecture

GGUF GPT-Generated Unified Format

GPU Graphics Processing Unit

HPC High-Performance Computing

ITL Inter-Token Latency

KV Key-Value (as in KV cache)

LLM Large Language Model

MoE Mixture of Experts

RQ Research Question

Slurm Simple Linux Utility for Resource Management

TTFT Time to First Token

1 Introduction

1.1 Motivation

In recent years, Large Language Models (LLMs) have developed rapidly and have been widely applied in scenarios such as chatbots, code assistants, document analysis, and intelligent search[Zha+23]. With the popularization of applications such as ChatGPT, LLMs are moving from research prototypes to large-scale production deployment, and the number of users and the volume of requests served continue to grow. For a long time, research has focused on model capabilities themselves, such as larger parameter sizes and stronger reasoning and generation abilities.In practical applications, however, the cost of deploying these models is increasingly determined by the inference stage[Li+24]: unlike training, which is performed only once, inference must run continuously and serve a large number of users with acceptable latency on expensive GPU hardware. Inference efficiency directly determines how many users the same hardware can support, the response speed of each request, and ultimately the cost of the service.

LLM inference itself poses a series of unique system challenges. Mainstream models generate output token by token in an autoregressive manner, and the inference process is divided into a prefill stage that processes the input and a decode stage that generates tokens step by step, with fundamentally different computational characteristics[Zho+24]. In addition, the generation process must maintain a KV cache in GPU memory that grows with the sequence length, making memory management a key factor affecting concurrency capacity[Kwo+23]. The inference engine is the system responsible for loading the model, scheduling requests, managing the KV cache, and generating output. It was created precisely to address these challenges and plays a decisive role in the practical performance and cost of LLM services: the same model may perform several times or even an order of magnitude differently on different engines[Wen+25].

A number of open-source inference engines with fundamentally different design goals have emerged. vLLM [Kwo+23]borrows the paging idea from operating systems and improves GPU memory utilization through PagedAttention to support higher concurrent throughput. SGLang[Zhe+23] reuses the KV cache of shared prefixes through RadixAttention and adopts a throughput-oriented runtime design. llama.cpp[ggm23c] takes a different path, targeting consumer-grade hardware and achieving lightweight local deployment through GGUF quantized models. The differences among these engines in scheduling strategies, memory management, and execution make them likely to exhibit completely different performance characteristics under different load conditions.

For practitioners deploying LLMs, this diversity raises a practical question: which engine should be chosen, and under which conditions should it be used? Existing evaluation studies cover only a limited range of engines, model sizes, prompt lengths, or load levels, and often overlook important factors such as concurrency scaling, long inputs, and Mixture-of-Experts (MoE) architectures. In addition, performance reports from the engines themselves are usually based on scenarios favorable to them and are difficult to compare directly. Therefore, a systematic comparison of mainstream inference engines under unified hardware, unified workloads, and controlled variables has both research value and direct practical significance. This is precisely the starting point of this thesis.

1.2 Contribution

This thesis contributes a controlled performance evaluation of vLLM, SGLang, and llama.cpp on production-grade HPC hardware (NVIDIA A100). The experiments cover six models, including an MoE architecture, three prompt-length workloads, multiple concurrency levels, and additional stress tests, providing a broader experimental scope than previous evaluations. Based on these experiments, the three engines are compared quantitatively in terms of TTFT, ITL, throughput, and peak GPU memory, and the observed differences are attributed to their respective design choices. Finally, the research findings are distilled into scenario-oriented recommendations for selecting the most suitable inference engine for typical deployment scenarios.

1.3 Outline

The remainder of this thesis is organized as follows. Section 2 introduces the background on LLM inference, the three inference engines, and the performance metrics. Section 3 reviews related work on inference engine optimization and benchmarking. Section 4 describes the research questions, experimental setup, and benchmark implementation. Section 5 presents and analyzes the evaluation results. Section 6 discusses the challenges encountered and interprets the findings. Section 7 concludes the thesis.

2 Background

2.1 Overview of Large Language Models

Most current Large Language Models are based on the Transformer architecture, whose core mechanism is the self-attention mechanism[Vas+17]. Unlike traditional recurrent neural networks, the Transformer does not need to process the input sequence step by step according to time steps, but can process the entire input sequence in parallel at once. With the help of Self-Attention, each token attends to all other tokens in the sequence, thereby effectively capturing the global dependencies in the input sequence.

In Self-Attention, the input representation of each token is transformed through three different linear projections to obtain the Query (Q), Key (K), and Value (V):

$$Q = XW^Q, \qquad K = XW^K, \qquad V = XW^V. \tag{1}$$

Where X represents the input token representation, and W^Q, W^K, and W^V are weight matrices learned during model training. The computation of Self-Attention can then be expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{2}$$

Where d_k denotes the dimension of the Key vectors. The Query can be understood as the information that the current token is looking for, the Key is used to match with the Query and determine the relevance of different tokens, while the Value contains the actual information to be aggregated. By computing the similarity between the Query and all Keys, the corresponding attention weights can be obtained. These weights are then used to form a weighted combination of the Values, thereby generating the contextual representation of the current token.It is also important to note that the self-attention operation involves matrix multiplication, and its computational complexity grows quadratically with the length of the input sequence[Zho+24].

It is worth noting that the K and V of each token are determined only by the representation of that token and do not change once they have been computed. Therefore, during autoregressive inference, the K and V tensors of previously processed tokens can be cached and directly reused in subsequent generation steps without being recomputed[Zho+24].

While the original Transformer follows an encoder–decoder design, most modern LLMs, including all models evaluated in this thesis, adopt a decoder-only architecture consisting of stacked Transformer blocks[Zha+26].

As shown in Figure 1, a Transformer model usually consists of multiple stacked Transformer Blocks. Before entering the Transformer Block, the Transformer model first converts the input text into a series of tokens through Tokenization, maps each token into a vector representation through the Embedding Layer, and adds position embeddings to give the model the ability to perceive the order of the sequence. After entering the Transformer Block, the data passes sequentially through Multi-Head Self-Attention, the Feed Forward Network, and Layer Normalization. The output features of a module serve as the input of the next module. Through this layer-by-layer stacked structure, the model can further extract contextual features and, combined with the already generated content, finally predict the output text step by step.

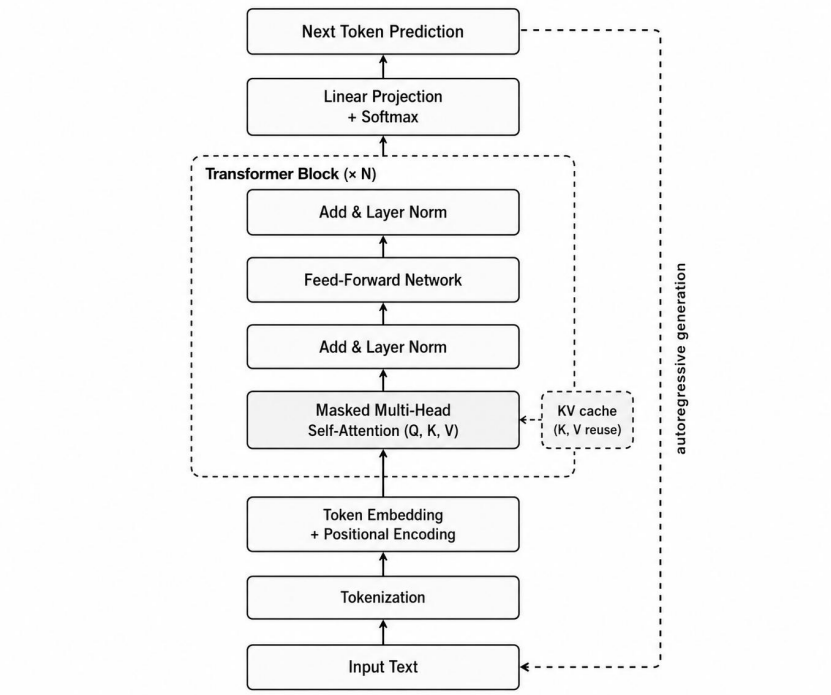

Figure 1: Simplified causal decoder-only Transformer architecture for autoregressive language generation.

2.2 LLM Inference Process

Most mainstream LLMs usually adopt an autoregressive method to generate output tokens[Zho+24]. When the model predicts the next token, it must rely on all known historical tokens. The newly generated token is added to the end of the sequence, and the whole process continues repeatedly until the model outputs a stop marker, such as the `<EOS>` token [Li+24].

The inference process of LLMs can be divided into two stages: the prefilling stage and the decoding stage [Zho+24] [Li+24]:

- **Prefilling Stage:** The LLM processes all tokens in the input prompt in parallel and calculates their corresponding key and value representations. These representations are stored in the KV cache. At the end of the prefill stage, the first token of the output sequence is generated. Since this stage needs to process the entire input sequence at once and perform large-scale matrix multiplications, it is typically compute-bound, meaning that it is mainly limited by the computing power of the GPU and its cost grows with the prompt length.

- **Decoding Stage:** The decoding stage starts after the prefilling stage is completed. In this stage, the model generates output tokens one by one in an autoregressive manner.Since only one newly generated token is processed per step, the available parallelism is limited, and the decoding stage is therefore typically memory-bound. At each generation step, the model must read the model weights from GPU memory and access the previously accumulated KV cache, so its performance is limited more by memory bandwidth than by GPU computing power. In addition, the size of the KV cache grows with the sequence length and the number of concurrent requests, making GPU memory capacity a key constraint on concurrent serving capacity.

Prefilling and Decoding have different performance bottlenecks and therefore place different optimization requirements on inference engines. The prefilling stage mainly benefits from efficient parallel computation, while the decoding stage relies more heavily on efficient memory management and effective utilization of GPU memory bandwidth. The next section further introduces the main optimization methods adopted by the inference engines.

The KV cache stores the key (K) and value (V) tensors of historical tokens in GPU memory. When the LLM autoregressively generates a new token, it only needs to calculate the K and V tensors of the newly generated token and append them to the cache. In the subsequent attention computation, the model can directly reuse the historical K and V tensors. This avoids repeated computation of the historical context and significantly improves the inference efficiency of LLMs.

Without the KV cache, during the autoregressive generation stage, every time a new token is generated, the K and V tensors of this token and all previous historical tokens would need to be recalculated. This causes the total amount of computation to grow quadratically, which leads to high computational cost when processing long texts and makes inference inefficient for long-context scenarios.

2.3 LLM Inference Engines

An LLM itself is only a trained model parameter file, while an inference engine applies the trained model to practical scenarios. The inference engine is responsible for loading the model, receiving input prompts, executing the generation process, and returning the generated text to the user. In addition, it also manages key parts of the inference process, such as managing concurrent requests, receiving the number of prompts, controlling the output length, managing the KV cache, and efficiently utilizing GPU memory. Choosing a proper inference engine can reduce request latency, improve system throughput, and enhance the utilization of hardware resources such as GPUs.

GitHub statistics, such as the total number of stars and the growth trend of stars over time, are used to measure the popularity and influence of Large Language Model (LLM) inference engines in the open-source community [Par+26]. Based on these statistics, this thesis selects three widely used open-source inference engines with high community attention as the research objects: vLLM [Kwo+23], SGLang[Zhe+23], and llama.cpp [ggm23c].

2.3.1 vLLM

vLLM is a high-performance, open-source inference engine designed for large language models (LLMs). Its core innovation lies in the introduction of PagedAttention, a concept inspired by the virtual memory and paging mechanisms of operating systems, which is applied to the KV cache during the inference process. vLLM first allocates a block of memory and divides it into multiple fixed-size physical blocks, each of which stores key and value vectors corresponding to a fixed number of tokens. These blocks are initially empty. When a new inference request arrives, vLLM allocates several logical blocks. Logically, multiple sets of blocks are contiguous in virtual space. However, in terms of actual allocation, vLLM maintains a block table and dynamically maps the generated KV cache to the pre-allocated physical blocks in a non-contiguous manner. In this way, vLLM significantly reduces storage fragmentation, thereby improving overall throughput.

Feature	Traditional KV Cache	PagedAttention
GPU memory allocation	Static and contiguous	Dynamic and non-contiguous
Memory waste	Up to 60–80% due to fragmentation and over-reservation	Near-zero waste, reported as less than 4%

Table 1: Comparison between traditional KV cache management and PagedAttention, based on Kwon et al. [Kwo+23]

Combined with the two-stage characteristics described in Section 2.2, PagedAttention mainly optimizes KV cache memory management during the decoding stage. By reducing GPU memory fragmentation of the KV cache, the same amount of GPU memory can accommodate the KV caches of more concurrent requests, thereby improving the system's concurrent processing capability and overall throughput.

2.3.2 SGLang

SGLang consists of a domain-specific programming frontend and a high-performance backend runtime [Zhe+23].

1. **Frontend Language:**

   SGLang provides a domain-specific language embedded in Python [Zhe+23]. Its goal is to simplify the development of LLM applications by providing primitives for prompt construction, generation, and parallel execution. These primitives can be combined with native Python control flow and libraries, making it easier to develop complex prompting workflows. For example, `gen` is used to generate and store model outputs, `select` allows the model to choose from a predefined set of options, and `fork` and `join` are used to create and merge parallel branches of the prompt state.

2. **Backend Runtime:**

   The backend runtime mainly addresses the problem of inference acceleration. Its core contribution is the use of the RadixAttention mechanism. At runtime, RadixAttention automatically reuses the KV cache of shared prefixes, thereby reducing repeated prefilling computation and GPU memory usage. This significantly reduces Time to First Token (TTFT) and improves throughput. In scenarios with a large number of identical prefixes, such as multi-turn conversations, few-shot prompting, self-consistency, and tree-of-thought, SGLang has very obvious performance advantages.

The working principle of RadixAttention is to use a radix tree to maintain the historical KV cache. SGLang organizes all previously computed token sequences, including prompts and generated results, into a radix tree. When the frontend sends the complete prompt to the backend, the runtime performs longest-prefix matching in the radix tree. The matched prefix directly reuses its existing KV cache, while the newly added remaining tokens require prefilling computation. The new incremental sequence, including the prefix, new tokens, and generated output, is then inserted back into the tree.

In this way, redundant computation is avoided, Time to First Token (TTFT) is significantly reduced, and the overall system throughput is improved.

2.3.3 llama.cpp

llama.cpp is an open-source C/C++ inference engine for Large Language Models (LLMs). It was originally designed to run LLaMA-like models efficiently on consumer-grade hardware with relatively low memory usage. Since then, it has been extended to support many popular models, such as Qwen and Mistral, and even non-Transformer architectures such as Mamba [Spa+25]. Compared with server-oriented inference engines such as vLLM and SGLang, llama.cpp focuses more on lightweight deployment and local inference.

The implementation of llama.cpp is based on the GGML tensor computation library [ggm23a] and the GGUF model file format [ggm23b]. GGML provides support for low-level tensor operations, memory management, and quantized computation. GGUF is a binary model file format used to store model weights together with important metadata required for inference, such as model architecture information, tokenizer data, tensor names, tensor shapes, and tensor types[Boh+26].

One of the core features of llama.cpp is its support for quantized models. By converting model weights into lower-precision representations, llama.cpp can significantly reduce model file size and runtime memory usage. This makes it possible to run larger models on standard CPUs or consumer-grade GPUs.

In addition to quantization, llama.cpp also supports partial offloading between CPU and GPU. Using the `-ngl` option, the number of model layers placed on the GPU can be configured, while the remaining model layers are kept on the CPU. In addition, the KV cache can also be kept in CPU memory. In the experiments of this thesis, all model layers were placed on the GPU by setting `-ngl 999`, so that the evaluation of llama.cpp more closely matches the same pure-GPU execution conditions as vLLM and SGLang.

Combined with the two-stage characteristics described in Section 2.2, the performance benefits of quantization are mainly reflected in the decoding stage. Since the decoding stage is more likely to be limited by GPU memory bandwidth, reducing the size of the model weights can reduce the amount of data that needs to be read from GPU memory when generating each token. In addition, quantization can also significantly reduce the GPU memory required to store the model weights, which is also an important reason why llama.cpp is able to run larger models on hardware with limited GPU memory.

2.4 Performance Metrics

In the following experiments, several performance metrics need to be introduced:

1. **Time to First Token (TTFT):**

   Time to First Token (TTFT) refers to the time required to generate the first output token after receiving the prompt. Simply speaking, it measures how quickly the user can see the output of the LLM after asking a question, and it determines the "response speed". A high TTFT may make users feel that the system is slow or unresponsive. Therefore, TTFT is one of the most important metrics for evaluating the interactive performance of large language model applications.

2. **Inter Token Latency (ITL):**

   Inter Token Latency (ITL) refers to the average time interval between consecutive generated tokens. It measures the speed of generating the next token after one token has already been generated.

$$\text{ITL} = \frac{\text{End-to-End Latency} - \text{TTFT}}{\text{Batch Size} \times (\text{Output Tokens} - 1)} \tag{3}$$

3. **Throughput:**

   Throughput refers to the number of tasks processed per unit of time, namely the number of tokens processed per second. This metric is mainly used to evaluate the overall utilization efficiency of GPU computing power and memory bandwidth by the inference engine. In the following experiments, total throughput and output throughput are used, which correspond to the following formulas:

$$\text{Total Throughput} = \frac{\text{Batch Size} \times (\text{Input Tokens} + \text{Output Tokens})}{\text{End-to-End Latency}} \tag{4}$$

$$\text{Output Throughput} = \frac{\text{Batch Size} \times \text{Output Tokens}}{\text{End-to-End Latency}} \tag{5}$$

End-to-End Latency measures the total response time of a request, from the moment the prompt is sent to the inference engine until the final generated token is returned.

2.5 HPC and Slurm

High Performance Computing (HPC) refers to a large cluster of nodes connected through a high-speed interconnection network. These nodes generally include CPUs, GPUs, memory, local storage, and other computing resources. The core goal of HPC is to complete large-scale, complex, or data-intensive tasks that are difficult for traditional computing devices to handle within a shorter period of time. At the same time, HPC improves system throughput, reduces task execution time, and enables efficient scheduling and management of computing resources [LKG25].

Compared with traditional computing devices, HPC supports parallel processing both within nodes and across nodes. By distributing tasks to multiple nodes for collaborative execution, HPC integrates the computing resources of multiple machines to solve complex problems in fields such as finance, healthcare, artificial intelligence, and scientific computing [Int26].

Simple Linux Utility for Resource Management (Slurm)[YJG03] is a workload and resource manager specifically designed for HPC clusters . It is mainly responsible for job scheduling and resource allocation in a cluster. Users can submit batch jobs through shell scripts. Slurm allocates tasks according to the status of the nodes and the requirements of the jobs in order to avoid interference between jobs and ensure efficient resource utilization.

3 Related Work

3.1 LLM Inference Engine Optimization

In recent years, large language models have developed rapidly, and the research focus is no longer limited to the models themselves. How to provide efficient inference services for them has become an equally important issue, leading to the emergence of various inference engines with different design goals: some are designed for high-concurrency serving in data centers, while others focus on lightweight deployment on consumer-grade hardware. This thesis focuses on three representative and widely used open-source LLM inference engines: vLLM, SGLang, and llama.cpp.

Kwon et al. proposed vLLM[Kwo+23]. Through PagedAttention, it applies the paging idea from operating systems to KV cache management, allowing logically continuous key and value tensors to be stored in physically non-contiguous GPU memory blocks. This method addresses the memory fragmentation problem in traditional LLM inference and significantly improves GPU memory utilization. As a result, more requests can be accommodated simultaneously, and throughput can be increased. Due to its efficient KV cache management mechanism and broad model support, vLLM is often used in existing studies as a representative high-performance baseline system for evaluating LLM inference performance.

SGLang[Zhe+23] focuses more on the efficient execution of structured language model programs and achieves shared-prefix reuse through RadixAttention. RadixAttention uses a radix tree to manage previously computed token sequences. For a new request, it searches for the longest matching prefix in the tree and directly reuses it. Only the remaining new tokens require prefill computation, and the newly computed sequence is then stored in the tree. By reducing repeated prefix computation, SGLang can reduce TTFT and improve system throughput.

Unlike vLLM and SGLang, which mainly target high-throughput server-side inference, llama.cpp[ggm23c] places greater emphasis on lightweight model deployment. In the field of LLM optimization, low-bit quantization is a core technique. Previous research on quantization methods such as GPTQ[Fra+22], SmoothQuant[Xia+23], and AWQ[Lin+24] has shown that LLMs can reduce model size through low-precision representations while maintaining accuracy as much as possible, thereby reducing memory usage and bandwidth bottlenecks[Mal25]. Quantization is applicable not only to CPU inference but also to GPU inference, because a smaller model size can reduce GPU memory usage and memory bandwidth requirements. llama.cpp supports multiple quantized GGUF models and can execute them through both CPU and GPU backends. In this thesis, llama.cpp uses GPU acceleration in order to compare the performance differences between a lightweight quantized inference engine and the vLLM and SGLang inference engines on the same GPU hardware.

3.2 Benchmarking and Performance Evaluation of LLM Inference Engines

Zhou et al. provided a detailed explanation of autoregressive decoding, attention computation, model size, KV cache, and quantization in *A Survey on Efficient Inference for Large Language Models*, providing solid theoretical support for this thesis [Zho+24].

Chitty-Venkata et al. proposed LLM-Inference-Bench to evaluate the inference performance of Large Language Models on different AI accelerators. This work provides a relatively systematic benchmarking method for evaluating LLM inference performance and explains that throughput, latency, and scalability under different hardware configurations are important aspects that need to be considered when evaluating inference systems. This thesis refers to its evaluation approach and adopts metrics such as TTFT, ITL, and throughput to evaluate different inference engines [Chi+24].

de Lima Luiz et al. studied the performance of different LLM models in a Slurm-based HPC architecture, demonstrating that HPC and Slurm can provide resource allocation, job scheduling, and scalable computing capabilities for LLM inference tasks [LKG25].

In addition to optimization research on individual inference engines, previous work has also begun to evaluate the performance of different Large Language Model inference engines. The work most closely related to this thesis is *An Evaluation of the State-of-the-Art LLM Serving Frameworks* by Weng et al. This study compared four open-source inference engines, namely vLLM, SGLang, Text Generation Inference, and CTranslate2, in a Google Colab environment equipped with an NVIDIA T4 GPU. It used two models with different parameter sizes, Llama-3.2-1B-Instruct and Gemma-2-2B-IT, to test latency, throughput, and resource usage. The results show that, even when the same model is used, differences among inference engines in scheduling strategies, memory management, and computational implementation can still lead to significantly different inference performance [Wen+25].

In comparison, this thesis compares vLLM, SGLang, and llama.cpp in a Slurm-managed HPC environment using an NVIDIA A100 GPU. It uses multiple models of different sizes, datasets with different prompt lengths, and multiple concurrency levels. In addition to latency and throughput, this thesis further analyzes TTFT, ITL, and peak GPU memory, thereby providing a more comprehensive investigation of how model size, input length, and the number of concurrent requests affect the performance of different inference engines.

4 Methodology

4.1 Research Questions

This thesis aims to evaluate and compare the performance of three currently popular open-source large language model inference engines: vLLM, SGLang, and llama.cpp. Although all three systems can be used to perform LLM inference tasks, they adopt different optimization strategies.

The actual performance of inference engines is affected by factors such as model size, prompt length, output length, request concurrency, model precision, and hardware resources. In order to investigate the influence of different factors on performance, the following specific research objectives were established:

- Compare the latency and throughput of vLLM, SGLang, and llama.cpp, where latency includes TTFT and ITL.
- Investigate how model size affects the performance of inference engines.
- Evaluate the influence of different concurrency levels on the inference engines, as well as the influence of prompts with different lengths.
- Compare their peak GPU memory usage.
- Determine the workload conditions under which each inference engine performs best through multiple comparisons.

Based on the objectives above, this thesis investigates the following research questions:

RQ1: What differences exist among vLLM, SGLang, and llama.cpp in terms of latency, throughput, and peak GPU memory?

RQ2: How do model size, prompt length, and concurrency affect the relative performance of the three inference engines?

RQ3: Under which practical usage scenarios is each inference engine the best choice?

This thesis focuses on the performance of inference engines and does not involve model training or fine-tuning. Therefore, its main focus is the performance of the inference engines rather than accuracy. It should also be noted that llama.cpp uses quantized GGUF models, whereas vLLM and SGLang execute floating-point model weights. Therefore, this comparison reflects performance under practical deployment configurations rather than a comparison strictly based on the same precision. The observed performance differences may be influenced by both the implementation of the inference engines and the model representation formats.

4.2 Experimental Setup

To answer RQ1 and RQ2, this thesis designed a set of controlled comparative experiments and applied the controlled-variable method to quantitatively evaluate the performance of vLLM, SGLang, and llama.cpp. The experiments mainly varied four core factors, namely the inference engine, model parameter size, prompt length, and maximum concurrency, while keeping the other experimental conditions as consistent as possible, in order to analyze the effects of these factors on latency, throughput, and peak GPU memory. For RQ3, this thesis will synthesize the experimental results obtained under different models,

prompt lengths, and concurrency settings, analyze the respective strengths and limitations of the three inference engines, and determine the practical application scenarios for which they are more suitable.

The following subsections describe the hardware and software environment, the selected models, the workloads, and the concurrency settings used in these experiments.

4.2.1 Hardware and Software Environment

All experiments were conducted on the GWDG high-performance computing cluster managed by Slurm. To ensure the feasibility and reproducibility of the experimental results, the hardware resources, driver version, and containerized environment used in the experiments are described in detail.

In terms of resource scheduling, the experiments were uniformly managed through the Slurm job scheduling system. Each Slurm job requested exactly the same physical resources: one NVIDIA A100-SXM4 GPU with 80 GB of GPU memory.

In HPC environments, the diversity of software dependencies and the security restrictions of multi-user shared clusters often present challenges for experimental deployment. To avoid software dependency conflicts and version incompatibilities caused by complex deep learning frameworks, and to ensure the consistency of the experimental environment to the greatest possible extent, vLLM, SGLang, and llama.cpp were deployed and executed in separate Apptainer containers. Unlike Docker containers, which require root privileges and a background daemon, Apptainer can be executed on HPC clusters without administrator privileges and can be integrated with the Slurm job scheduling system. In addition, Apptainer supports access to the host GPU and shared storage, making it suitable for GPU-accelerated large language model inference tasks [Pas+24].

Configuration Category	**Evaluation Item**	**Parameter / Version Information**
Computing Platform	Infrastructure	GWDG High-Performance Computing (HPC) Cluster
Computing Platform	Job Scheduler	Slurm Workload Manager
Physical Resource Allocation	Graphics Processing Unit (GPU)	1× NVIDIA A100-SXM4 with 80 GB of GPU memory
Host System	NVIDIA Driver Version	570.211.01
Host System	CUDA Version	CUDA 12.8
Container Runtime	Containerization Technology	Apptainer (formerly Singularity)
Inference Engine	vLLM	v0.17.0
Inference Engine	SGLang	v0.5.10
Inference Engine	llama.cpp	Build b9115

Table 2: Hardware and software configuration of the experimental environment

To reflect the typical deployment configurations of the inference engines, all three

engines were run with their default settings except for the parameters described below. For llama.cpp, all model layers were placed on the GPU using `-ngl 999`, and `-ctx-size` was configured according to the workload. The number of parallel slots was kept at the default value of 4 , and the context space was shared among all slots. For vLLM, the GPU memory preallocation ratio was set to `gpu_memory_utilization=0.8` (the default value is 0.9) to avoid out-of-memory errors during execution. SGLang used its default value of `mem_fraction_static=0.7`.

4.2.2 Models

To evaluate the influence of model size on the performance of different inference engines, this experiment selected representative open-source large language models from the current open-source community and covered different parameter sizes. This thesis selected the 3B, 7B, and 14B models from the Qwen2.5 series [Yan+24] for testing. Using different parameter sizes from the same model family can minimize the influence of differences in model architecture, vocabulary, and training methods as much as possible, thereby allowing a more focused analysis of the influence of increasing model parameter size on latency, throughput, and peak GPU memory.

To further verify whether the performance differences among the three inference engines (RQ1), as well as their best applicable scenarios (RQ3), can be extended to different model architectures, this experiment additionally selected other representative models.

- **Dense architecture comparison:** Llama-3.2-3B-Instruct[GDJ+24] and Llama-3.1-8B-Instruct[GDJ+24] were selected to observe the optimization effects of the inference engines on different Dense architectures with similar parameter sizes.
- **MoE architecture extension:** This thesis selected GPT-OSS-20B, which adopts a sparse expert routing mechanism. Unlike Dense models, MoE models activate only part of the expert parameters when processing each token and therefore have different computation patterns and GPU memory access characteristics.

llama.cpp supports multiple GGUF quantization schemes, and different schemes involve different trade-offs among model size, GPU memory usage, inference speed, and model quality. Table 3 summarizes the main quantization formats. Lower-bit quantization provides a higher compression ratio but usually results in more noticeable accuracy loss, whereas higher-bit quantization can better preserve model quality but requires more storage space and GPU memory.

To reflect common practical deployment scenarios of llama.cpp, this thesis selected the Q4_K_M quantization format in order to achieve a good balance among model size, GPU memory usage, and inference performance.

4.2.3 Datasets and Workloads

To evaluate the influence of prompt length on the performance of inference engines, this experiment constructed three workload categories: small, medium, and big. The small and medium workloads were derived from the ShareGPT dataset [Sha], whereas the big workload was generated using the random dataset functionality of the benchmark tool.

ShareGPT contains conversations between real users and large language models and can therefore be used to simulate natural-language inference requests. To ensure that each request could be used as an independent and complete input, this thesis extracted only

Scheme	Approx. Bits	Brief Description
Q3_K_M	~ 3	K-block 3-bit quantization providing a medium trade-off between model compression and model quality.
Q4_K_M	~ 4	A newer K-block 4-bit quantization scheme designed to balance model size, inference performance, and model quality.
Q6_K	~ 6	A higher-quality K-block quantization scheme that preserves more model information but requires more storage and memory.
Q8_0	~ 8	A high-precision quantization scheme that is close to floating-point model quality but provides less compression.

Table 3: Representative llama.cpp GGUF quantization schemes, adapted from Kurt [Kur26]

Model Name	Size	Architecture	Floating-Point Format	llama.cpp Quantization Format
Qwen2.5-3B-Instruct	3.0B	Dense	BF16	GGUF (Q4_K_M)
Qwen2.5-7B-Instruct	7.0B	Dense	BF16	GGUF (Q4_K_M)
Qwen2.5-14B-Instruct	14.0B	Dense	BF16	GGUF (Q4_K_M)
Llama-3.2-3B-Instruct	3.0B	Dense	BF16	GGUF (Q4_K_M)
Llama-3.1-8B-Instruct	7.0B	Dense	BF16	GGUF (Q4_K_M)
GPT-OSS-20B	21.0B	MoE	BF16	GGUF (Q4_K_M)

Table 4: Selected large language models and their configurations

the first prompt submitted by the human user in each conversation when constructing the small and medium workloads. This avoids later conversational turns containing short requests such as "continue" and "please elaborate," which depend on the preceding context. The resulting small and medium workloads therefore mainly simulate independent single-turn requests rather than complete multi-turn conversation scenarios.

This experiment divided the extracted ShareGPT requests according to the number of tokens in each prompt. The maximum prompt length of the small workload was set to 128 tokens, corresponding to `SMALL_MAX = 128`. For the medium workload, the maximum prompt length was set to 512 tokens `MEDIUM_MAX = 512`. The built-in ShareGPT sampler of the vLLM serving benchmark applies default filtering rules and retains only requests whose prompt length does not exceed 1,024 tokens and whose combined prompt and output length does not exceed 2,048 tokens. To better observe the influence of prompt length on inference engines, this experiment constructed the big workload using randomly generated token sequences. Each request in this workload had a fixed prompt length of 20,000 tokens and was used to evaluate the performance of the three inference engines when processing extremely long inputs.

To ensure comparability across different experiments, each workload contained 8,000 prompts, and the output length of each request was fixed at 256 tokens.

Workload	**Data Source**	**Prompt Length**	**Number of Prompts**	**Output Length**
Small	ShareGPT	$\leq$ 128 tokens	8,000	256 tokens
Medium	ShareGPT	$\leq$ 512 tokens	8,000	256 tokens
Big	Randomly generated	20,000 tokens	8,000	256 tokens

Table 5: Dataset and workload configurations

4.2.4 Concurrency Control

To evaluate the impact of concurrent requests on inference performance, this thesis configured three maximum concurrency levels: 1, 4, and 16, representing typical low-, medium-, and high-load scenarios, respectively.

Maximum concurrency refers to the upper limit on the number of simultaneously active requests and is not equivalent to a fixed internal batch size. Modern inference engines such as vLLM use continuous batching, dynamically adjusting the actual batch size at runtime according to GPU memory availability and the status of active requests.

In addition, this thesis conducted stress tests without a concurrency limit. Requests were sent to the inference service at the highest possible rate to evaluate the throughput capability of each inference engine under high-load conditions.

4.3 Benchmark Implementation

The experiments were run using Slurm batch scripts. Each experiment requested one NVIDIA A100-SXM4 GPU with 80 GB of GPU memory. After the resources were allo-

cated, the script started the inference engine in the corresponding Apptainer container and loaded the specified model.

The entire evaluation adopted a client–server architecture. The inference server and the benchmark client ran on the same computing node and communicated through a local network address, thereby reducing the influence of external network latency on the experimental results.

Each benchmark run was defined by the inference engine, model, workload, and maximum concurrency. The execution script was responsible for setting the model path, dataset path, number of prompts, output length, context length, request rate, and maximum concurrency. Each workload contained 8,000 prompts, and the output length of each request was fixed at 256 tokens. In the fixed-load experiments, the maximum concurrency was set to 1, 4, and 16. For the stress tests, no maximum concurrency was set, and the request rate was set to infinity through `-request-rate inf`, allowing the benchmark client to submit requests at the highest possible rate.

Before each benchmark started, the script waited for the inference server to finish loading the model and become ready to receive requests. If the server failed to start, exited unexpectedly, or exceeded the available GPU memory, the benchmark was terminated, and the corresponding server log was retained for later analysis. After a benchmark was successfully completed, the server process was stopped before the next experimental configuration was executed.

The benchmark tools were used to record latency- and throughput-related metrics, including Time to First Token, Inter Token Latency, total throughput, and output throughput. GPU memory usage was independently collected at fixed time intervals using `nvidia-smi`, and the maximum value observed in the monitoring log was used as the peak GPU memory usage of the corresponding experiment.

Listing 1: General benchmark execution procedure.

```
# Phase 1: Initialization & Environment Setup
allocate_slurm_resources()
load_container_environment(engine)
check_model_and_dataset_files()

# Phase 2: Server Startup & Monitoring
start_gpu_memory_monitoring()
start_inference_server(
    engine,
    model,
    context_length
)
wait_until_server_is_ready()

# Phase 3: Benchmark Execution
run_benchmark(
    workload,
    num_prompts=8000,
    output_length=256,
    request_rate,
    max_concurrency
)

```

```
# Phase 4: Data Processing & Cleanup
save_benchmark_results()
save_server_logs()
calculate_peak_gpu_memory()

stop_inference_server()
stop_gpu_memory_monitoring()
```

5 Evaluation

This chapter presents the experimental results of vLLM, SGLang, and llama.cpp, comparing their performance under different model sizes, prompt lengths, and concurrency settings. The evaluation mainly focuses on latency, throughput, peak GPU memory usage, and performance under stress testing conditions.

Section 5.1 presents the latency results, including TTFT and ITL. Section 5.2 compares the overall throughput and output throughput of the three inference engines. Section 5.3 analyzes the peak GPU memory usage across different models and workloads. Section 5.4 shows the results of the stress tests without a fixed concurrency upper bound. Finally, Section 5.5 summarizes the main findings of this chapter.

5.1 Latency Evaluation

This section evaluates the latency performance of vLLM, SGLang, and llama.cpp under different model sizes, workloads, and concurrency settings. The analysis focuses on TTFT and ITL. TTFT represents the time from sending a request until the first output token is returned, whereas ITL describes the delay between consecutively generated tokens. Together, these two metrics reflect the initial responsiveness and generation speed of an inference engine.

5.1.1 TTFT

Figure 2 presents the TTFT of vLLM, SGLang, and llama.cpp under different models, workloads, and maximum concurrency levels.

Overall, TTFT increased as prompt length and concurrency increased. vLLM showed the lowest TTFT under most medium- and high-concurrency conditions. SGLang performed similarly to vLLM at low concurrency and even achieved better TTFT than vLLM in some cases with longer prompts under low-concurrency conditions.

Under the small workload and low-concurrency conditions, the TTFT of llama.cpp was similar to that of the other two inference engines. However, llama.cpp was more sensitive to changes in concurrency and prompt length. As the concurrency increased to 4 and 16, its TTFT increased significantly and became considerably higher than that of vLLM and SGLang.

Within the Qwen2.5 series, TTFT generally increased as the model size grew from 3B to 14B, and larger models usually required more time to return the first token. In addition, differences in TTFT were also observed between Qwen and Llama models with similar parameter sizes, indicating that model architecture also affects inference latency.

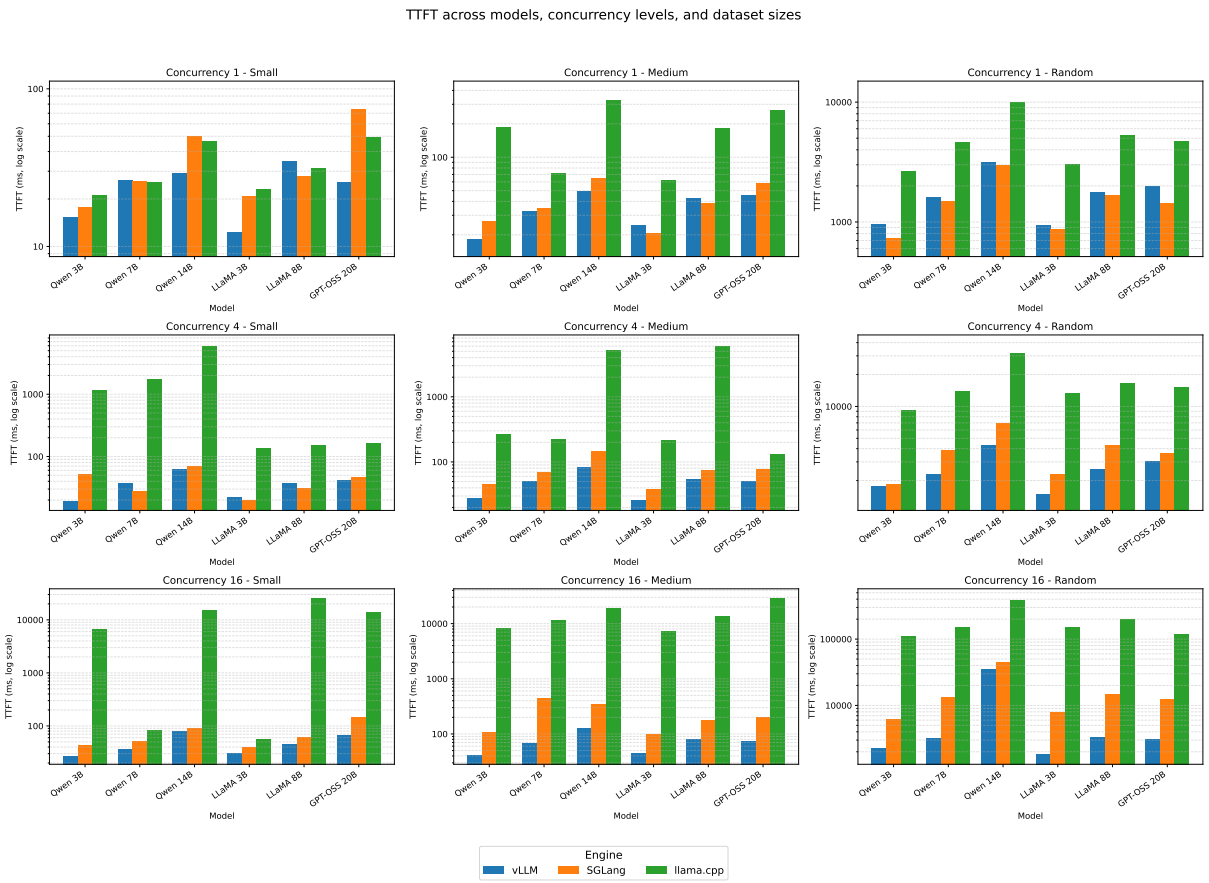

Figure 2: TTFT comparison across different models, workload types, and maximum concurrency levels. Rows represent concurrency levels of 1, 4, and 16, while columns represent the small, medium, and random 20K workloads. The y-axis uses a logarithmic scale.

5.1.2 Inter-Token Latency

Figure 3 shows the mean inter-token latency of vLLM, SGLang, and llama.cpp across different models, workloads, and maximum concurrency levels.

As shown in the figure, ITL generally increases with the concurrency level. At concurrency 1, the differences among the three inference engines are relatively small overall, with llama.cpp exhibiting lower ITL. As concurrency increases to 4 and 16, the ITL of vLLM and SGLang remains relatively low, while llama.cpp shows a more pronounced increase under some configurations.

This difference is most prominent in the random 20K workload. At concurrency 4 and 16, the ITL of llama.cpp is significantly higher than that of vLLM and SGLang. Among these, Qwen 14B shows the largest gap, with llama.cpp's ITL reaching approximately 350–400 ms, several times higher than that of the other two inference engines. In contrast, vLLM and SGLang exhibit more stable performance under long-prompt and high-concurrency conditions.

Further comparison between vLLM and SGLang reveals that, under high-concurrency and long-prompt workload conditions, namely the random workload at concurrency levels 4 and 16, SGLang's ITL is generally lower than that of vLLM. This pattern is observed across all six models, with a gap of approximately 1.5–2 times.

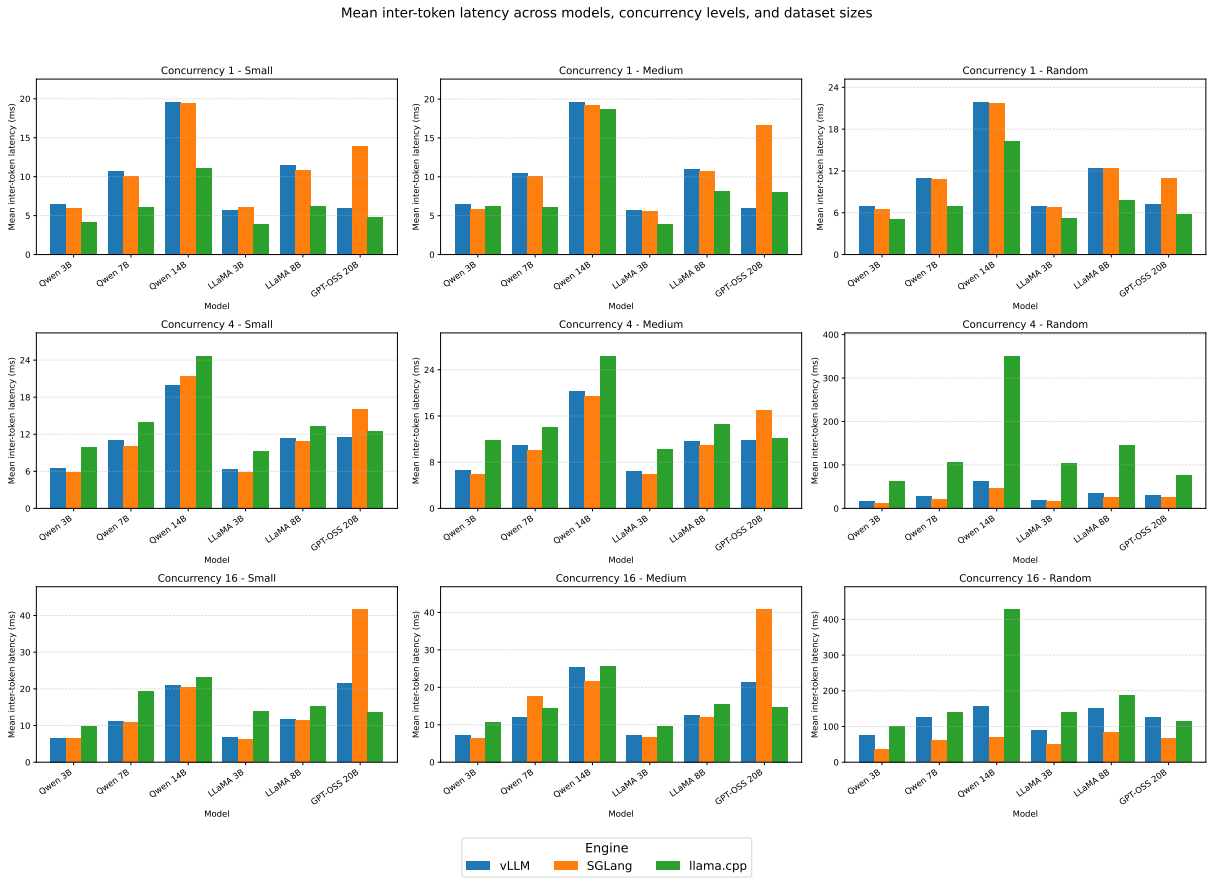

Figure 3: Inter-token latency comparison across different models, workload types, and maximum concurrency levels. Rows represent concurrency levels of 1, 4, and 16, while columns represent the small, medium, and random 20K workloads.

Regarding the effect of model scale, ITL generally increases with parameter scale. This trend holds for both the Qwen and LLaMA series and can be observed across all three inference engines.

However, this pattern does not fully hold when comparing different model families. Although GPT-OSS 20B has more parameters than Qwen 14B, the figure shows that its ITL does not increase accordingly. More notably, under the high-concurrency random workload, although GPT-OSS 20B also exhibits higher ITL on llama.cpp than on vLLM and SGLang, the magnitude of this increase is considerably smaller than that observed for Qwen 14B.

This suggests that model parameter count is not the sole determinant of ITL and that the underlying model architecture also has a significant impact on token generation speed.

5.2 Throughput Evaluation

This section evaluates the processing capacity of the three inference engines across different models, workloads, and maximum concurrency settings. Throughput represents the number of tokens processed per unit of time and thus serves as a key metric for measuring the efficiency of inference serving.

Two throughput metrics are employed in this section. Output throughput denotes the number of output tokens generated per second, while total throughput encompasses both

input and output tokens. The following analysis first compares the output throughput and subsequently examines the total throughput.

5.2.1 Output Throughput

Figure 4 compares the output token throughput of the three inference engines across different models, workloads, and maximum concurrency levels.

At concurrency 1, llama.cpp achieves the highest output throughput in most configurations under the small and medium workloads, while vLLM and SGLang show relatively similar performance. This is mainly due to the Q4_K_M quantized weights used by llama.cpp: under a single request, the decode stage is limited by GPU memory bandwidth, and the amount of weight data accessed per token with 4-bit weights is only approximately one quarter of that with BF16. However, under the random 20K workload, llama.cpp already falls behind even at concurrency 1, indicating that its advantage is limited to short-prompt scenarios.

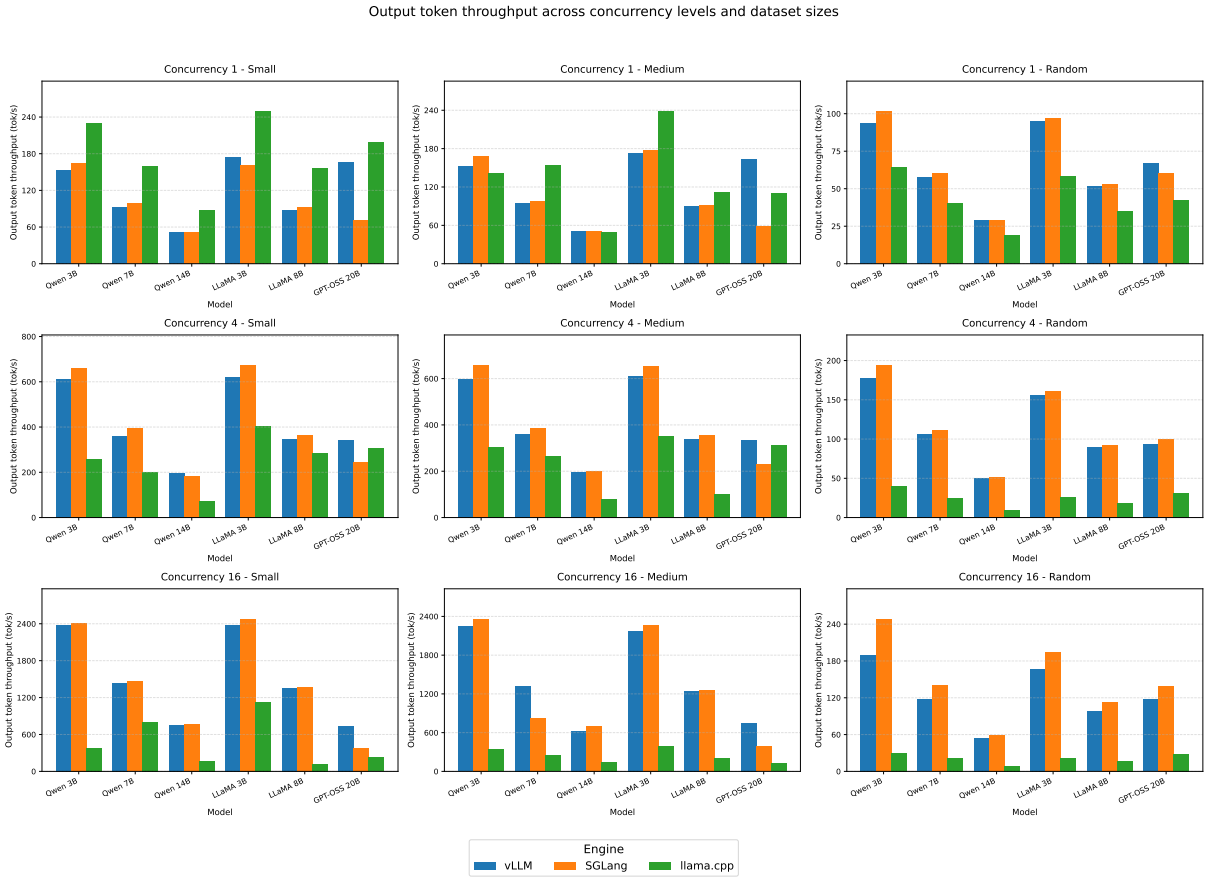

Figure 4: Output token throughput across different models, workload types, and maximum concurrency levels. Rows represent concurrency levels of 1, 4, and 16, while columns represent the small, medium, and random 20K workloads.

As concurrency increases, the performance pattern reverses. From concurrency 1 to 16, the throughput of vLLM and SGLang increases by approximately 14–15 times, reaching about 2,000–2,500 tokens/s on the 3B models. At concurrency 4, SGLang slightly outperforms vLLM in most configurations. In contrast, the throughput growth of llama.cpp is very limited, and it clearly falls behind the other two engines under high concurrency.

Under the random 20K workload, the absolute output throughput is significantly lower than under the other workloads because most of the processing time for each request is spent on the prefill of 20K tokens. Under this configuration, the advantage of SGLang increases with concurrency: at concurrency 16, its output throughput is higher across all six models. Combined with the results in Section 5.1, where SGLang shows higher TTFT but lower ITL under this configuration, this indicates that the two engines exhibit different performance characteristics under long-prompt and high-load conditions: SGLang tends to favor batched throughput efficiency, while vLLM tends to favor faster first-token response. llama.cpp shows lower performance than the other two inference engines in all long-prompt and high-concurrency configurations.

In addition, the consistently low output throughput of SGLang on GPT-OSS 20B under the small and medium workloads may indicate that its support for this MoE architecture is not well optimized.

From the model perspective, the output throughput of all engines decreases as the model size increases. However, GPT-OSS 20B does not fully follow this parameter-count trend: because its MoE architecture activates only a subset of parameters for each token, its throughput is comparable to that of dense models at the 8B scale. This indicates that output throughput is also affected by model architecture.

5.2.2 Total Throughput

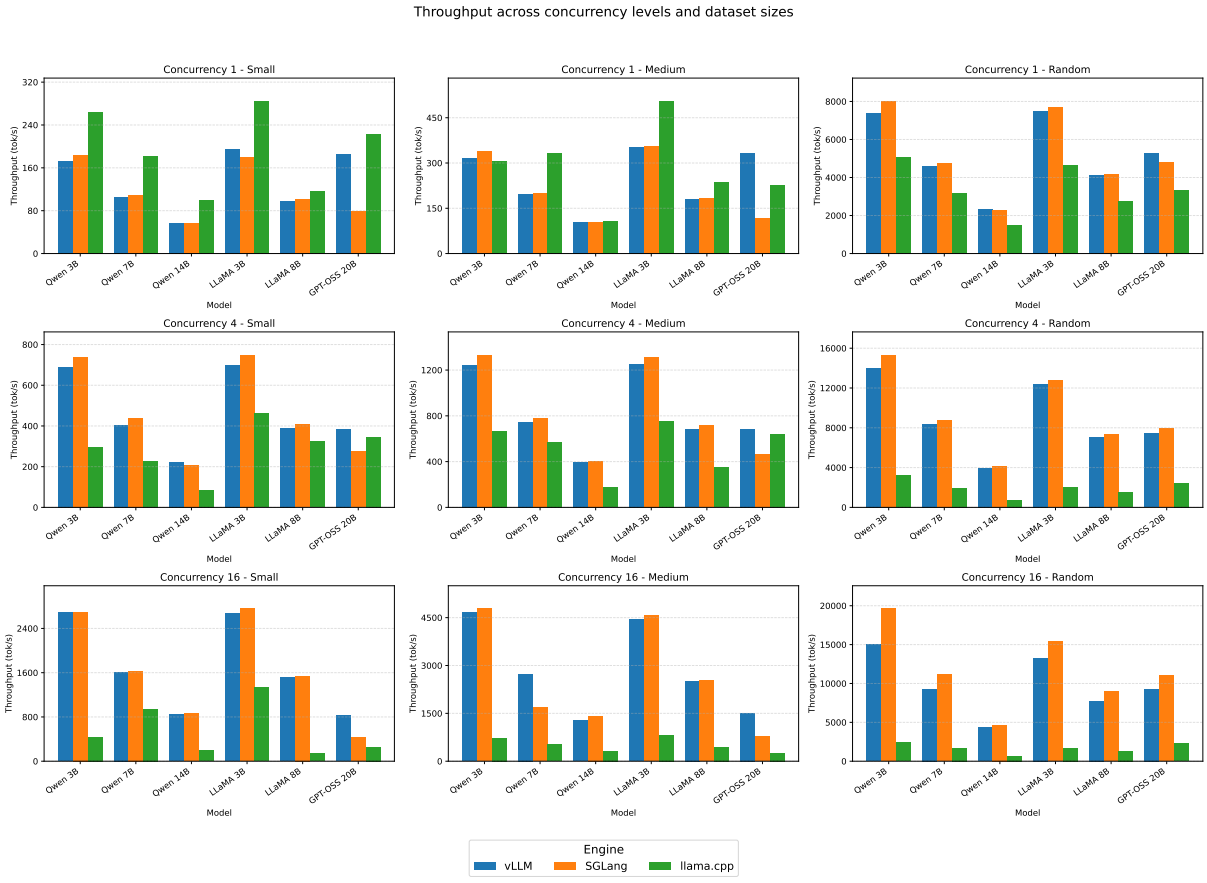

Figure 5: Total throughput across different models, workload types, and maximum concurrency levels. Rows represent concurrency levels of 1, 4, and 16, while columns represent the small, medium, and random 20K workloads.

Figure 5 shows the total throughput of the three inference engines across different models, workloads, and maximum concurrency levels. Unlike output throughput, total throughput includes both input and output tokens. Therefore, the values under the random 20K workload are significantly higher than those under the small and medium workloads. Since each request in this workload contains 20,000 input tokens, while the output length is only 256 tokens, the total throughput under this configuration mainly reflects the engines' processing capability during the prefill stage.

Under the small and medium workloads, the overall pattern of total throughput is broadly consistent with that of output throughput: at concurrency 1, llama.cpp leads in most configurations; as concurrency increases to 4 and 16, vLLM and SGLang improve significantly, while the growth of llama.cpp remains very limited.

The random 20K workload shows a different result. First, even at concurrency 1, the total throughput of vLLM and SGLang is already higher than that of llama.cpp. This is because prefill dominates under long-input conditions, and the 4-bit quantization used by llama.cpp cannot provide a memory-access advantage at this stage.

At concurrency 16 under the random workload, SGLang achieves the highest total throughput across all models. This ranking is consistent with the output throughput results in Section 5.2.1, where SGLang also outperforms vLLM under the same configurations. However, the two metrics reflect different aspects of performance: total throughput is mainly affected by the large number of input tokens, whereas output throughput only counts generated tokens. Taken together, these results show that SGLang can achieve a higher overall processing rate under long-input and high-load conditions. Combined with the results in Section 5.1, where SGLang shows higher TTFT but lower ITL, it can be seen that SGLang is more favorable for overall batched processing efficiency in this scenario, while vLLM can return the first token more quickly.

From the model perspective, total throughput decreases as the parameter count increases within the same model family. GPT-OSS 20B does not fully follow this parameter-count trend, as its MoE architecture allows it to achieve performance close to that of dense models at the 8B scale, indicating that model architecture also affects overall token processing efficiency.

5.2.3 Relationship between Total Throughput and Concurrency under the Random 20K Workload

Figure 6 shows how the total throughput of each engine changes with the maximum concurrency level under the random 20K workload. The three engines exhibit distinctly different scaling curves.

Both vLLM and SGLang scale positively with increasing concurrency, but their trends differ. The throughput of vLLM increases mainly between concurrency levels 1 and 4, with only a slight improvement from concurrency 4 to 16. In contrast, SGLang continues to increase beyond concurrency 4, from approximately 8,000 to 15,300 and then to 19,600 tokens/s. SGLang's scheduling continues to improve the batching efficiency of long-input requests at higher concurrency levels.

llama.cpp, by contrast, exhibits negative scaling: its throughput decreases rather than increases as concurrency rises, and almost all models achieve their highest throughput at concurrency 1. This means that, for llama.cpp, increasing concurrency not only fails to improve the overall processing capacity, but also reduces efficiency because multiple long-prompt requests compete for limited parallel execution slots and interfere with one another. This further confirms that the appropriate operating point for llama.cpp is a

single request or very low concurrency, and that it should not be used for concurrent serving.

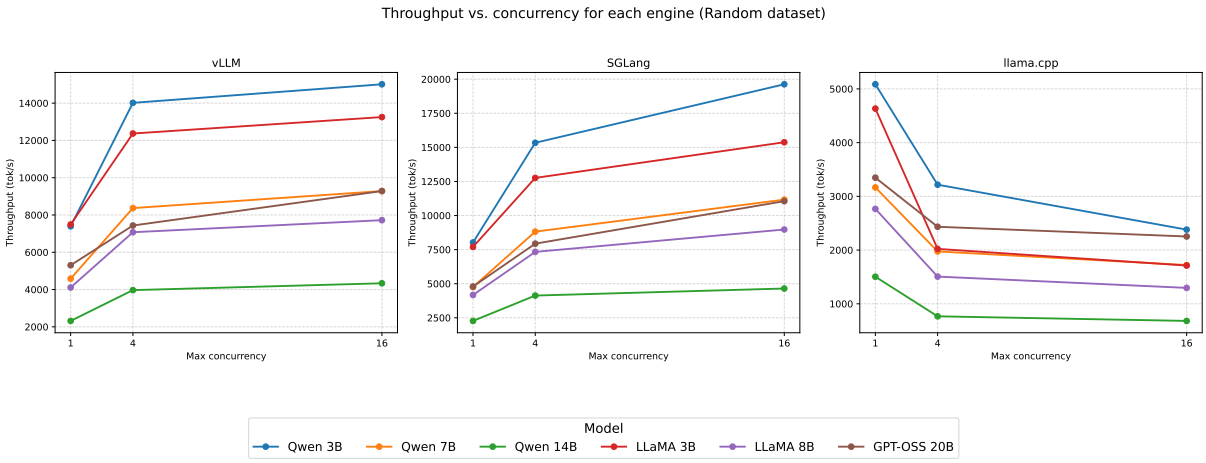

Figure 6: Total throughput as a function of maximum concurrency for vLLM, SGLang, and llama.cpp under the random 20K workload. Each panel represents one inference engine, while the lines represent different models.

The scaling curves under the small and medium workloads are provided in Appendix B. Unlike the random workload, vLLM and SGLang maintain approximately linear scaling under these two workloads, with no clear sign of saturation even at concurrency 16. The throughput of llama.cpp increases slightly from concurrency 1 to 4. Between concurrency 4 and 16, only some of the smaller models continue to improve, while the remaining models tend to stagnate or even decline.

5.3 Peak GPU Memory Evaluation

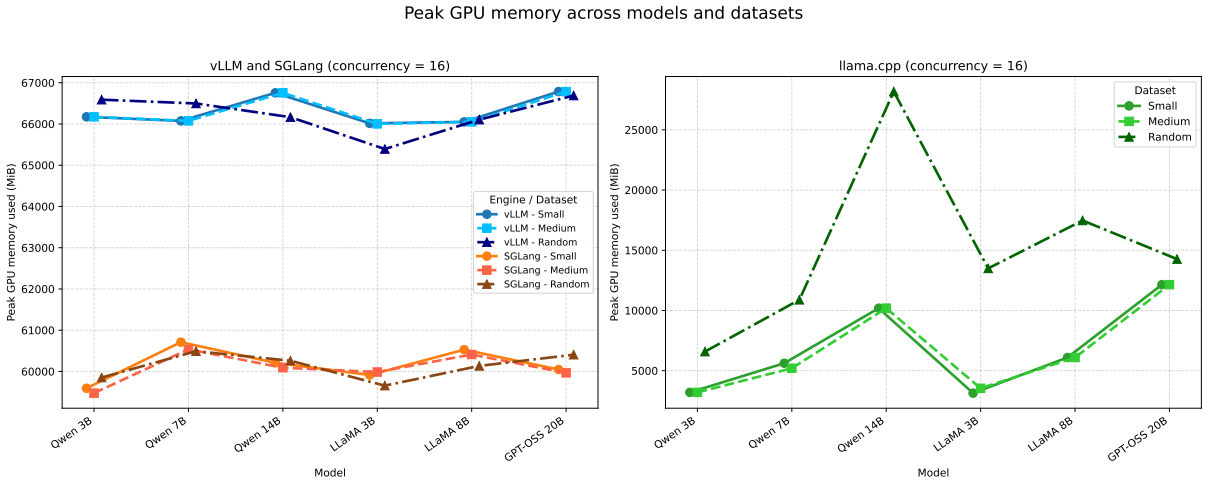

Figure 7: Peak GPU memory usage across different models and workloads at concurrency 16. The left panel shows the results for vLLM and SGLang, while the right panel shows the results for llama.cpp.

Figure 7 shows the peak GPU memory usage of the three engines across different models and workloads at concurrency 16. The two categories of engines exhibit distinctly different memory usage behavior.

The peak GPU memory usage of vLLM and SGLang remains almost unchanged across different model sizes and prompt lengths, staying at approximately 66 GB and 60 GB, respectively. This is because both engines preallocate a fixed proportion of the total GPU memory as a KV cache pool during startup and subsequently manage memory dynamically within this pool. Their peak GPU memory usage therefore reflects the startup configuration rather than the actual workload demand.

In contrast, llama.cpp shows substantially lower peak GPU memory usage, but it is more strongly affected by model size and prompt length. Under the small and medium workloads, its memory usage generally increases with model size, rising from approximately 3 GiB for the 3B models to approximately 12 GiB for GPT-OSS 20B. The random 20K workload requires more memory because longer input sequences produce a larger KV cache. Among all configurations, Qwen 14B shows the highest peak memory usage, reaching approximately 28 GiB.

Overall, vLLM and SGLang reserve a large and relatively fixed amount of GPU memory in advance to support KV cache allocation under high concurrency, thereby trading memory usage for higher throughput. llama.cpp, by contrast, tends to allocate memory dynamically according to the model and workload. As a result, it uses less memory under lighter workloads, but its memory usage increases noticeably with model size and prompt length.

The results at concurrency levels 1 and 4 show the same overall trend. Since vLLM and SGLang reserve a certain proportion of GPU memory during server initialization, their peak GPU memory usage remains nearly unchanged. In contrast, the memory usage of llama.cpp increases with concurrency. The complete results for all concurrency settings are provided in Appendix A.

5.4 Stress-Test Evaluation

This section evaluates the maximum throughput capacity of vLLM and SGLang under extreme load by setting the request rate to infinity (`-request-rate inf`) and removing the concurrency limit. llama.cpp was not included in this test. The previous fixed-concurrency experiments showed that its throughput improved only slightly as concurrency increased and remained significantly lower than that of vLLM and SGLang at higher concurrency levels. Therefore, this section compares only vLLM and SGLang.

Figure 8 (with total throughput shown on the y-axis) shows that both engines achieve substantially higher throughput in the stress test than at a concurrency level of 16. This indicates that a fixed concurrency level of 16 was not sufficient to saturate the A100. After removing the concurrency limit, the engines were able to schedule more requests simultaneously, thereby further increasing total throughput.

Under the small and medium workloads, vLLM achieves higher throughput on Qwen 3B and LLaMA 3B, while SGLang performs better on Qwen 7B, Qwen 14B, LLaMA 8B, and GPT-OSS 20B.

Under the random 20K workload, SGLang outperforms vLLM across all six models. This is consistent with the trend observed at fixed concurrency 16 in Section 5.2, indicating that SGLang's prefill processing advantage under workloads dominated by long prompts remains as the load increases.

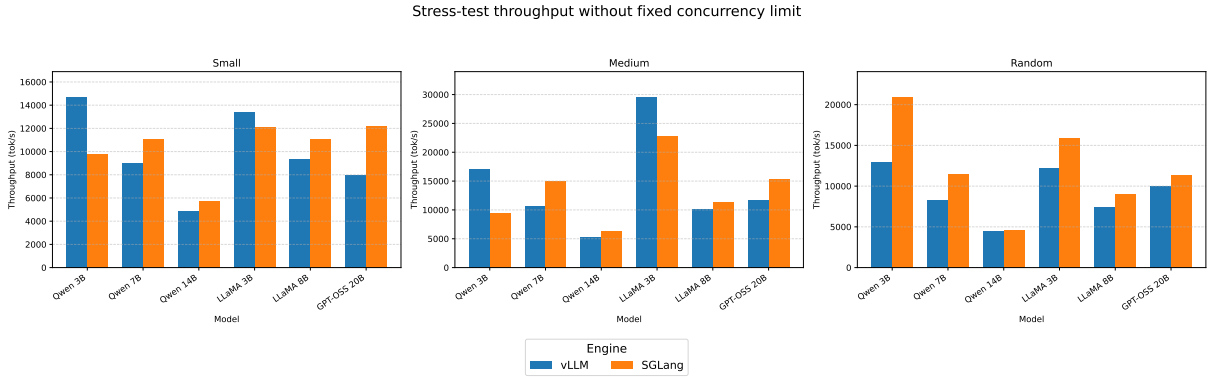

Figure 8: Total throughput of vLLM and SGLang under the stress-test setting without a fixed concurrency limit. The results are shown across different models and workloads.

The stress-test results show that the throughput capacities of both engines are far from saturated at concurrency 16, and that their relative advantages depend on workload characteristics: vLLM achieves higher throughput for smaller models and short-prompt scenarios, while SGLang performs better for larger models and long-prompt scenarios. Meanwhile, llama.cpp is unable to maintain stable service under unrestricted load, further confirming that it is not suitable for high-load multi-user scenarios.

5.5 Summary of Evaluation Results

We address the following research questions:

RQ1: What differences exist among vLLM, SGLang, and llama.cpp in terms of latency, throughput, and peak GPU memory?

The three inference engines exhibit clearly different performance characteristics. vLLM achieves the lowest TTFT in most medium- and high-concurrency configurations and therefore provides the fastest first-token response in this evaluation. The throughput of SGLang is generally comparable to that of vLLM, and it performs particularly well under long-input and high-concurrency workloads. However, under these conditions, its TTFT also increases more noticeably. The advantages of llama.cpp mainly appear under single-request and short-prompt workloads, while its throughput scaling is clearly weaker than that of the other two engines as concurrency increases.

In terms of peak GPU memory usage, vLLM and SGLang remain stable at approximately 66 GiB and 60 GiB, respectively, because of their preallocation mechanisms. In contrast, llama.cpp dynamically allocates GPU memory according to concurrency, model size, and workload, resulting in peak memory usage of approximately 3–28 GiB.

RQ2: How do model size, prompt length, and concurrency affect the relative performance of the three inference engines?

All three factors clearly affect the relative performance of the inference engines. Among them, concurrency is the most distinct dividing factor. llama.cpp performs well at concurrency 1, whereas vLLM and SGLang achieve larger throughput improvements as concurrency increases. llama.cpp shows no clear additional throughput improvement beyond low concurrency under its default configuration of four parallel slots.

Prompt length changes the dominant stage of the inference process. Under short-prompt workloads, the decode stage accounts for a larger proportion of the total execution time, allowing the quantized llama.cpp configuration to achieve a certain advantage.

Under the random 20K workload, the prefill stage dominates. In this case, llama.cpp falls behind the other two engines even at concurrency 1, while SGLang becomes more competitive relative to vLLM.

Within the same model family, performance generally decreases as model size increases. However, GPT-OSS 20B does not fully follow this trend because its MoE architecture activates only a subset of parameters during the generation of each token.

RQ3: Under which practical usage scenarios is each inference engine the best choice?

The experimental results show that the three inference engines are suitable for different deployment scenarios. llama.cpp is more suitable for single-user or local deployment scenarios with limited GPU memory. vLLM is more suitable for latency-sensitive online services, particularly when a fast first-token response is required. SGLang is more suitable for workloads that prioritize throughput, especially those involving long prompts or larger models.

A detailed discussion of these deployment recommendations is provided in Table 6.

Table 6: Recommended inference engines for typical deployment scenarios

Scenario	**Recommended Engine**
Single-user or local deployment with limited GPU memory	llama.cpp
Latency-sensitive interactive serving	vLLM
High-throughput serving with long inputs	SGLang
Serving larger models under heavy load	SGLang
Serving small models with short prompts	vLLM

6 Discussion

6.1 Challenges

6.1.1 Challenges in Containerized Engine Deployment

One of the main challenges of this experiment was the deployment and compatibility of multiple inference engines within the same HPC environment. This study required vLLM, SGLang, and llama.cpp to be run separately on the GWDG cluster, while these systems depend on different software stacks.

To address this issue, the three inference engines were packaged and executed in separate Apptainer containers. Through containerization, each engine was able to use a runtime environment matching its version, thereby reducing dependency conflicts and improving the reproducibility of the experiments.

Even so, establishing comparable execution environments still involved several practical issues. For example, SGLang also depends on several runtime cache directories during execution, including FlashInfer, Triton, and the CUDA cache. Therefore, this experiment additionally used the cluster's temporary workspace as a writable cache directory and mounted it inside the container through Apptainer bind options and environment-variable settings. This step was necessary because some default paths inside the container could be read-only. If the cache directories were not configured correctly, runtime compilation or initialization could fail.

In addition, some earlier versions of SGLang produced throughput far below the expected level when running the models selected in this study. To exclude the influence of version-specific defects on the comparison results, multiple SGLang versions were tested during the experiment. Version v0.5.10, which showed stable throughput performance, was ultimately selected for the evaluation.

Furthermore, because the compute nodes had no external network access, Hugging Face offline mode had to be enabled, and all model files had to be downloaded in advance.

Engine deployment involved more than simple software installation. It also required handling container construction, file-system mounting, cache-path configuration, and offline model loading. These environment compatibility issues increased the complexity of the experimental setup and required extensive debugging and validation during the early stages of the study.

6.1.2 Constructing a Controllable Long-Prompt Workload

The experiment initially planned to derive three workloads of different sizes from ShareGPT in order to keep the data source consistent. However, during the experiments, it was found that the actual input lengths of the long-prompt group differed substantially across the inference engines. When the same workload was used, the number of input tokens processed by SGLang was several times larger than that processed by vLLM, making it impossible to control the variables required for the experiment.

Further investigation confirmed that the built-in ShareGPT sampler of `vllm bench serve` applies default filtering rules. It only retains requests with a prompt length of no more than 1,024 tokens and with a combined prompt and output length of no more than 2,048 tokens. Therefore, it was not possible to obtain long inputs of several thousand tokens or more from this dataset.

For this reason, this study instead used the random dataset functionality of the benchmark tool to construct the random 20K workload, with the input length fixed at 20,000 tokens. The advantage of this approach is that the input length can be strictly controlled, making it easier to isolate prompt length as an experimental variable. The trade-off is that random token sequences do not possess the statistical characteristics of natural language.

6.2 Result Interpretation

The experimental results indicate a trade-off between TTFT and throughput. Under long-input and high-concurrency conditions, the TTFT of SGLang is noticeably higher than that of vLLM, differing by roughly an order of magnitude in some configurations, but SGLang achieves a lower ITL and higher throughput. In contrast, vLLM usually returns the first token faster, while its overall throughput is slightly lower than that of SGLang.

This phenomenon suggests that the two engines may have different optimization priorities in request scheduling. SGLang favors batch-processing efficiency: some requests may wait longer before generation starts, but once a request begins execution, subsequent tokens are generated faster and the overall processing capacity is higher. vLLM places more emphasis on first-token responsiveness and is therefore better suited to scenarios with strict interactive latency requirements.

Neither approach is inherently superior, but rather represents a concrete manifestation of the classic latency–throughput trade-off in LLM inference scheduling.

The Q4_K_M quantization used by llama.cpp gives it strong single-request performance under short-prompt, low-concurrency conditions. Our experiments show that quantization does not deliver the same performance gains across all workloads. When the decoding stage dominates, such as in the small workload, llama.cpp can benefit more from its quantized models because the smaller model weights reduce the GPU memory access requirements during token generation. When the prefilling stage dominates, this advantage becomes significantly weaker. In this case, even under low-concurrency conditions, SGLang and vLLM can achieve higher throughput because quantization does not provide any benefit for the compute-bound prefilling stage.

This means that in practical deployment, an inference engine should not be selected based only on the result of a single benchmark. The performance of an inference engine depends not only on the overall size of the workload, but also on the proportion of the workload spent in the Prefill and Decode stages, which is mainly determined by the input and output lengths. An engine that performs well for short-input, decode-dominated workloads may not maintain the same advantage for long-input, prefill-dominated workloads. Therefore, when selecting an inference engine, the expected prompt length, output length, and concurrency level of the actual application should be considered.

Compared with its performance benefits, the effect of quantization on GPU memory usage is more consistent. Owing to the Q4_K_M quantized models and on-demand memory allocation, the peak GPU memory usage of llama.cpp in our experiments is approximately 3–28 GiB, which is significantly lower than that of vLLM and SGLang. Therefore, the low memory requirement is one of the clearest and most consistent advantages of llama.cpp in this study. However, this advantage does not come without a cost. Although llama.cpp uses significantly less GPU memory than vLLM and SGLang, its throughput is also lower under long-input and high-concurrency workloads. In contrast, vLLM and SGLang allocates a larger proportion of GPU memory, thereby providing more space for

KV cache management and concurrent request processing. As a result, they can achieve higher throughput and stronger concurrency capability under high-load conditions. This shows that lower GPU memory usage does not necessarily imply higher overall efficiency, but rather reflects a different trade-off between memory usage and concurrent serving capability.

Within the same model family, as the model parameter size increases, the throughput of the three engines generally decreases, while latency and GPU memory requirements generally increase. However, performance across different model families cannot be compared solely on the basis of total parameter count: although GPT-OSS 20B is the largest model in our experiments, its performance is close to that of 8B-class dense models in most configurations.This behavior is attributable to its MoE architecture, in which only a subset of expert parameters is activated for each generated token. Therefore, the total parameter count does not represent the actual amount of computation performed in a single inference, since the inference cost is determined by the number of activated parameters.

In our experiments, the advantage of SGLang does not come from RadixAttention. The workloads in this study consist of mutually distinct single-turn requests or randomly generated token sequences, so there are almost no shared prefixes that can be reused across requests, leaving little room for RadixAttention's cache-reuse mechanism to take effect. Therefore, SGLang's throughput advantage under high load stems from the efficiency of its scheduling and execution itself rather than from prefix caching. What our experiments measure is its performance lower bound under the "no prefix reuse" condition. In practical scenarios such as multi-turn conversations or shared system prompts, its advantage may be further amplified.

6.3 Limitations

This experimental still has several limitations.

First, llama.cpp was evaluated using GGUF models quantized with Q4_K_M, whereas vLLM and SGLang used BF16 weights. This reflects relatively typical deployment configurations of the respective inference engines, but it also means that the performance differences observed in the experiments cannot be attributed entirely to the inference engine implementations themselves. For example, llama.cpp's performance advantage in some single-request scenarios and its lower GPU memory usage are also influenced by the use of quantized models.

Second, llama.cpp used the default configuration of four parallel slots. Therefore, in the experiments with a concurrency level of 16, at most four requests could be processed at the same time, while the remaining requests had to wait for an available slot. The context space of llama.cpp is configured when the server starts and is shared among all parallel slots. Therefore, when the number of parallel slots is increased, the total context space usually also needs to be increased accordingly, which further increases GPU memory requirements. Consequently, the conclusions of this thesis regarding the high-concurrency performance of llama.cpp should be understood as its performance under the default parallel-slot configuration used in this study and should not be directly generalized to other `-parallel` configurations.

Finally, this study also has certain limitations in terms of workload coverage. Prompt length was represented by only three ranges: no more than 128 tokens, no more than 512 tokens, and a long-input workload fixed at 20,000 tokens. Since the intermediate length ranges were not further tested, this study cannot determine the specific prompt-length

range at which the relative performance ranking among the inference engines changes. In addition, the output length was fixed at 256 tokens in all experiments, and workloads dominated by long-text generation, such as reasoning models that generate longer reasoning processes, were not further investigated.

6.4 Future work

Future work can extend the experiments of this thesis in several directions. First, more prompt lengths between 512 and 20,000 tokens could be included to analyze the effect of input length on inference performance in greater detail and to identify the specific prompt-length ranges at which the relative performance ranking among the inference engines changes. Second, the three inference engines could be compared under the same numerical precision. For example, quantized models could also be used for vLLM and SGLang, making it possible to better distinguish the impact of quantization itself from the impact of the inference engine implementation on performance differences. Similarly, future work could evaluate llama.cpp with a larger number of parallel slots to investigate how increasing `-parallel` affects throughput, latency, and GPU memory usage under high-concurrency workloads.

7 Conclusion

This thesis presents a systematic performance evaluation of three widely used open-source LLM inference engines: vLLM, SGLang, and llama.cpp. The research methodology was carefully designed to ensure the reliability of the results: all experiments were conducted on a Slurm-managed HPC cluster using an NVIDIA A100 GPU, with the three engines deployed in mutually isolated Apptainer containers. Using a controlled-variable approach, six models of different sizes and architectures were evaluated under three workloads and three concurrency levels, measuring TTFT, ITL, throughput, and peak GPU memory, complemented by stress tests without a concurrency limit.

The evaluation results reveal a clear stratification among the three engines. vLLM achieved the lowest TTFT in the vast majority of medium- and high-concurrency configurations, making it the best choice for latency-sensitive interactive serving. SGLang delivered the highest throughput under long-input, high-load conditions, reflecting a scheduling design that prioritizes batch throughput. llama.cpp, benefiting from 4-bit quantization, offered the best single-request performance and a memory footprint far below that of the other engines. The experiments further show that concurrency, prompt length, and model architecture jointly determine the relative ranking of the engines: no single engine is optimal under all conditions.

The contribution of this study lies in a comprehensive comparison of inference engines on production-grade HPC hardware across model size, prompt length, and concurrency, and in distilling the results into scenario-oriented selection recommendations:llama.cpp for single-user local deployment and memory-constrained environments, vLLM for interactive multi-user serving, and SGLang for throughput-oriented serving dominated by long inputs or larger models.

The findings of this thesis can help practitioners who deploy LLMs to select and configure the most appropriate inference engine for their specific deployment requirements.

References

[Boh+26] David Bohman et al. "Profiling and Visualizing CPU-Based LLM Inference in llama.cpp". In: *Bachelor Thesis, Chalmers University of Technology* (2026). URL: https://hdl.handle.net/20.500.12380/311717.

[Chi+24] Krishna Teja Chitty-Venkata et al. "LLM-Inference-Bench: Inference Benchmarking of Large Language Models on AI Accelerators". In: *2024 IEEE/ACM Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. 2024. arXiv: 2411.00136. URL: https://arxiv.org/abs/2411.00136.

[Fra+22] Elias Frantar et al. "GPTQ: Accurate Post-Training Quantization for Generative Pre-Trained Transformers". In: *CoRR* abs/2210.17323 (2022). DOI: 10.48550/arXiv.2210.17323. arXiv: 2210.17323. URL: https://arxiv.org/abs/2210.17323.

[GDJ+24] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. "The Llama 3 Herd of Models". In: *arXiv preprint arXiv:2407.21783* (2024). DOI: 10.48550/arXiv.2407.21783.

[ggm23a] ggml-org. *GGML: Tensor Library for Machine Learning*. https://github.com/ggml-org/ggml. Accessed: 2026-08-16. 2023.

[ggm23b] ggml-org. *GGUF File Format Specification*. https://github.com/ggml-org/ggml/blob/master/docs/gguf.md. Accessed: 2026-08-16. 2023.

[ggm23c] ggml-org. *llama.cpp*. https://github.com/ggml-org/llama.cpp. Accessed: 2026-06-28. 2023.

[Int26] Intel. "What Is High Performance Computing?" In: *Intel* (2026). URL: https://www.intel.com/content/www/us/en/learn/what-is-hpc.html.

[Kur26] Uygar Kurt. "Which Quantization Should I Use? A Unified Evaluation of llama.cpp Quantization on Llama-3.1-8B-Instruct". In: *arXiv preprint arXiv:2601.14277* (2026). DOI: 10.48550/arXiv.2601.14277. arXiv: 2601.14277 [cs.LG].

[Kwo+23] Woosuk Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *CoRR* abs/2309.06180 (2023). arXiv: 2309.06180. URL: https://arxiv.org/abs/2309.06180.

[Li+24] Baolin Li et al. "LLM Inference Serving: Survey of Recent Advances and Opportunities". In: *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. 2024, pp. 1–8. URL: https://arxiv.org/abs/2407.12391.

[Lin+24] Ji Lin et al. "AWQ: Activation-Aware Weight Quantization for On-Device LLM Compression and Acceleration". In: *Proceedings of Machine Learning and Systems* 6 (2024), pp. 87–100. URL: https://proceedings.mlsys.org/paper_files/paper/2024/hash/42a452cbafa9dd64e9ba4aa95cc1ef21-Abstract-Conference.html.

[LKG25] Anderson de Lima Luiz, Shubham Vijay Kurlekar, and Munir Georges. "Scalable Engine and the Performance of Different LLM Models in a SLURM Based HPC Architecture". In: *CoRR* abs/2508.17814 (2025). DOI: 10.48550/arXiv.2508.17814. arXiv: 2508.17814. URL: https://arxiv.org/abs/2508.17814.

[Mal25] Kyrylo S. Malakhov. "Deploying LLMs on CPU-only Environments with llama.cpp Library Set: MedLocalGPT Project Case". In: *Proceedings of the 5th International Workshop of IT-professionals on Artificial Intelligence (ProfIT AI 2025)*. Vol. 4164. CEUR Workshop Proceedings. CEUR-WS.org, 2025, pp. 163–176. URL: https://ceur-ws.org/Vol-4164/paper11.pdf.

[Par+26] Sihyeong Park et al. "A Survey on Inference Engines for Large Language Models: Perspectives on Optimization and Efficiency". In: *ACM Transactions on Intelligent Systems and Technology* (2026). DOI: 10.1145/3803798. URL: https://doi.org/10.1145/3803798.

[Pas+24] Rajesh Pasupuleti et al. "Apptainer-Based Containers for Legacy and Platform Dependent Machine Learning Applications on HPC Systems". In: *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 2024, pp. 6083–6091. DOI: 10.1109/BigData62323.2024.10825376.

[Sha] ShareGPT Community. *ShareGPT Vicuna Unfiltered Dataset*. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered. Accessed: 2026-08-16.

[Spa+25] Lorenz Sparrenberg et al. "Small and Fast LLMs on Commodity Hardware: Post-Training Quantization in llama.cpp". In: *2025 IEEE 12th International Conference on Data Science and Advanced Analytics (DSAA)* (2025). DOI: 10.1109/DSAA65442.2025.11247985.

[Vas+17] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[Wen+25] Yuhua Weng et al. "An Evaluation of the State-of-the-Art LLM Serving Frameworks". In: *Technical Report, University of Illinois Urbana-Champaign* (2025). DOI: 10.13140/RG.2.2.10209.16485. URL: https://www.researchgate.net/publication/399863135_An_Evaluation_of_the_State-of-the-Art_LLM_Serving_Frameworks_E.

[Xia+23] Guangxuan Xiao et al. "SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models". In: *Proceedings of the 40th International Conference on Machine Learning*. Vol. 202. Proceedings of Machine Learning Research. PMLR, 2023, pp. 38087–38099. URL: https://proceedings.mlr.press/v202/xiao23c.html.

[Yan+24] An Yang et al. "Qwen2.5 Technical Report". In: *arXiv preprint arXiv:2412.15115* (2024). DOI: 10.48550/arXiv.2412.15115. arXiv: 2412.15115 [cs.CL].

[YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing* (2003), pp. 44–60. DOI: 10.1007/10968987.

[Zha+23] Wayne Xin Zhao et al. "A Survey of Large Language Models". In: *arXiv preprint arXiv:2303.18223* (2023). DOI: 10.48550/arXiv.2303.18223.

[Zha+26] Wayne Xin Zhao et al. "A Survey of Large Language Models". In: *Frontiers of Computer Science* 20 (2026), p. 2012627. DOI: 10.1007/s11704-026-60308-3.

[Zhe+23] Lianmin Zheng et al. "SGLang: Efficient Execution of Structured Language Model Programs". In: *CoRR* abs/2312.07104 (2023). arXiv: 2312.07104. URL: https://arxiv.org/abs/2312.07104.

[Zho+24] Zixuan Zhou et al. “A Survey on Efficient Inference for Large Language Models”. In: *CoRR* abs/2404.14294 (2024). arXiv: 2404.14294. URL: https://arxiv.org/abs/2404.14294.

A Additional Materials

A Complete Peak GPU Memory Results

This appendix presents the complete peak GPU memory results for concurrency levels 1 and 4. The results for concurrency 16 are presented in the main evaluation chapter.

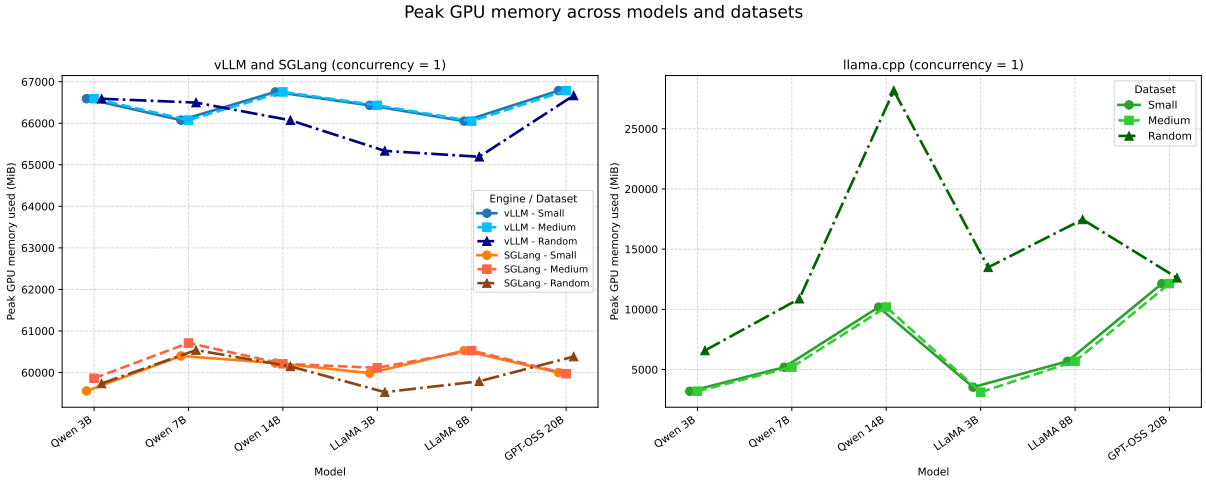

Figure 9: Peak GPU memory usage across different models and workloads at concurrency 1. The left panel shows the results for vLLM and SGLang, while the right panel shows the results for llama.cpp.

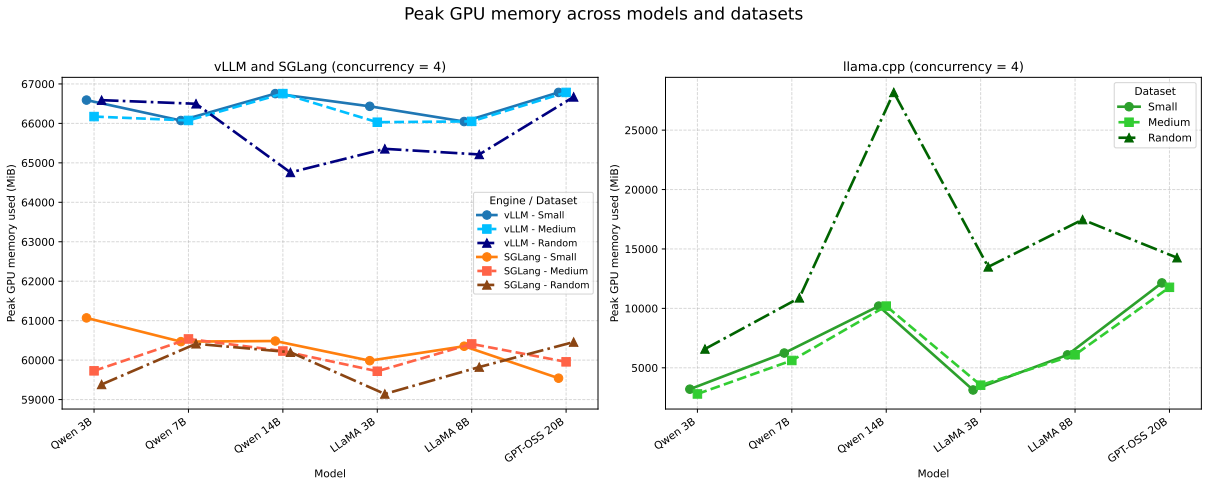

Figure 10: Peak GPU memory usage across different models and workloads at concurrency 4. The left panel shows the results for vLLM and SGLang, while the right panel shows the results for llama.cpp.

B Additional Throughput Scaling Results

This appendix presents the total-throughput scaling results under the small and medium workloads. The corresponding results for the random 20K workload are discussed in the main evaluation chapter.

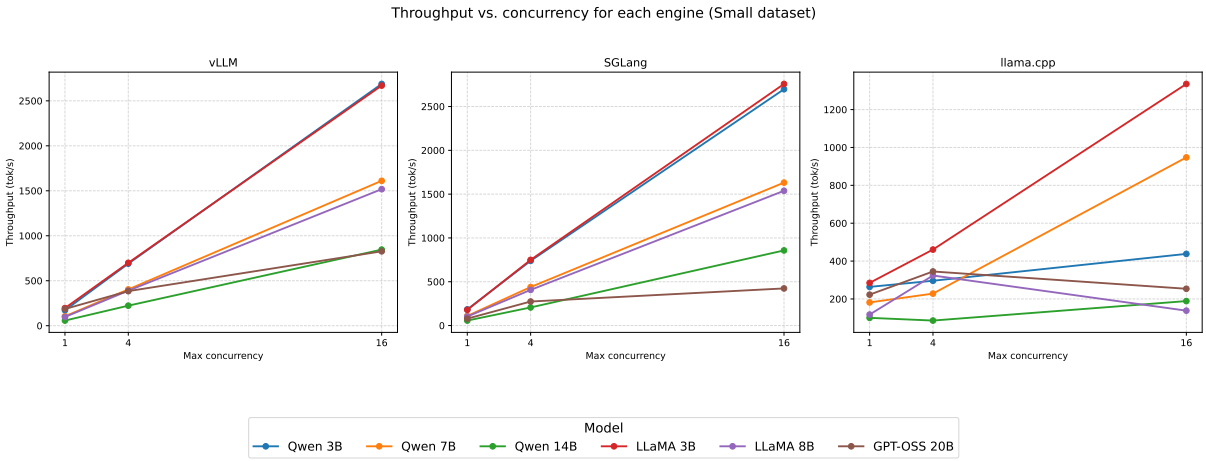

Figure 11: Total throughput as a function of maximum concurrency for vLLM, SGLang, and llama.cpp under the small workload. Each panel represents one inference engine, while the lines represent different models.

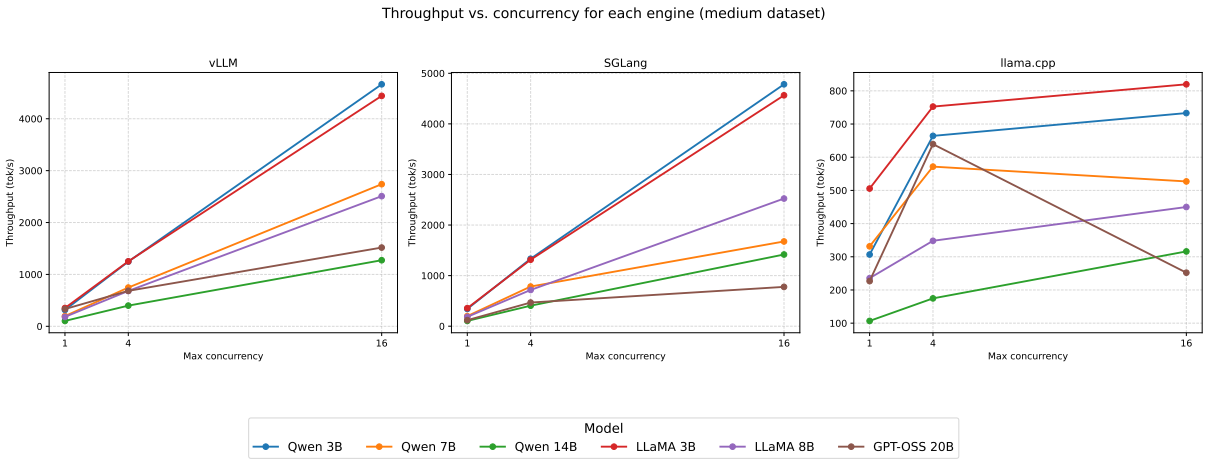

Figure 12: Total throughput as a function of maximum concurrency for vLLM, SGLang, and llama.cpp under the medium workload. Each panel represents one inference engine, while the lines represent different models.

The benchmark scripts and reproduction instructions are available at: `https://gitlab.gwdg.de/yining.liu/performance-evaluation-of-llm-inference-engines/-/tree/main`