Journal of Computer Science and Technology Studies
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

JCSTS
AL-KINDI CENTER FOR RESEARCH AND DEVELOPMENT

| RESEARCH ARTICLE

Demystifying LLM Serving Pipeline: From Prompt to Response

Reeshav Kumar
Independent Researcher, USA
Corresponding Author: Reeshav Kumar, **E-mail**: reachreeshav@gmail.com

| ABSTRACT

Each response from an LLM application follows a carefully optimized sequence of steps designed to balance quality, latency, and cost efficiency. This article outlines a typical LLM serving pipeline, beginning with user prompt capture, retrieval augmentation, tokenization, request routing, followed by auto-regressive token generation and post-processing to produce the final response. We evaluate critical system elements in the LLM serving pipeline, including client interfaces, policy verification mechanisms, admission control systems, KV-cache management, speculative decoding techniques, and post-processing operations. The article also examines the trade-offs among latency and throughput, memory and compute efficiency, and concurrency and response time that system architects and product leaders must balance to develop robust LLM applications.

| KEYWORDS

Inference Optimization, Key-Value Cache, Speculative Decoding, Retrieval-Augmented Generation, Dynamic Batching

| ARTICLE INFORMATION

ACCEPTED: 12 November 2025 **PUBLISHED:** 02 December 2025 **DOI:** 10.32996/jcsts.2025.7.12.37

1. Introduction

Large Language Models (LLMs) may seem to operate instantaneously, but each response is meticulously guided through a multi-stage pipeline. This pipeline, designed with utmost precision, ensures a delicate balance of quality, latency, and cost efficiency. This article delves into the comprehensive system architecture that powers LLM inference, detailing each stage from initial prompt capture to final token generation.

The complexity behind modern LLM serving systems is a testament to the strategic decisions made by system architects. It represents a fascinating intersection of distributed systems engineering, hardware acceleration, and algorithmic optimization. While users experience these systems as responsive conversational interfaces, the underlying infrastructure implements sophisticated techniques to manage computational resources efficiently. Recent research reveals that web-based LLM deployments face unique optimization challenges that require careful balancing of multiple competing factors across the entire serving pipeline [1]. Similarly, a comprehensive analysis of distributed inference architectures demonstrates how system-level design decisions significantly impact both performance and scalability characteristics in production environments [2].

This paper will discuss the entire journey of the LLM serving process, starting with the point at which a user enters a prompt and ending with the point at which a response is displayed back on the user's screen. Engineers and technical leaders can make informed choices about deployment architectures, system optimization, and performance tuning due to their in-depth technical knowledge of these systems. The serving pipeline is made up of multiple steps: client capture and prompt assembly, tokenization and policy validation, request routing and hardware acceleration, retrieval augmentation, admission control and batch processing, and, last but not least, the decoder loop, in which generation is performed. The system's designers will need to make

severe trade-offs among competing priorities at each stage, including throughput, latency, memory efficiency, and response quality.

As the use of LLMs in organizations becomes an increasingly integral component of their technical infrastructure, understanding these architectural patterns becomes vital. The LLM serving pipeline, with its significant impact on system performance, is a crucial element in the process of creating systems that deliver predictable performance and effectively utilize available resources. The following sections discuss each element of the LLM serving pipeline individually, considering the technical issues and optimization opportunities at each level of the inference process.

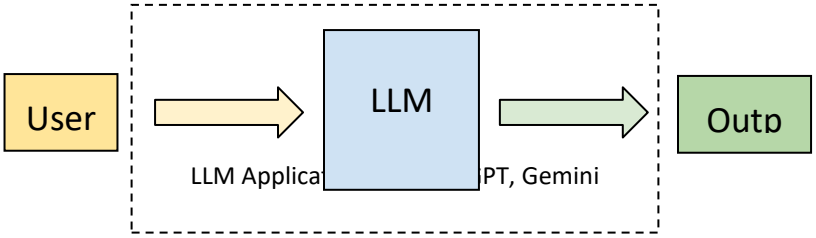

Fig 1: LLM Inference [1, 2]

2. The Serving Path: A System-Level Overview

As a user makes a prompt to an LLM system, the prompt triggers a complex sequence of processes that have different performance properties and optimization problems.

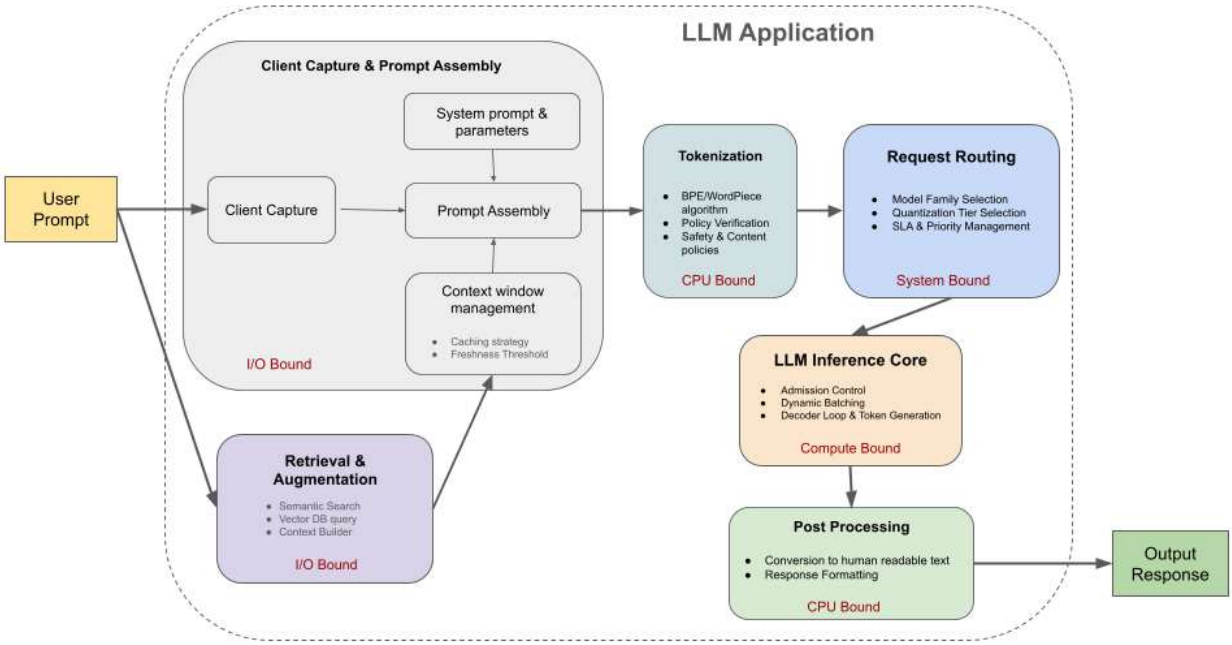

Fig 2: Typical LLM Serving Path: System-Level Overview [3, 4]

2.1 Client Capture and Prompt Assembly

The journey begins at the client interface, where user input is captured and assembled into a formal prompt. This crucial initial stage establishes the foundation for all subsequent processing. Modern LLM systems employ a variety of prompt engineering techniques that significantly impact both response quality and computational efficiency. Context window management determines how much historical conversation data is included in each request, directly affecting memory requirements throughout the pipeline. Instruction formatting organizes directives using model-specific patterns, based on empirical studies, to

enhance response quality. System prompts and role definitions establish behavioral parameters and operational boundaries, while parameter specification configures generation settings such as temperature and sampling methods.

Research indicates this stage is predominantly IO-bound, with network latency emerging as the primary constraint for remote client connections. The implementation of effective prompt length discipline represents one of the most impactful optimization strategies available to system designers. Each additional token included in the prompt creates multiplicative computational costs that propagate throughout the entire serving pipeline. Analysis of production deployments demonstrates that enforcing reasonable prompt length limits can reduce overall computational requirements by substantial margins without significantly impacting response quality [3].

2.2 Retrieval and Augmentation

In the case of systems that employ Retrieval-Augmented Generation (RAG), the phase enhances the prompt with contextual information provided by other knowledge sources. The queries to the vector database correspond to the semantic relevance between the user's query and the stored data, and more complex embedding models are used to retrieve the most relevant content. Cache policies determine whether to use previously retrieved information for similar queries, significantly reducing latency for common request patterns. Freshness thresholds determine when cached data needs to be updated, striking a balance between performance benefits and potential accuracy concerns.

This stage typically presents an IO-bound challenge, particularly when retrieving from remote databases distributed across multiple data centers. A comprehensive analysis of production RAG systems reveals that retrieval latency often becomes the dominant performance bottleneck in these architectures, particularly for requests that require specialized domain knowledge. Sophisticated caching strategies become essential for maintaining responsive performance, especially for frequently asked questions or domain-specific applications. Recent innovations in this area include predictive retrieval mechanisms that anticipate information needs based on conversation context, preemptively caching relevant information before it's explicitly requested [4].

2.3 Tokenization and Policy Verification

Once assembled, the raw text undergoes tokenization—the conversion from human-readable characters to numeric token IDs that the model can process. This transformation uses language-specific rules, such as Byte-Pair Encoding (BPE), which identifies standard subword units. Additionally, it employs WordPiece or SentencePiece algorithms, which utilize slightly different tokenization strategies based on other language properties, as well as vocabulary lookup systems that incorporate special tokens and out-of-vocabulary elements.

Concurrent with tokenization, policy verification systems assess the prompt against safety guidelines, content policies, and usage quotas. These verification processes have evolved from simple pattern matching to sophisticated embedding-based approaches that can identify potentially problematic content with higher precision. Despite this increasing complexity, tokenization and policy verification typically represent lightweight CPU-bound operations that rarely become system bottlenecks. Comprehensive performance analysis demonstrates that even complex policy verification logic adds minimal overhead to overall request processing time, typically measured in single-digit milliseconds even for elaborate verification pipelines [4].

2.4 Request Routing and Model Selection

The tokenized prompt then enters a sophisticated routing layer that makes critical decisions about resource allocation. This component determines which model family handles each request based on specific capability requirements, selects the appropriate quantization tier to balance performance against accuracy, and identifies the optimal hardware acceleration path using GPUs, TPUs, or specialized inference chips. Modern routing systems implement increasingly sophisticated decision algorithms that consider multiple factors simultaneously.

The routing decision-making process incorporates request priority levels, current system load across the entire inference fleet, and service level agreements (SLAs) that may specify maximum acceptable latency for different request categories. Research demonstrates that effective request routing optimizations have a significant impact on global system efficiency by intelligently balancing workloads across heterogeneous compute resources. Production systems typically implement adaptive routing strategies that continuously adjust allocation patterns based on changing load conditions and request characteristics, achieving substantially higher resource utilization compared to static allocation approaches [3].

Stage	Process	Bottleneck Type	Key Optimization Strategy	Impact
Client Capture	Prompt assembly	IO-bound	Prompt length discipline	Reduces computational costs
Retrieval (RAG)	Knowledge integration	IO-bound	Caching strategies	Reduced latency for common queries
Tokenization	Text to token IDs	CPU-bound	Efficient algorithms	Minimal overhead
Policy Verification	Safety & content checks	CPU-bound	Embedding-based approaches	Single-digit ms overhead
Request Routing	Resource allocation	System-bound	Adaptive routing	Improved resource utilization

Table 1: LLM Inference Pipeline: Performance Characteristics and Bottlenecks [3, 4]

3. The Inference Core: Processing and Generation

3.1 Admission Control and Dynamic Batching

Before reaching the model itself, requests navigate through sophisticated admission control systems that implement multiple critical functions. Priority ordering and queue management ensure that high-priority tasks are given a higher priority compared to workloads with lower priorities, thereby avoiding starvation of the lower-priority workloads. Organizing similar requests into active batches ensures a high level of computational efficiency as the fixed costs of processing a request are amortized over several similar requests executed at a given time. These systems, during peak loads, utilize backpressure mechanisms that gracefully degrade service instead of allowing system overload to lead to cascading failures or unpredictable performance properties.

The research demonstrates that the optimal batch size is contingent upon a complex interaction of variables, including the particular model architecture, inherent hardware capabilities, and specified latency requirements. Advanced production systems dynamically adjust batch composition to maximize hardware utilization while meeting response time targets across different priority tiers. Analysis of high-throughput LLM serving environments reveals that adaptive batching strategies can significantly improve overall throughput compared to static approaches, particularly under variable load conditions. Performance studies indicate that implementing sophisticated admission control mechanisms becomes increasingly crucial as model scale increases, with larger models showing greater sensitivity to batch size optimization [5].

3.2 The Decoder Loop: Where Tokens Emerge

The decoder loop is the central part of the system, carrying out the actual computations of the neural network that converts input tokens into output tokens. This step is the most computationally intensive part of the entire pipeline, where hardware ease of use and algorithm optimization have the most significant impact on the system's overall performance. The decoder loop uses various sophisticated methods that enable a substantial enhancement of effectiveness and response compared to naive implementation strategies.

3.2.1 KV-Cache Management

Key-Value (KV) caches retain intermediate attention states, eliminating redundant calculations when generating sequences through autoregressive processes. This optimization significantly reduces the computational requirements for token generation, particularly for more extended output sequences. Paged attention mechanisms optimize memory utilization by implementing efficient memory management strategies that maximize effective context length while minimizing resource requirements. Cache eviction policies maintain optimal working sets based on sophisticated relevance metrics, vital for extended conversations or complex document processing tasks. Hardware-specific memory hierarchies significantly influence caching strategies, with different approaches optimal for various accelerator architectures.

This component represents one of the most memory-intensive operations in the entire serving pipeline, often becoming the limiting factor for concurrent request handling as models scale to larger parameter counts and longer context windows. Comprehensive analysis demonstrates that inefficient KV-cache implementations can reduce effective throughput by an order of magnitude or more in production environments. Recent innovations in this area focus on more efficient memory utilization patterns and novel data structures specifically optimized for transformer attention mechanisms [6].

Aspect	Description	Technical Impact	Optimization Approach
Function	Retains intermediate attention states	Eliminates redundant calculations	Autoregressive optimization
Memory Usage	Highly memory-intensive	Limiting factor for concurrent requests	Paged attention mechanisms
Context Length	Affects working memory requirements	Crucial for long conversations	Efficient eviction policies
Hardware Dependency	Varies by accelerator architecture	Different approaches for different hardware	Architecture-specific implementations
Performance Impact	Critical for throughput	Poor implementation reduces performance	Memory utilization patterns
Recent Innovations	Memory efficiency focus	Novel data structures	Transformer-specific optimizations

Table 2: KV-Cache: The Memory Bottleneck in LLM Inference [5, 6]

3.2.2 Speculative and Assisted Decoding

To accelerate generation, modern systems implement sophisticated acceleration strategies that fundamentally change the traditional autoregressive generation paradigm. Speculative decoding leverages smaller, more efficient models to predict likely token sequences, which the primary model then verifies and refines. This approach effectively trades additional computation for reduced latency, particularly valuable in interactive applications. Verification through larger models ensures quality while maintaining the responsiveness advantages of smaller models. Parallel candidate evaluation improves throughput by considering multiple potential completions simultaneously rather than generating tokens strictly sequentially.

These techniques substantially reduce apparent latency, particularly for predictable outputs such as common phrases or standard responses, where smaller models can effectively anticipate the behavior of larger models. Research indicates that well-implemented speculative decoding can reduce perceived generation latency by substantial margins while maintaining output quality nearly identical to traditional generation approaches. The effectiveness of these techniques varies across content types and application domains, with more predictable outputs showing greater improvements compared to highly creative or specialized generations [5].

3.3 Post-Processing and Response Formatting

Finally, generated tokens undergo essential post-processing steps to prepare them for delivery to the end user. Conversion from token IDs back to human-readable text reverses the initial tokenization process, handling special tokens, whitespace normalization, and other language-specific formatting requirements. Application of safety filters to generated content provides an additional layer of policy enforcement, crucial for preventing harmful outputs that might not have been anticipated during initial prompt verification. Formatting according to client expectations ensures that the response meets specific integration requirements, particularly important for applications that consume structured outputs.

This stage concludes the journey from prompt to response, completing the inference pipeline. While typically less computationally intensive than the decoder loop itself, efficient post-processing implementation remains essential for maintaining overall system responsiveness, particularly for streaming delivery models where perceived latency depends on minimizing processing time for each generated token. Research indicates that post-processing optimizations can measurably improve end-to-end response times, particularly for multi-modal outputs or specialized formatting requirements [6].

Component	Process	Primary Challenge	Key Technique	Performance Impact
Admission Control	Request management	Load balancing	Dynamic batching	Improved throughput
Decoder Loop	Token generation	Computational intensity	Algorithm optimization	Overall system performance
KV-Cache	State retention	Memory consumption	Paged attention	Reduced redundant calculations
Speculative Decoding	Generation acceleration	Latency reduction	Small model prediction	Faster perceived response time
Post-Processing	Output preparation	Format conversion	Safety filtering	End-to-end response time

Table 3: Inference Core Components: Technical Challenges and Solutions [5, 6]

4. Performance Optimization: The Critical Balancing Act

The serving systems that operate adequately and efficiently bring into play a balancing effect on conflicting priorities within multiple dimensions, resulting in the need for high levels of monitoring and constant realignment to ensure optimal performance. The trade-offs inherent in these systems are intrinsic engineering issues that require thoughtful system design and ongoing operational fine-tuning, rather than relying on single-point optimization activities.

The performance trade-off present in the tension between throughput and latency is one of the most observable performance trade-offs in a production setting. Streaming responses may begin faster, creating an improved perceived responsiveness for end users, but potentially reduce overall system efficiency compared to hold-and-release approaches, which can better utilize computational resources through more effective batching. Research demonstrates that the optimal approach varies significantly based on specific application requirements and usage patterns. Interactive applications typically benefit more from streaming delivery despite the potential throughput reduction, while batch processing workloads may achieve substantially higher efficiency through hold-and-release mechanisms. Performance analysis reveals that hybrid approaches can sometimes achieve the benefits of both paradigms by implementing token buffering with dynamic release thresholds adjusted based on current system load [7].

Memory versus compute trade-offs emerge most prominently in quantization decisions, where higher precision representations improve output quality but significantly increase both memory requirements and computational demands. This relationship becomes particularly critical when deploying models at scale, where memory efficiency often determines the practical limits of concurrent request handling. Advanced quantization techniques aim to minimize quality degradation while maximizing efficiency gains, with methods such as mixed-precision quantization showing promising results across diverse workloads. Production deployments increasingly implement dynamic precision selection based on request characteristics, applying more aggressive quantization for less sensitive workloads while maintaining higher precision for applications where output quality is paramount [8].

Another fundamental trade-off for system architecture is concurrency versus response time. Latency increases for larger batches, whereas a larger batch size yields better overall throughput, as the value of available computational resources is better utilized, at the cost of higher average and tail response times. Complex batching algorithms aim to mitigate this effect through techniques such as priority-based batching and dynamic timeouts, although the underlying trade-off is an inherent property of such systems. According to performance studies, the optimal batch size depends significantly on hardware capabilities, model architecture, and service-level objectives [7].

Engineers continuously monitor these and other performance characteristics, adjusting system parameters based on observed usage patterns, available hardware capabilities, and defined business priorities. Modern LLM serving infrastructures implement increasingly sophisticated observability mechanisms that provide detailed visibility into system behavior across multiple dimensions. This monitoring enables both automated adjustments through feedback-driven control systems and human-guided optimization based on comprehensive performance data. The most effective approaches combine deep technical understanding of the underlying systems with precise alignment to business objectives, ensuring that performance optimization efforts focus on the metrics that most directly impact user experience and operational efficiency [8].

Trade-off Area	Approach A	Approach B	Optimization Strategy	Application Context
Latency vs. Throughput	Streaming responses	Hold-and-release	Token buffering with dynamic thresholds	Interactive vs. batch processing
Memory vs. Compute	Higher precision	Aggressive quantization	Mixed-precision quantization	Quality-critical vs. standard tasks
Concurrency vs. Response Time	Large batch size	Small batch size	Priority-based batching	Throughput-focused vs. latency-sensitive
System Monitoring	Automated adjustments	Human-guided tuning	Sophisticated observability	Resource utilization optimization

Table 4: Critical Trade-offs in LLM Serving Systems [7, 8]

Conclusion

Any communication with a large language model (LLM) is based on a well-organized chain of steps in the serving pipeline, including the assembly of instructions, tokenization, resource distribution, admission control, and token production. The trade-offs and optimization opportunities at each stage of this pipeline are different. The conflicting priorities require ongoing monitoring and adjustment to ensure that the system remains capable of meeting the application's requirements. Finally, the user experience is based on the trade-offs that are practical at every stage of the serving pipeline, considering both contextual and operational requirements. The development of further applications of LLMs in pipelines will necessitate a more innovative architecture that can assess trade-off effects and efficiently avoid them to provide the best user experiences in limited operating environments.

Funding: This research received no external funding.
Conflicts of Interest: The authors declare no conflict of interest.

References

[1] Shanmugasundaram Sivakumar, "Performance Optimization of Large Language Models (LLMs) in Web Applications," International Journal of Trend in Scientific Research and Development (IJTSRD), Volume 8, Issue 1, 2024. [Online]. Available: https://www.researchgate.net/profile/Shanmugasundaram-Sivakumar/publication/386342544
[2] Joyjit Kundu et al., "Performance Modeling and Workload Analysis of Distributed Large Language Model Training and Inference," arXiv:2407.14645, 2024. [Online]. Available: https://arxiv.org/abs/2407.14645
[3] Amey Agrawal et al., "On Evaluating Performance of LLM Inference Serving Systems," arXiv:2507.09019, 2025. [Online]. Available: https://arxiv.org/abs/2507.09019
[4] Shailja Gupta et al., "A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions,". [Online]. Available: https://arxiv.org/pdf/2410.12837
[5] Sahin Ahmed, "LLM Inference Optimization Techniques: A Comprehensive Analysis," Medium, 2025. [Online]. Available: https://medium.com/@sahin.samia/llm-inference-optimization-techniques-a-comprehensive-analysis-1c434e85ba7c
[6] Towards AI, "KV Cache: The Key to Efficient LLM Inference,". [Online]. Available: https://pub.towardsai.net/kv-cache-the-key-to-efficient-llm-inference-7260a504efed
[7] Tobiloba Kollawole Adenekan, "Optimizing LLM Latency and Throughput for Interactive Web Interfaces," ResearchGate, 2023. [Online]. Available: https://www.researchgate.net/publication/387223217_Optimizing_LLM_Latency_and_Throughput_for_Interactive_Web_Interfaces
[8] Renren Jin et al., "A Comprehensive Evaluation of Quantization Strategies for Large Language Models," arXiv:2402.16775v1, 2024. [Online]. Available: https://arxiv.org/html/2402.16775v1