

Seminar Report

State-of-the-Art Large Language Model Inference Engines: A Comparative Architectural and Performance Analysis

Vaishnavi Dharmeshbhai Lalwala

MatrNr: 11679855

Supervisor: Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2026

Abstract

Serving large language models (Large Language Models (LLMs)) in production is constrained less by raw compute than by Graphics Processing Unit (GPU) memory bandwidth, streaming-multiprocessor occupancy, and key-value (Key-Value (KV)) cache fragmentation, which has led the AI infrastructure ecosystem to fragment into specialised inference engines rather than converge on a single dominant design. This report addresses the practical problem of choosing among these engines: existing surveys tend to describe individual systems in isolation, which makes it difficult to see why, for example, a compiled engine outperforms a dynamically interpreted one under heavy concurrency but not at low load. To close this gap, the report undertakes a structured, desk-based comparative analysis of four representative engines – vLLM, TensorRT-LLM, llama.cpp, and Modular MAX – each embodying a distinct design philosophy, ranging from dynamic PagedAttention-based memory management to ahead-of-time compilation and unified Multi-Level Intermediate Representation (MLIR)-based graph compilation. The analysis proceeds by tracing a single inference request through the architecture of each engine and then comparing the four systems across memory management, batching and scheduling, quantization support, and multi-GPU scaling, drawing on published benchmark data collected on an NVIDIA H100 instance serving a Llama 3.1 8B model. The comparison shows that TensorRT-LLM and Modular MAX lead in raw throughput under high concurrency (15–18% above vLLM at 50 concurrent requests), vLLM leads in deployment flexibility and ecosystem maturity, and llama.cpp remains the clear choice for edge and consumer hardware, so that no single engine is best overall and the appropriate choice depends on deployment context rather than on a universal ranking.

I hereby declare that all uses of tools and AI as well as assistance from third parties in this work have been fully and accurately disclosed in the Transparency Report in Appendix Section B.

Missing or incorrect information may be considered an attempt to cheat.

Contents

List of Tables	iii
List of Figures	iii
List of Abbreviations	iv
1 Introduction	1
2 Background and Theoretical Foundation	2
2.1 Autoregressive Generation and Why Inference Is Different	2
2.2 The KV Cache and the Fragmentation Problem	2
2.3 From Static to Continuous Batching	3
2.4 Quantization as a Memory-Accuracy Trade-off	3
2.5 Compiled versus Interpreted Execution	3
3 Architectural and Performance Analysis	4
3.1 Architectural Overview of the Four Engines	4
3.2 The Inference Pipeline in Practice	5
3.3 KV Cache Management and Prefix Caching	6
3.4 Batching and Scheduling	6
3.5 Memory Management Across Model Scales	6
3.6 GPU-Level Optimisation: CUDA Graphs, FlashAttention, and Kernel Fusion	7
3.7 Quantization Support	7
3.8 Multi-GPU and Distributed Inference	8
4 Methodology	9
5 Results and Findings	9
5.1 Latency and Throughput	9
5.2 Cost per Token	10
5.3 Comprehensive Scoring Summary	10
6 Limitations and Challenges	11
7 Future Work and Implications	12
8 Conclusion	13
References	14
A Work Sharing	B1
B Transparency Report: Tools and Third-Party Contributions	B1
B.1 Overview	B1
B.2 Per Section	B2

List of Tables

1	Quantization format support across the four engines, as reported in engine documentation and vendor material [NVI26; Mod26; ggm26; vLL26]. . . .	8
2	Time-to-first-token (TTFT) latency by concurrency level (lower is better).	9
3	Peak throughput in tokens per second by concurrency level (higher is better).	9
4	Estimated cost per one million output tokens, derived from the throughput figures in table 3.	10
5	Comprehensive qualitative evaluation matrix (1–10 scale).	11

List of Figures

1	vLLM inference pipeline	4
---	-----------------------------------	---

List of Abbreviations

HPC	High-Performance Computing
LLM	Large Language Model
GPU	Graphics Processing Unit
CPU	Central Processing Unit
KV	Key-Value
HBM	High-Bandwidth Memory
SRAM	Static Random-Access Memory
SM	Streaming Multiprocessor
TTFT	Time-to-First-Token
ITL	Inter-Token Latency
MoE	Mixture-of-Experts
TP	Tensor Parallelism
PP	Pipeline Parallelism
EP	Expert Parallelism
DP	Data Parallelism
NCCL	NVIDIA Collective Communications Library
MLIR	Multi-Level Intermediate Representation
GGUF	GPT-Generated Unified Format
AOT	Ahead-of-Time
DAG	Directed Acyclic Graph
RCE	Remote Code Execution
CVE	Common Vulnerabilities and Exposures
VRAM	Video Random-Access Memory
FP8	8-bit Floating Point
FP4	4-bit Floating Point
AWQ	Activation-aware Weight Quantization
GPTQ	Generative Pre-trained Transformer Quantization

RAM Random-Access Memory

CUDA Compute Unified Device Architecture

ZMQ ZeroMQ

1 Introduction

Training a large language model (LLM) and serving it in production are, in practice, almost unrelated engineering problems. Training is dominated by gradient computation and large-batch parallelism across thousands of GPUs over weeks, whereas inference has to respond to unpredictable, one-request-at-a-time traffic, often within a latency budget measured in milliseconds, while still trying to keep expensive GPU hardware busy. This mismatch between what GPUs are good at (dense, parallel computation) and what autoregressive generation actually requires (repeatedly reloading a huge weight matrix to produce one token at a time) is the central problem that inference engines are built to solve.

As LLMs have moved from research demonstrations into production products, the cost of serving them has become a first-order concern. A poorly designed serving stack can waste a large share of GPU memory on fragmented KV caches, leave the GPU idle between requests, or force operators to over-provision hardware simply to meet latency targets. The existing response to this problem has not been a single dominant serving system, but rather a fragmentation of the ecosystem into several specialised inference engines, each of which addresses the underlying memory-bandwidth and scheduling constraints differently and, as a direct consequence, each of which is also good at a different subset of deployment scenarios and comparatively weak at others.

This report treats that fragmentation itself as the object of study. Rather than proposing a new engine, it undertakes a structured comparative analysis of four engines that are representative of the main design philosophies currently in use: vLLM (dynamic, PagedAttention-based memory management), TensorRT-LLM (Ahead-of-Time (AOT) compilation on NVIDIA hardware), llama.cpp (portability-first execution on consumer and edge hardware), and Modular MAX (a unified MLIR-based compiler that removes the Python/C++ boundary). The approach taken is to trace a single inference request through each engine’s architecture and then compare the four systems systematically across memory management, batching, quantization, and multi-GPU scaling, drawing on official engine documentation and previously published benchmark data rather than on new experiments.

The resulting comparison shows that the four engines occupy genuinely different, and largely non-overlapping, positions in the trade-off space between flexibility and raw throughput: TensorRT-LLM and Modular MAX pull ahead under heavy concurrency, vLLM remains the most practical choice wherever models or configurations change frequently, and llama.cpp is essentially unrivalled on constrained edge hardware, so that the evaluation supports a contextual recommendation rather than a single ranking.

Contributions

This report contributes the following:

- A structured architectural comparison of four inference engines that is organised around a single request’s path through the system, rather than around isolated feature lists for each engine.
- A synthesis of published throughput, latency, and cost-per-token benchmarks for these engines under a common hardware and model configuration.

- A discussion of a shared security vulnerability pattern (Remote Code Execution (RCE) via unsafe ZeroMQ (ZMQ)/pickle deserialization) that has propagated across several of these engines through code reuse, which is not usually discussed alongside performance comparisons.
- A set of context-dependent deployment recommendations rather than a single overall ranking.

Report Outline

Section 2 introduces the background concepts needed to follow the remainder of the report: autoregressive decoding, the KV cache, and the evolution of batching strategies. Section 3 forms the main analytical body, covering the architecture of each engine and comparing them across the inference pipeline, KV cache handling, batching and scheduling, memory management, GPU-level optimisation, quantization, and multi-GPU scaling. Section 4 explains the approach taken to compile the comparison, Section 5 presents the benchmark findings and cost analysis, Section 6 discusses limitations, and Section 7 considers likely future developments before Section 8 concludes.

2 Background and Theoretical Foundation

2.1 Autoregressive Generation and Why Inference Is Different

Transformer-based LLMs generate text autoregressively: each new token is predicted conditional on all previous tokens, and the model has to be run once per generated token. This creates two distinct phases of computation for a single request. The prefill phase processes the entire input prompt in one pass and is compute-bound, since a long sequence gives the GPU plenty of matrix-multiplication work to parallelise over. The decode phase that follows produces one token at a time and is memory-bandwidth bound, because generating each token requires reloading the full set of model weights (and the growing KV cache) from High-Bandwidth Memory (HBM), even though the actual arithmetic performed at each step is comparatively small. This asymmetry is the reason inference engines look so different from training frameworks: an engine that is efficient at prefill is not automatically efficient at decode, and a scheduler has to actively manage the mix of both phases across many concurrent requests.

2.2 The KV Cache and the Fragmentation Problem

To avoid recomputing attention over the entire prior context at every step, inference engines cache the key and value tensors produced for each previous token – the KV cache. The eventual length of a sequence is not known in advance, so a naive implementation has to either over-allocate memory for the worst case (internal fragmentation) or use variable-sized allocations that leave irregular unused gaps as sequences grow and shrink (external fragmentation). Kwon et al. report that this kind of fragmentation could waste well over half of the memory nominally available for the KV cache in a conventional serving system

[Kwo+23]. Because the KV cache, not the model weights, is usually what limits how many requests can be served concurrently, managing this fragmentation efficiently is one of the defining engineering problems in LLM serving.

2.3 From Static to Continuous Batching

Batching multiple requests together is the standard way of keeping a GPU busy, since a single request rarely uses the full compute capacity of a modern accelerator. Early serving systems used static batching, in which a fixed group of requests is processed together and the whole batch has to wait for its slowest (i.e. longest) member to finish before a new batch can start, leaving the GPU idle whenever request lengths vary. Continuous batching (also called in-flight batching by NVIDIA) removes this restriction by allowing the scheduler to insert a newly arrived request into an already-running batch as soon as any sequence in that batch finishes. All four engines examined in this report rely on some form of continuous or in-flight batching, although, as later sections show, the amount of scheduling sophistication built around it differs substantially.

2.4 Quantization as a Memory-Accuracy Trade-off

Because decode-phase performance is bandwidth-bound, reducing the number of bytes that have to move between HBM and the compute cores for each token is one of the most direct ways to speed up inference. Quantization achieves this by storing model weights, and sometimes activations, at lower numerical precision than the 16-bit formats typically used for training. Moving from 16-bit to 8-bit Floating Point (FP8) representations roughly halves memory traffic with a generally small effect on output quality, while 4-bit formats compress further but introduce a more noticeable accuracy cost, particularly on tasks that require careful multi-step reasoning. The four engines differ substantially in which quantization formats they support natively, which becomes an important practical constraint on deployment choices, as discussed in section 3.7.

2.5 Compiled versus Interpreted Execution

A final conceptual distinction that runs through this report is between engines that interpret a dynamic computation graph at runtime, as vLLM largely does at the orchestration level, and engines that compile the model into a fixed, optimised execution plan ahead of time, as TensorRT-LLM does, and as Modular MAX does via MLIR. Interpreted execution is more flexible, since a new model architecture can usually be run without engineering changes, but it introduces Central Processing Unit (CPU)-side overhead from Python’s scheduling and kernel-launch logic. Compiled execution removes much of that overhead through operator fusion and pre-planned memory layouts, at the cost of a compilation step that must be repeated whenever the model, batch configuration, or hardware target changes. This trade-off between flexibility and peak efficiency recurs throughout the comparison that follows.

3 Architectural and Performance Analysis

3.1 Architectural Overview of the Four Engines

vLLM, developed originally at UC Berkeley’s Sky Computing Lab, is built around an asynchronous Engine Core that separates the logical scheduling of sequences from the physical layout of GPU memory [Kwo+23]. Its defining contribution is PagedAttention, which divides the KV cache into fixed-size physical blocks – 16 tokens by default – and maps each sequence’s logical tokens onto these blocks through a per-sequence block table, in much the same way an operating system maps virtual memory pages onto physical Random-Access Memory (RAM). A KV Cache Manager tracks free blocks through a doubly linked list, allocating and releasing them as sequences grow and finish. Because this scheduling logic lives largely in Python, vLLM captures Compute Unified Device Architecture (CUDA) graphs during a warm-up phase so that the repetitive decode step can later be replayed as a single kernel launch rather than incurring per-token Python overhead. Figure 1 sketches this pipeline schematically.

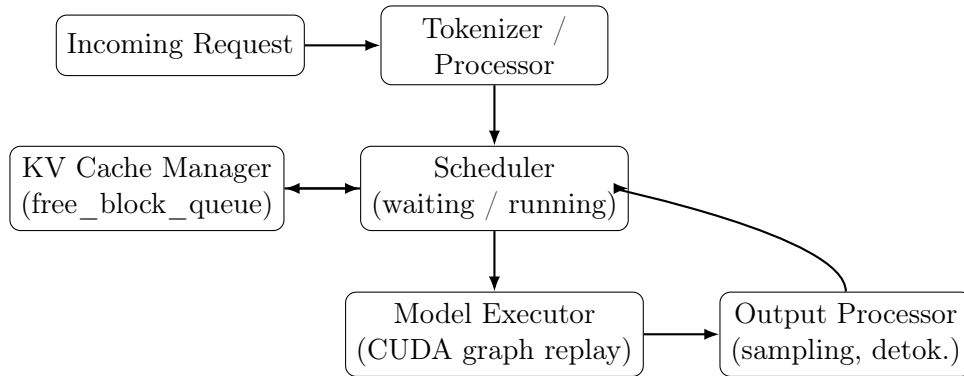


Figure 1: Simplified view of vLLM’s Engine Core: the scheduler mediates between the paged KV cache manager and the model executor, and a finished sequence’s freed blocks feed back into the scheduler for the next iteration.

TensorRT-LLM takes an almost opposite approach. Rather than interpreting a graph at runtime, it compiles the model ahead of time into a fused CUDA engine using NVIDIA’s proprietary TensorRT compiler [NVI26]. This offline compilation step fuses operations such as matrix multiplications, activation functions, and layer normalisation into single, arithmetically dense kernels, and hard-codes the engine for specific maximum batch sizes, sequence lengths, and numerical precisions. At runtime, a C++ executor drives in-flight batching and manages a dual-paged KV cache capable of handling both self-attention and cross-attention caches for encoder–decoder models. The practical consequence is that any change to the model, the target batch size, or the parallelism configuration requires recompiling the engine, which can take a substantial amount of time.

llama.cpp follows a very different set of priorities again: portability and minimal dependencies rather than data-centre throughput [ggm26]. It is built on the C-based GGML tensor library and stores models in the GPT-Generated Unified Format (GGUF) format, which bundles the quantized weights, tokenizer vocabulary, and hyperparameters into a single memory-mappable file. Using `mmap` to load weights lets the operating system

page data in directly from disk as needed, keeping host RAM usage low. A backend abstraction layer lets the same computational graph run on CPU (using AVX2, AVX-512, or ARM NEON instructions) or be offloaded to CUDA, Metal, or Vulkan, but the scheduler itself is comparatively simple, handling requests sequentially or through static batching rather than the continuous batching used by the other three engines.

Modular MAX represents the newest and most structurally distinct approach [Mod26]. It compiles an entire computation graph – from the request-handling code down to hardware-specific kernels – through the MLIR framework, using Mojo, a systems programming superset of Python, to write custom operators. Because the compiler can see across what would normally be a Python/C++ boundary, it can fuse operations, for instance rotary positional embeddings, self-attention, and the following projection layer, into a single kernel pass, avoiding the need to write large intermediate tensors back to HBM. MAX also performs topology-aware placement on bare-metal, hypervisor-free systems, querying the physical NVLink layout so that tensor-parallel communication can be routed over the fastest available hardware paths.

3.2 The Inference Pipeline in Practice

Following a single request through each engine illustrates how these architectural choices play out. All four systems begin with tokenization: vLLM wraps HuggingFace tokenizers inside a Python-level processor, TensorRT-LLM uses a multithreaded C++ tokenizer, llama.cpp embeds tokenization directly in the GGUF file, and MAX wraps the tokenizer into its native Python serving loop. The prefill phase then diverges more noticeably: vLLM applies chunked prefill, splitting a very long prompt into smaller segments processed across multiple scheduler iterations so that a single enormous prompt cannot stall ongoing decode work for other users. TensorRT-LLM instead computes the full prompt in one pass using fused FlashAttention kernels [Dao+22], relying on its in-flight batching manager to balance this against concurrent decode steps. MAX processes the prefill inside its compiled MLIR graph, keeping intermediate activations in GPU registers rather than writing them to HBM, while llama.cpp falls back on its CPU or GPU-offloaded quantized kernels, which scale less efficiently once concurrency rises.

Once prefill completes, the resulting keys and values have to be written into the KV cache. vLLM computes the number of physical blocks required, rounding up to the nearest 16-token block, and pulls them from its free-block pool; TensorRT-LLM writes into pre-allocated paged memory pools; and MAX uses a similar paging structure to vLLM’s but implements the block bookkeeping inside compiled Mojo code rather than in Python dictionaries, which removes a layer of interpreter overhead. The decode phase that follows is memory-bandwidth bound for all engines, which is precisely why CUDA graph replay (vLLM), a tight compiled C++ loop (TensorRT-LLM), and a pre-compiled loop graph (MAX) all exist for the same underlying reason: to avoid re-paying Python or generic launch overhead on every single token. Sampling – applying temperature, top- p and top- k , checking stop conditions, and freeing the KV cache once a sequence ends – is performed on the host in vLLM’s case and largely fused onto the GPU by TensorRT-LLM and MAX, which reduces host–device transfer for these final steps.

3.3 KV Cache Management and Prefix Caching

The clearest technical dividing line in this comparison is between llama.cpp’s segmented allocation strategy and the paged approach used by the other three engines. Because llama.cpp allocates roughly contiguous cache space, variable-length requests inevitably leave unusable gaps as sequences of different lengths are created and freed – external fragmentation in the classic operating-systems sense. vLLM, TensorRT-LLM, and MAX all instead map logical token positions onto scattered, fixed-size physical blocks, which mathematically eliminates external fragmentation and confines waste to, at most, one partially filled block per sequence.

vLLM extends this idea further with automatic prefix caching. Incoming prompts are split into 16-token chunks, each of which is hashed; if an identical block has already been computed for another request – a shared system prompt is the obvious example – the engine simply increments a reference count on the existing physical block rather than recomputing or re-storing it. This is a genuinely useful optimisation for applications that repeatedly send the same instructions or few-shot examples with a changing final query, since it can skip both the compute and the memory allocation for the shared portion entirely. MAX supports a broadly similar prefix-caching mechanism within its own serving framework. Under memory pressure – extremely long contexts or very high concurrency – vLLM’s scheduler falls back to a preemption policy that either swaps lower-priority KV blocks out to host memory over PCIe or discards and later recomputes them, whereas TensorRT-LLM and MAX handle very long contexts by sharding the KV cache itself across multiple GPUs using tensor parallelism, which sidesteps the problem rather than managing it dynamically on a single device.

3.4 Batching and Scheduling

As outlined in section 2, continuous (or in-flight) batching is what allows a scheduler to swap a finished sequence out for a new one without waiting for the whole batch to complete. vLLM implements this through two queues, waiting and running, with its V1 scheduler mixing decode and prefill work within a single step. Active decode requests are prioritised: the scheduler works out how many tokens they need to generate, allocates the corresponding cache slots, and deducts this from a global per-step token budget, with only the leftover budget going to new prefill work, which is chunked if it does not fit. TensorRT-LLM separates these concerns further, asynchronously managing encoder-phase and decoder-phase batch sizes independently within its decoupled in-flight batching design. This kind of scheduling matters because throughput and latency pull in different directions: maximising throughput generally means packing the largest batch that Video Random-Access Memory (VRAM) allows, while minimising Time-to-First-Token (TTFT) requires making sure a handful of very large prompts cannot monopolise the schedule and delay everyone else’s next token.

3.5 Memory Management Across Model Scales

How much these architectural differences matter depends heavily on model size. For an 8-billion-parameter model, VRAM requirements are modest, in the order of 8–10 GB, and the workload remains largely compute-bound, which is exactly the regime in which MAX’s graph-level fusion is reported to be most advantageous, since operations can be

kept close to the GPU’s L2 cache rather than repeatedly touching HBM. At the 70-billion-parameter scale, the picture shifts decisively into memory-bandwidth-bound territory, and tensor parallelism across multiple GPUs becomes effectively mandatory; here, vLLM’s paged KV cache and TensorRT-LLM’s tuned NVIDIA Collective Communications Library (NCCL) communication routines are both aimed squarely at sustaining high GPU occupancy despite this constraint. Serving a 70B model at FP8 on an 80 GB H100 leaves very little headroom: vLLM was reported to consume around 71 GB, SGLang 72 GB, and TensorRT-LLM 74 GB, with the gap attributed to TensorRT-LLM’s larger persistent activation buffers, which support its fused kernels at the cost of leaving less room for the KV cache [Sph26c]. Beyond 70B, and particularly for very large Mixture-of-Experts (MoE) models such as DeepSeek V3, dynamic expert routing introduces a further layer of memory fragmentation that none of these engines has fully solved; TensorRT-LLM’s Expert Parallelism (EP) implementation is presented as the most mature response to this specific problem, while MAX’s graph compiler is described as still maturing in this area.

3.6 GPU-Level Optimisation: CUDA Graphs, FlashAttention, and Kernel Fusion

Because decoding involves many small, repetitive kernel launches, the overhead of a Python-based runtime, for example Global Interpreter Lock contention and general scheduling latency, can, at low batch sizes, exceed the time actually spent computing on the GPU. vLLM addresses this by capturing CUDA graphs during a warm-up phase, effectively baking the sequence of GPU operations for a given batch size into a single replayable unit so that only one launch command is issued per decode step. FlashAttention, developed independently of any single engine, is used across the ecosystem [Dao+22] to compute attention exactly while tiling the query, key, and value blocks inside the GPU’s fast on-chip Static Random-Access Memory (SRAM) rather than repeatedly reading and writing the full attention matrix to HBM; TensorRT-LLM builds on this with additional hardware-specific optimisation for NVIDIA’s tensor cores. Kernel fusion, merging multiple operations into a single GPU kernel to avoid materialising intermediate results in HBM, is where TensorRT-LLM and MAX diverge most clearly from vLLM: because both compile ahead of time, they can fuse across a wider span of the computation graph, and MAX in particular is described as fusing positional embeddings, attention, and the subsequent projection layer into a single pass, precisely because its compiler is not constrained by a Python/CUDA boundary in the way a dynamically interpreted framework is.

3.7 Quantization Support

Quantization support differs enough across the four engines that it becomes a practical deciding factor in its own right. Table 1 summarises the picture presented across the reviewed documentation.

Two broad patterns emerge from table 1. First, TensorRT-LLM is consistently positioned at the leading edge of NVIDIA-specific numerical formats, including early native support for the 4-bit Floating Point (FP4) format expected on Blackwell-generation hardware. Second, llama.cpp occupies a genuinely separate niche: its GGUF k-quant formats are described as the de facto standard for edge and consumer hardware, but the same design that makes them efficient on CPUs and consumer GPUs makes it comparatively difficult to write custom matrix-multiplication kernels that fully exploit data-centre tensor

Table 1: Quantization format support across the four engines, as reported in engine documentation and vendor material [NVI26; Mod26; ggm26; vLL26].

Format	Precision	vLLM	TensorRT-LLM	llama.cpp	Modular MAX
FP32 / FP16 / BF16	32/16-bit	Native	Native	Native	Native
FP8	8-bit float	Excellent	Native / optimised	Lacking	Built-in
INT8 / SmoothQuant	8-bit int	Good	Excellent	Good	Good
INT4 / AWQ / GPTQ	4-bit	Excellent	Excellent	No	Good
GGUF / k-quants	Mixed (2–8 bit)	Experimental	No	Industry standard	Partial
FP4 (Blackwell B200)	4-bit float	Experimental	Native	No	Built-in (upcoming)

cores, which is one reason llama.cpp is not positioned as a hyperscale serving solution. Moving from 16-bit to 8-bit precision is generally reported to cost very little in output quality, whereas 4-bit formats trade a larger, more noticeable amount of accuracy for a roughly 75% reduction in memory footprint, a trade-off that matters more for tasks requiring extended reasoning than for simpler generation tasks.

3.8 Multi-GPU and Distributed Inference

Scaling beyond a single GPU introduces communication as a first-class bottleneck alongside compute and memory bandwidth. Data Parallelism (DP), where the whole model is simply replicated across GPUs or nodes, requires no special engine support and is handled by ordinary load balancing. Tensor Parallelism (TP), which shards individual weight matrices, attention heads for instance, across GPUs within a node, is far more communication-intensive, since every forward pass requires an all-reduce synchronisation step and therefore depends heavily on fast intra-node interconnects such as NVLink. vLLM coordinates tensor parallelism through Ray or a custom multiprocessing backend, while TensorRT-LLM and MAX rely on tightly tuned NCCL communication graphs; TensorRT-LLM additionally supports Pipeline Parallelism (PP), splitting the model’s layers sequentially across GPUs or nodes for models too large to fit in a single node’s collective VRAM, and EP, which routes individual tokens to whichever node hosts the relevant MoE expert weights. Where vLLM’s reliance on Ray introduces a modest amount of process-serialisation overhead, TensorRT-LLM compiles down to a standalone MPI/C++ binary that reduces inter-node latency but demands that the parallelism topology be fixed before compilation, another instance of the same flexibility-versus-throughput trade-off seen throughout this report. MAX’s contribution here is topology awareness: on bare-metal, hypervisor-free deployments it can query the physical NVLink layout directly and route tensor-parallel communication accordingly, an optimisation that is only meaningful outside virtualised cloud environments where such low-level hardware visibility is available.

4 Methodology

This report is a desk-based comparative analysis rather than an original benchmarking exercise. The comparison draws on official engine documentation [NVI26; Mod26; vLL26], the GGML/llama.cpp project repository [ggm26], and previously published third-party benchmark results [Sph26c; Sph26b; Sph26a], most notably a set of throughput and latency measurements collected on a bare-metal NVIDIA H100 SXM5 80GB instance serving a Llama 3.1 8B model at FP8 precision. No new experiments, model deployments, or hardware tests were carried out as part of preparing this report; all quantitative figures reported in sections 3 and 5 are taken directly from the sources cited alongside them and should be read as representative of the specific hardware, model, and concurrency settings under which they were originally measured, rather than as universal constants.

The four engines were selected because they represent structurally distinct approaches to the same underlying problem, namely dynamic interpretation (vLLM), AOT compilation on proprietary hardware (TensorRT-LLM), portability-first CPU/edge execution (llama.cpp), and a newer unified-compiler approach (Modular MAX), rather than being an exhaustive survey of every available serving framework. Other systems mentioned in passing, such as SGLang, appear only where the underlying source material includes them for benchmark comparison and are not analysed architecturally in their own right.

5 Results and Findings

5.1 Latency and Throughput

The benchmark figures compiled in tables 2 and 3 reflect tests run on a Llama 3.1 8B model at FP8 precision under varying concurrency levels on the H100 platform described above [Sph26c].

Table 2: Time-to-first-token (TTFT) latency by concurrency level (lower is better).

Metric (concurrency)	vLLM	SGLang	TensorRT-LLM	Modular MAX
p50 TTFT (1 request)	45 ms	38 ms	42 ms	<38 ms
p50 TTFT (10 requests)	120 ms	112 ms	105 ms	105 ms
p95 TTFT (100 requests)	1,450 ms	1,380 ms	1,280 ms	–

Table 3: Peak throughput in tokens per second by concurrency level (higher is better).

Concurrent requests	vLLM	SGLang	TensorRT-LLM	Modular MAX
1 (low load)	110 tok/s	115 tok/s	120 tok/s	125 tok/s
50 (high load)	1,850 tok/s	1,920 tok/s	2,100 tok/s	2,150 tok/s

Two things stand out from these figures. At a single request, the gap between engines is fairly small, around 8% between the slowest and fastest, because the GPU has ample

spare compute headroom regardless of which engine is issuing the work. Under heavier concurrency (50 simultaneous requests), the gap widens considerably: TensorRT-LLM and MAX pull ahead of vLLM by roughly 15–18%, which is consistent with the architectural discussion in section 3, since AOT compilation and deeper kernel fusion pay off precisely when the GPU is saturated and every avoided memory round-trip counts. It is worth noting, however, that these are single-model, single-hardware measurements; vLLM’s more efficient KV cache footprint is reported to allow a higher maximum context-length ceiling at the 70B scale, meaning the throughput advantage seen here for an 8B model does not automatically generalise to larger models or different concurrency patterns.

5.2 Cost per Token

Translating throughput into cost, using a representative bare-metal H100 SXM5 80GB spot-instance rate of \$1.69 per hour against the 50-request throughput figures above, gives the picture shown in table 4.

Table 4: Estimated cost per one million output tokens, derived from the throughput figures in table 3.

Engine	Peak throughput (50 req)	Cost / 1M tokens	Operational complexity	
vLLM	1,850 tok/s	\$0.25	Low (agile deployment)	
TensorRT-LLM	2,100 tok/s	\$0.22	High (extensive compilation)	
Modular MAX	2,150 tok/s	\$0.22	Medium (cold-start penalty)	

The roughly 12–15% cost reduction implied by moving from vLLM to TensorRT-LLM or MAX is meaningful at scale – an organisation generating tens of billions of tokens a month would plausibly see savings in the thousands of dollars – but this has to be weighed against the operational cost of getting there. TensorRT-LLM’s advantage comes bundled with a compilation step that has to be repeated for meaningful model or configuration changes, and MAX’s comes with a first-run compilation penalty reported at around eight minutes on an H100, dropping to roughly 65 seconds on subsequent cached starts. In other words, the cheaper cost-per-token figures are not free; they are paid for in engineering time and deployment rigidity, which is why vLLM remains the more sensible option for teams whose models or configurations change frequently even though it is not the cheapest option per token generated.

5.3 Comprehensive Scoring Summary

Table 5 presents a qualitative scoring exercise across eleven evaluation categories, rated out of ten. These scores should be read as informed judgments synthesised from the documentation and benchmarks discussed above, rather than as outputs of a formal, independently validated methodology; they are included because they usefully summarise the overall shape of the comparison.

The pattern in table 5 is broadly consistent with the architectural analysis in section 3: vLLM leads on deployment ease and ecosystem maturity, which is exactly what

Table 5: Comprehensive qualitative evaluation matrix (1–10 scale).

Category	vLLM	TensorRT-LLM	llama.cpp	Modular MAX
Throughput (max concurrency)	8	9.5	4	10
Latency (p50 TTFT)	8.5	10	7	10
Memory efficiency (paged)	9.5	9	5	9
Quantization support	9	10	10 (GGUF)	8
Ease of deployment	10	5	8	7
Multi-GPU / TP scalability	9	10	4	9
HPC bare-metal readiness	8	9	6	10
Enterprise readiness (sec/ops)	9	10	4	7
Community and ecosystem	10	8	9.5	5
Cost efficiency per token	8.5	9.5	N/A	9.5
Future potential architecture	8	8	7	10

one would expect from a system built around dynamic, HuggingFace-compatible execution; TensorRT-LLM leads on latency, quantization, and enterprise security hardening, consistent with its position inside NVIDIA’s tightly controlled and heavily audited software stack; llama.cpp trails badly on throughput and multi-GPU scaling but leads decisively on GGUF quantization support, reflecting its deliberate focus on single-device and edge use rather than data-centre serving; and MAX scores highest on raw throughput, High-Performance Computing (HPC) readiness, and future architectural potential, while scoring lowest on community and ecosystem maturity, which is unsurprising for a system still in the early stages of adoption.

6 Limitations and Challenges

Several limitations apply to the engines themselves, and a further set apply to this report as a comparison of them. On the engine side, TensorRT-LLM’s central strength, AOT compilation, is simultaneously its most significant weakness: any change to model weights, batch-size limits, or parallelism configuration requires a fresh, time-consuming compilation cycle, and its closed-source compiler limits the extent to which the community can extend or audit it. llama.cpp’s portability comes at the cost of a comparatively simple scheduler that lacks the dynamic memory pooling and continuous batching sophistication needed to saturate a data-centre GPU, which is precisely why it is not recommended for high-concurrency enterprise endpoints despite being extremely well suited to constrained hardware. Modular MAX, being the newest entrant among the four, currently lacks native multi-LoRA serving and mature support for the complex, dynamic routing required by

large MoE models, and its eight-minute cold-start compilation penalty is a real operational cost even though it largely disappears on cached restarts.

A further consideration, which the source material for this report treats as a shared industry issue rather than a flaw specific to any one engine, concerns security. Oligo Security’s ShadowMQ research documented a recurring pattern of unsafe deserialization, combining ZMQ’s `recv_pyobj()` method with Python’s pickle module over unauthenticated network sockets, that had propagated by code reuse across several major inference frameworks, including vLLM (Common Vulnerabilities and Exposures (CVE)-2025-30165), NVIDIA TensorRT-LLM (CVE-2025-23254), and Modular MAX Server (CVE-2025-60455) [Oli25; The25]. All three vendors have since patched the specific vulnerabilities identified, but the fact that structurally identical unsafe code was independently copied into multiple, competing projects suggests that the broader pattern, treating internal distributed-communication channels as implicitly trusted, may not be fully resolved by patching individual instances of it, and is a reasonable ongoing concern for anyone deploying these systems in multi-tenant or internet-facing environments.

As for the limitations of this report itself: the quantitative benchmark and cost figures in section 5 come from a single model (Llama 3.1 8B, FP8) on a single hardware configuration (H100 SXM5 80GB), and while the underlying architectural reasons for the observed differences are likely to generalise, the exact percentage gaps almost certainly would not hold for a 70B or MoE-scale model, for different GPUs, or for different request-length distributions. The qualitative scoring matrix in table 5, while a useful summary, reflects subjective judgment on how to weight categories such as “future potential architecture” against more measurable ones such as latency, and a different set of assumptions could reasonably produce somewhat different scores. Finally, this is a fast-moving area of software engineering: cold-start times, supported quantization formats, and even security postures are the kind of thing that can change within months, so the comparison should be read as a snapshot of the ecosystem at the time the underlying material was gathered rather than a permanent ranking.

7 Future Work and Implications

Several trajectories seem reasonably likely to shape how this comparison looks in a few years’ time. If Modular MAX’s MLIR-based compiler matures to properly support dynamic MoE routing and multi-LoRA serving, its throughput advantage on dense models could plausibly extend to the workloads where TensorRT-LLM currently has the clearest lead, particularly given that MAX’s compiler is designed to target CUDA, ROCm, and Metal from a single codebase rather than being locked to one vendor’s hardware. Whether this materialises depends on execution and adoption as much as on the technical design, and should be treated as a plausible direction rather than a foregone conclusion.

On the NVIDIA side, the arrival of Blackwell-generation hardware and native FP4 tensor cores is likely to reinforce TensorRT-LLM’s position in fixed, hyperscale production deployments, since its compilation-based approach is well suited to extracting the last percentage points of throughput from a known, unchanging model and hardware target. For research and rapidly iterating deployments, vLLM’s advantages in ecosystem compatibility and deployment simplicity are less easily displaced by a compiled competitor, because the value it provides, being able to try a new checkpoint or architecture within minutes, is orthogonal to raw throughput. At the edge, llama.cpp’s GGUF format

and broad CPU fallback support appear likely to remain the standard for on-device and consumer-hardware inference, an area where the other three engines are not seriously positioned to compete.

The shared ShadowMQ vulnerability pattern discussed in section 6 also points to a broader implication worth flagging: as inference engines increasingly borrow code and design patterns from one another, security practices such as authenticating internal communication channels and avoiding unsafe deserialization need to be treated as ecosystem-wide concerns rather than something each project can address in isolation after the fact.

8 Conclusion

This report set out to compare four structurally different LLM inference engines and to identify which is the more sensible choice under which circumstances, rather than to declare an overall winner. The answer that emerges from the architectural and performance evidence gathered here is that the four engines are, in effect, optimised for different points on the same underlying trade-off between flexibility and raw efficiency. vLLM’s dynamic, PagedAttention-based design makes it the most practical choice for research settings, start-ups, and any environment where models and configurations change often, at a modest cost in absolute throughput. TensorRT-LLM’s AOT compilation extracts the most throughput and the lowest latency from a fixed model at hyperscale, but only once an organisation is willing to accept the operational overhead of maintaining compiled engines. llama.cpp remains the clear choice wherever the deployment target is a laptop, a phone, or otherwise resource-constrained hardware rather than a GPU cluster. Modular MAX, as the newest of the four, currently shows the strongest raw throughput numbers on dense models and an architecturally interesting path toward hardware-agnostic compilation, but its practical adoption is still limited by ecosystem maturity, particularly around MoE routing and multi-LoRA serving. Taken together, the comparison suggests that the choice of inference engine should follow from the specific deployment context, model size, hardware environment, and how often the underlying model or configuration is expected to change, rather than from a single, universally best-performing system.

References

- [Dao+22] Tri Dao et al. “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness”. In: *Advances in Neural Information Processing Systems*. Vol. 35. 2022. URL: <https://arxiv.org/abs/2205.14135> (visited on 09/30/2026).
- [ggml26] ggml-org. *llama.cpp*. 2026. URL: <https://github.com/ggml-org/llama.cpp> (visited on 09/30/2026).
- [Kwo+23] Woosuk Kwon et al. “Efficient Memory Management for Large Language Model Serving with PagedAttention”. In: *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP ’23)*. 2023. URL: <https://arxiv.org/abs/2309.06180> (visited on 09/30/2026).
- [Mod26] Modular Inc. *MAX Graph Compiler and Mojo Documentation*. 2026. URL: <https://docs.modular.com/> (visited on 09/30/2026).
- [NVI26] NVIDIA Corporation. *TensorRT-LLM Documentation*. 2026. URL: <https://nvidia.github.io/TensorRT-LLM/> (visited on 09/30/2026).
- [Oli25] Oligo Security. *ShadowMQ: How Code Reuse Spread Critical Vulnerabilities Across the AI Ecosystem*. 2025. URL: <https://www.oligo.security/blog/shadowmq-how-code-reuse-spread-critical-vulnerabilities-across-the-ai-ecosystem> (visited on 09/30/2026).
- [Sph26a] Spheron Network. *Modular MAX and Mojo on GPU Cloud: Deploy an LLM Inference Engine*. 2026. URL: <https://spheron.network/> (visited on 09/30/2026).
- [Sph26b] Spheron Network. *TensorRT-LLM Production Deployment on GPU Cloud: Engine Build, Multi-GPU Serving, and In-Flight Batching (2026)*. 2026. URL: <https://spheron.network/> (visited on 09/30/2026).
- [Sph26c] Spheron Network. *vLLM vs TensorRT-LLM vs SGLang: H100 Benchmarks (2026)*. 2026. URL: <https://spheron.network/> (visited on 09/30/2026).
- [The25] The Hacker News. *Researchers Find Serious AI Bugs Tied to ZeroMQ and Pickle Deserialization Across Inference Engines*. Nov. 2025. URL: <https://thehackernews.com/2025/11/researchers-find-serious-ai-bugs.html> (visited on 09/30/2026).
- [vLL26] vLLM Project. *vLLM Documentation: Performance and Tuning; Optimization and Tuning*. 2026. URL: <https://docs.vllm.ai/> (visited on 09/30/2026).

A Work Sharing

This report was prepared entirely by a single author (Vaishnavi Dharmeshbhai Lalwala). No group work division applies. The author was responsible for the research, analysis, data collection, interpretation, comparison of the four engines, evaluation criteria, ratings, conclusions, and preparation of the original slide deck and report content.

B Transparency Report: Tools and Third-Party Contributions

B.1 Overview

The research, analysis, arguments, data, findings, and conclusions presented in this report are entirely the author’s own work. The author independently selected the four engines, researched and analyzed their architectures, collected and interpreted benchmark and cost data, developed the comparison criteria and qualitative ratings, and formulated the conclusions. The report was originally developed as a slide deck by the author and subsequently prepared in report form by the author.

Generative AI was used only as a supporting tool for limited technical and editorial tasks, such as checking wording, assisting with LaTeX formatting, generating TikZ code for a figure based on the author’s own architecture description, and helping locate potentially relevant sources. The substantive content and intellectual work of the report were produced by the author. All AI-assisted output was reviewed, corrected where necessary, and approved by the author before inclusion.

- Writing
 - The report was drafted and written by the author.
 - L^AT_EX was used for typesetting, based on a report template provided by the course/institute.
 - AI assistance was used only for limited language polishing and formatting support where appropriate. It did not generate the underlying research, arguments, analysis, data, or conclusions.
- Diagrams
 - The vLLM pipeline diagram (fig. 1) was designed by the author based on the architecture described and analyzed in the author’s work.
 - AI assistance was used to generate the corresponding TikZ code. The author reviewed and adapted the generated code and is responsible for the final figure.
- Tables
 - All table data, comparisons, and qualitative ratings were researched, selected, and determined by the author.

- AI assistance was used only for table formatting and LaTeX typesetting.
- Code
 - No substantive software code or program listings are part of this report.
 - The only generated code was TikZ code used for the report figure, as described above.
- AI
 - AI was used as a supporting tool for limited formatting, language refinement, and technical typesetting tasks.
 - AI was also used to help identify potentially relevant citations, including academic papers, vendor documentation, and the ShadowMQ CVE disclosure. The author independently checked the cited information against the original sources before including it in the report.
 - AI did not perform the research, determine the comparison criteria, produce the benchmark or cost analysis, assign the qualitative ratings, develop the arguments, or formulate the conclusions.
- Third-Party
 - No third-party human contributions were made to the research, analysis, writing, or conclusions of this report.

B.2 Per Section

All sections of the report were researched, analyzed, and drafted by the author. The arguments, data, comparisons, findings, and conclusions originate from the author’s own work. AI assistance was limited to specific supporting tasks as follows:

- Section 1 Introduction – drafted and written by the author; no substantive AI contribution.
- Section 2 Background and Theoretical Foundation – drafted and written by the author; AI assistance was limited to locating potentially relevant references to the PagedAttention and FlashAttention papers, which were subsequently checked by the author.
- Section 3 Architectural and Performance Analysis – research, analysis, and content were completed by the author. AI assistance was used for generating TikZ code for the vLLM pipeline figure and for formatting the quantization table.
- Section 4 Methodology – drafted and written by the author; no substantive AI contribution.
- Section 5 Results and Findings – benchmark and cost data, interpretation, comparisons, and scoring were determined by the author. AI assistance was limited to formatting the corresponding tables.

- Section 6 Limitations and Challenges – analysis and discussion were completed by the author. AI assistance was used to help verify the ShadowMQ CVE identifiers against publicly available sources.
- Section 7 Future Work and Implications – drafted and written by the author; no substantive AI contribution.
- Section 8 Conclusion – drafted and written by the author; no substantive AI contribution.

Overall, the author retains full responsibility for the content, research, analysis, interpretation, and conclusions of this report. AI tools were used only as supplementary aids for selected technical, formatting, and verification tasks.