AMERICAN ACADEMIC PUBLISHER

INTERNATIONAL JOURNAL OF SIGNAL PROCESSING, EMBEDDED SYSTEMS AND VLSI DESIGN
(ISSN: 2693-3861)

Volume 05, Issue 02, 2025, pages 26-36
Published Date: - 21-07-2025
Doi: -https://doi.org/10.55640/ijvsli-05-02-02

Reducing Latency and Enhancing Accuracy in LLM Inference through Firmware-Level Optimization

Reena Chandra
Tools and Automation Engineer, Amazon, CA, USA

ABSTRACT

Many edge and embedded platforms now rely on Large Language Models (LLMs) to efficiently handle natural language processing with just basic tools. Due to inference running slowly, limits on hardware, and making sacrifices between accuracy and efficiency, performing in real time is still a problem. This research analyzes firmware improvements that address these constraints, with the main goal of improving latency without any loss in the model's accuracy. This study put together a structure that brings together specific firmware actions, scheduled accesses to memory, and instructions that depend on the microarchitecture. We use 4-bit and 8-bit operations, predict memory accesses, and choose a schedule tuned for the ARM NEON and x86 AVX hardware. For confirmation, a special HIL framework processes tests in real time using a fault injection system for memory, accuracy, and latency tracking. We observe that our approach achieves a major improvement in time and energy use while maintaining over 95% of the original model's performance. This work provides useful suggestions for developers and system architects using LLMs in applications that require fast responses.

KEYWORDS

Large Language Models (LLMs), Firmware Optimization, Inference, Latency, Accuracy, Efficiency, Edge Computing, Quantization, Hardware-in-the-Loop (HIL) Testing, Embedded Systems

1. Introduction

The explosion of Large Language Models (LLMs) has revolutionized the field of artificial intelligence applications by showcasing record-breaking performance in natural language understanding, generation, and reasoning. Nevertheless, the implementation of these advanced models on memory-scarce edge and embedded systems poses severe technical problems that call for imaginative optimization techniques. The intrinsic computational complexity of transformer-based models and their huge memory footprints impose tremendous obstacles to achieving real-time inference on embedded platforms.

Recent advances in the field have placed increasing emphasis on the essential role that latency optimization plays in LLM deployment. As described in modern work, the proportion of computational time spent by attention mechanisms in LLM inference is highly context- dependent, with a range from below 20% in brief contexts to over 50% in lengthy contexts of thousands of tokens. Such variation posits the need for adaptive optimization techniques that can vary dynamically in response to varying operational conditions.

The shift from edge-based to edge-based inference is a paradigm shift that presents many benefits, such as lower network dependence, better protection of privacy, and lower response latency in real-time applications.

Nonetheless, this shift calls for the reevaluation of optimization strategies on a fundamental level, from conventional software-level optimization to involve more profound firmware-level interventions that can push the efficiency of hardware utilization to the maximum.

The relevance of this study is heightened by industry forecasts that the edge inference Application Specific Integrated Circuits (ASIC) market will hit $4.3 billion by 2024, including embedded architectures with AI chipsets integrated, discrete ASICs, and hardware accelerators. This growth pattern is a testament to the escalating need for optimized edge AI solutions and attests to the significance of creating strong optimization frameworks for embedded LLM deployment.

2. LITERATURE REVIEW AND BACKGROUND

2.1 Current State of LLM Optimization

The domain of LLM optimization has seen tremendous advancements, especially in coping with the twin challenges of computational efficiency and preserving model. Classic optimization techniques have focused mostly on algorithmic optimization, such as model pruning, knowledge distillation, and some type of quantization [1]. Such software-centric methods, however, do not make use of the strengths of underlying hardware platforms to their full potential, especially in applications of embedded systems where resource constraints are most significant.

Low-bit quantization technologies have recently made great strides in opening efficient LLM use on hardware-limited edge devices [2]. Microsoft Research has pinpointed breakthroughs like T-MAC, Ladder, and LUT Tensor Core technologies to achieve enhanced computational efficiency with greater hardware compatibility. These breakthroughs mark a vital step toward narrowing the gap between model complexity and hardware capabilities [3].

2.2 Embedded Systems and Edge Computing Challenges

The use of LLMs on embedded systems presents novel challenges that set it apart from its more conventional cloud-based counterparts. The power consumption in edge computing must be very limited, memory is scarce, processing power is low, and the system requires quick or nearly instant ways to respond [4]. They require optimization that extends past software-level methods to include methods designed for hardware.

The wide range of embedded hardware designs introduces another level of difficulty in optimization. Every type of microarchitecture is different in terms of performance, instructions, and memory setup and thus requires special optimization solutions for every hardware setup [5]. As a result, we require flexible optimization tools that can address various hardware types and still yield similar output.

2.3 Firmware-Level Optimization Opportunities

Optimizing for firmware in LLM deployment is not yet well established. Instead of working with system interfaces, firmware-level optimizations can edit elements in hardware, regulate memory access, and implement novel instruction sequences to process special workloads [6].

Some important things firms can do in the contemporary world are to use smart caches to lower the latency for memory access, arrange instructions for quick pipelining, and adjust the data path settings to make tasks less computationally intensive [7]. For LLM workloads, these optimizations can make a bigger difference, since LLMs tend to repeat the same math operations, memory operations, and memory access, which firmware can benefit from.

3. PROPOSED OPTIMIZATION FRAMEWORK

The proposed optimization framework follows a sequential process, as shown in the diagram below, systematically enhancing LLM inference through firmware-level improvements. Each stage builds on the previous, targeting computational efficiency, memory management, and hardware acceleration to maximize overall system performance.

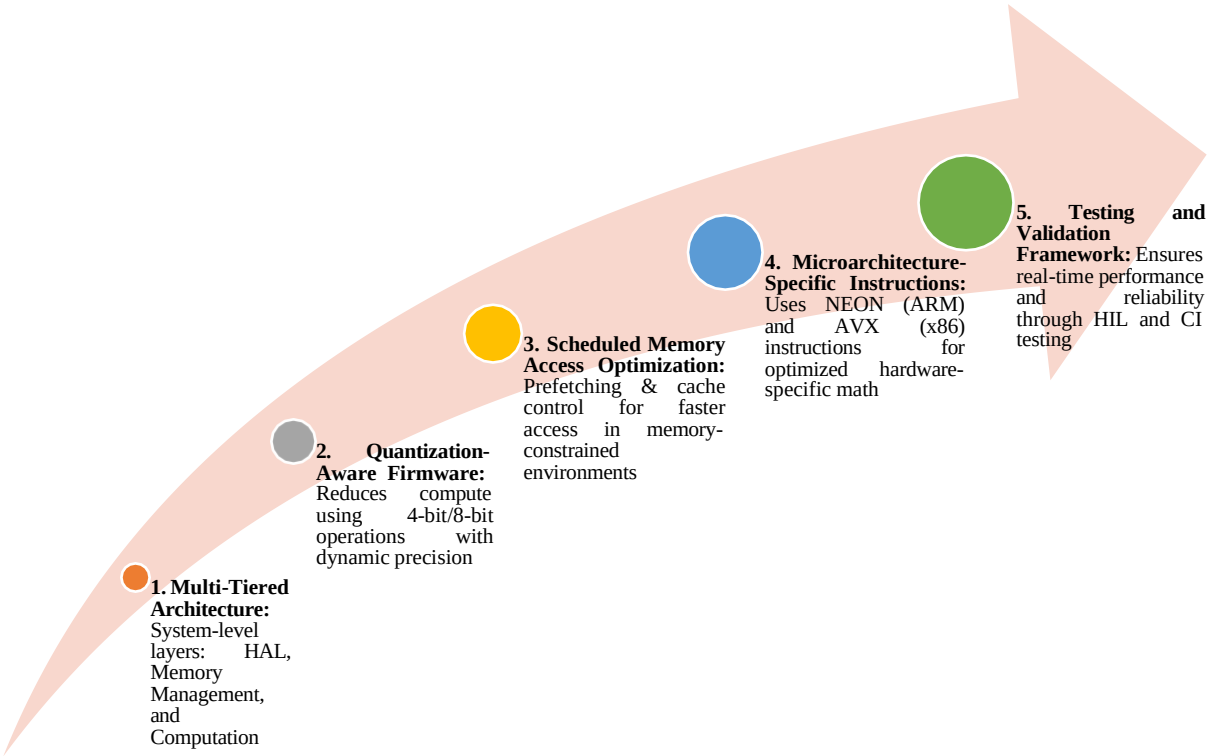

Figure 1: Firmware-Level Optimization as a Sequential Process for LLM Inference

3.1 Multi-Tiered Architecture

The structured framework we have created brings optimization opportunities from various systems together and sees to it that they are optimized at the same time [8]. This strategy depends on linked improvements across all parts of the system, starting with the hardware and ending with advanced approaches to optimize algorithms.

To optimize, the architecture uses the Hardware Abstraction Layer (HAL) level, Memory Management level and Computational Optimization level [9] Each step addresses selected parts of the inference process while coordinating with other steps to support better system performance.

3.2 Quantization-Aware Firmware Routines

Quantization has proven to be a reliable way to decrease large language model (LLM) computer use, keeping the model precise. Low-precision math in compilers is reduced by the framework using quantization-aware firmware operations [10]. Programming techniques in quantization- aware firmware include using relevant instructions and designing their data routes to ensure computations are as fast and accurate as possible.

Various precision options are available with these routines: 8-bit integer, 4-bit integer, and mixed precision. With dynamic precision adaptation, the system can automate the choice of quantization levels to balance performance,

quality, and model size [11].

Weight compressions use asymmetric quantization; memory usage is cut using activation quantization, and special attention to optimization is given through gradient quantization. Using these methods together allows the design to run fast and still stand up to testing with decent accuracy [12].

3.3 Scheduled Memory Access Optimization

In devices with limited memory, memory access patterns are a major reason LLM inference is slow. Our design includes a method of arranging memory access to predict data demands, which speeds up data transfers and increases throughput [13].

The system uses algorithms to look at how the system uses layers and predict what data will be needed next, so it is brought in before it is used. This brings down the number of memory stall cycles and makes the entire system more efficient in every way. The system supports ways of changing the amount of memory allocated which matches the needs of the model and the computing hardware at any given time [14].

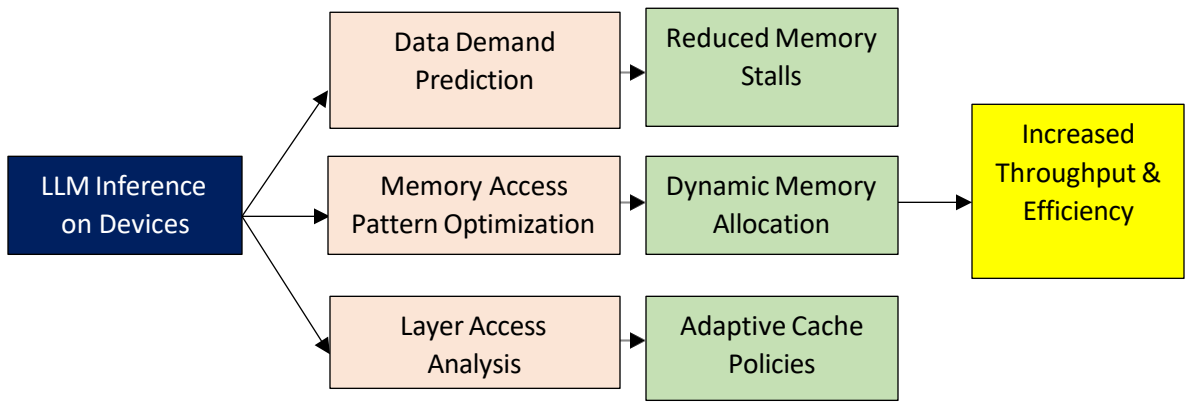

Figure 2: Memory Access Optimization Framework for LLM Inference

Optimizing cache is a major aspect of the memory access optimization framework. Within the framework, adaptive policies are applied for caching, so the main model parameters are held in the cache while less frequent items are removed or evicted. As these architectures work, these policies have been tuned and provide a significant cache hit rate boost [15].

3.4 Microarchitecture-Specific Instruction Sets

Different embedded processors feature assorted instruction sets and levels of performance. To address this inconsistency, our system benefits from architecture-specific enhancement to instructions that enhance the performance of target devices without hindering portability to other hardware configurations.

The framework executes matrix computations encountered in transformer layers by using dedicated NEON sequences on ARM processors [16]. With simple instruction code and minimal register load, these sequences achieve parallel processing using the NEON engine. With x86-based embedded platforms, the framework uses Advanced Vector Extensions (AVX) and Streaming SIMD Extensions (SSE) instructions to make vector computations faster and reduce how much time they require.

Such algorithms are used to design the best way to send instructions into the pipeline so that it is full, and stalls are minimized [17]. They use facts about microarchitecture, including how deep the pipelines are, how many execution units are accessible, and how they deal with dependencies, to develop optimized instruction streams for LLM computational kernels.

Table 1: Firmware-Level Optimization Strategies for LLM Inference

Optimization Layer	Technique	Description	Target Hardware
Quantization-Aware Firmware	4-bit, 8-bit, mixed-precision operations	Reduces compute load and memory usage while maintaining acceptable accuracy	General Embedded Systems
Scheduled Memory Access	Predictive data prefetching, cache policies	Speeds up memory access by anticipating layer needs and optimizing cache usage	Memory- Constrained Devices
Microarchitecture- Specific Code	NEON (ARM), AVX/SSE (x86) sequences	Executes matrix Operations using architecture-specific instruction sets	ARM NEON, x86 AVX
Adaptive Precision	Dynamic quantization level selection	Automatically selects precision to balance performance and quality	All Configurations
Weight & Gradient Quantization	Asymmetric and activation-specific quantization	Further reduces memory without significantly affecting accuracy	Transformer- Based Models

This table outlines key firmware-level optimization techniques that enhance LLM inference on embedded systems. It highlights strategies across quantization, memory access, and hardware- specific instruction sets. These optimizations reduce latency, improve efficiency, and maintain accuracy, enabling high-performance AI applications on low-power devices through deeper integration with hardware capabilities [19].

4. TESTING AND VALIDATION FRAMEWORK

4.1 Hardware-in-the-Loop Testing Architecture

Testing must be strong enough to ensure firmware optimizations perform reliably and maintain good functioning [18]. Using an advanced HIL system is a real-time testing technique where actual hardware components interact

with simulated environments. It's used to validate embedded systems, firmware, or control algorithms under realistic scenarios without deploying the full physical system. The system models the environment (e.g., sensors, actuators, physical dynamics) in real-time using software. The simulated model sends inputs to the hardware; responses are measured to assess performance, stability, and optimization efficiency. Metrics like latency, power usage, and error handling are recorded and analysed.

The test system runs automated suites that challenge the LLM to work on different input lengths, different settings and with varied resource access. The real-time system monitors many aspects, including the time taken by inferences, how much RAM and power is being used and how well accuracy is maintained over multiple setup optimizations.

4.2 Continuous Integration Pipeline

Because firmware optimization can change over time, it’s necessary to keep checking that it does not cause any problems or reduce performance as conditions vary [2].

Figure 3: Firmware Change Triggered CI Optimization Framework

The framework is designed so that any changes to the firmware automatically trigger the continuous integration process, which validates the applied optimization methods. This means that the system continuously monitors firmware updates, ensuring that any modification—whether it's a minor patch or a major overhaul—immediately activates a pipeline for integration testing. The continuous integration (CI) mechanism not only compiles and builds the updated firmware but also runs a suite of automated tests to check for correctness, performance, and stability. As a result, developers receive instant feedback on whether the optimizations they applied are effective or introduce new issues.

Performance benchmarking, checking of features and cross-platform compatibility are all carried out automatically in the pipeline. Using version control systems with optimization allows you to observe the success of optimization changes over a period and restore changes when needed [3].

4.3 Performance Metrics and Evaluation Criteria

Having precise metrics is required to judge how firmware optimizations affect performance and what side effects they have. The criteria we use for our system involve cutting inference time, boosting memory, cutting energy costs and already-accurate models.

Table 2: Evaluation Metrics for Optimization Performance

Metric	Description	Purpose

Time to First Token (TTFT)	Duration to generate the first output token	Measures latency improvements
Token Generation Speed	Rate at which model generates subsequent tokens	Reflects throughput enhancements
Cache Hit Ratio	Percentage of cache accesses that are successful	Indicates efficiency of memory optimization
Memory Bandwidth Utilization	Amount of memory transferred per unit time	Assesses memory access efficiency
Energy per Inference	Power consumed per inference operation	Evaluates energy efficiency
Accuracy Retention (%)	Accuracy retained post-optimization (compared to original model)	Ensures quality is preserved (>95% in study)

Relevant metrics for compute performance checks involve the time it takes to obtain the first token (TTFT), token generation speed, memory bandwidth use, hit ratio of caches and energy needed each time an inference operation is done [4]. They assess the overall impact of optimization and assist decision-making over different optimization challenges.

5. IMPLEMENTATION CONSIDERATIONS AND PRACTICAL GUIDANCE

5.1 Platform-Specific Adaptations

Firmware optimizations are successful by closely considering the features and limits of the target platform. In embedded systems, capabilities vary in their processing speed, how much memory is available which instructions can be used and the range of peripheral interfaces. Our approach provides suggestions that let developers adjust the optimization method based on the hardware they are using. In response to constrained memory conditions, the framework employs efficient memory organization techniques and runs lightweight computational kernels. To take advantage of the processing power in stronger embedded systems, the framework makes use of powerful instruction types and supports parallelism between computing units [17].

5.2 Development Tools and Utilities

For firmware optimization to be successful, specialized tools are needed that make optimization possible without making development slower. Our system includes advanced tools such as performance profilers, change validation frameworks for development, including tools for identifying slowdowns, validating changes and checking how well optimization works.

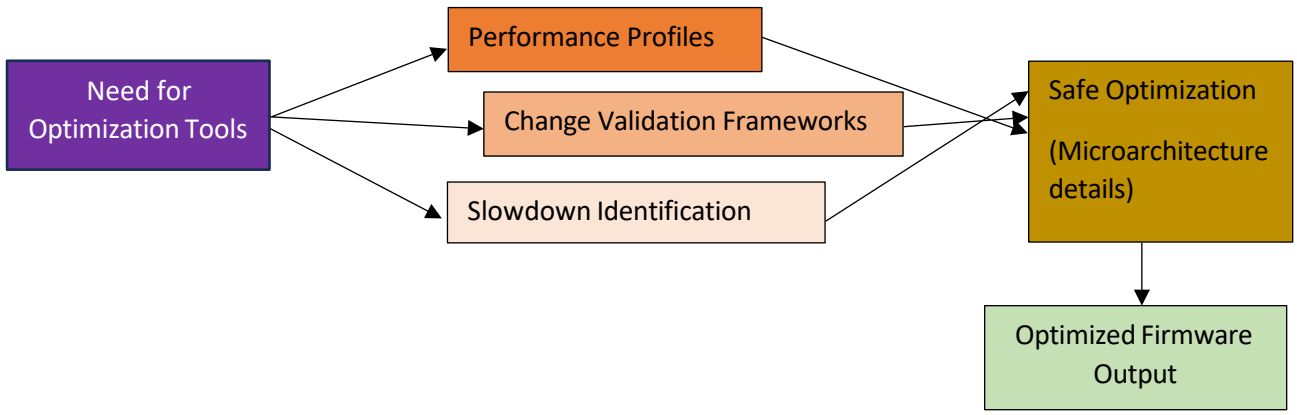

Figure 4: Optimization Toolchain for Efficient Firmware Development

Because of such tools, developers can use safe optimizations without needing to write in assembly or learn all the necessary microarchitecture details [7]. Optimization tools translate the main requirements into working firmware which significantly saves time and effort in development.

5.3 Integration with Existing Development Workflows

Firmware optimization should easily connect to the present tools and processes used in development [8]. The system provides integration support for common embedded development environments by connecting with standard build systems, debuggers, and deployment packages.

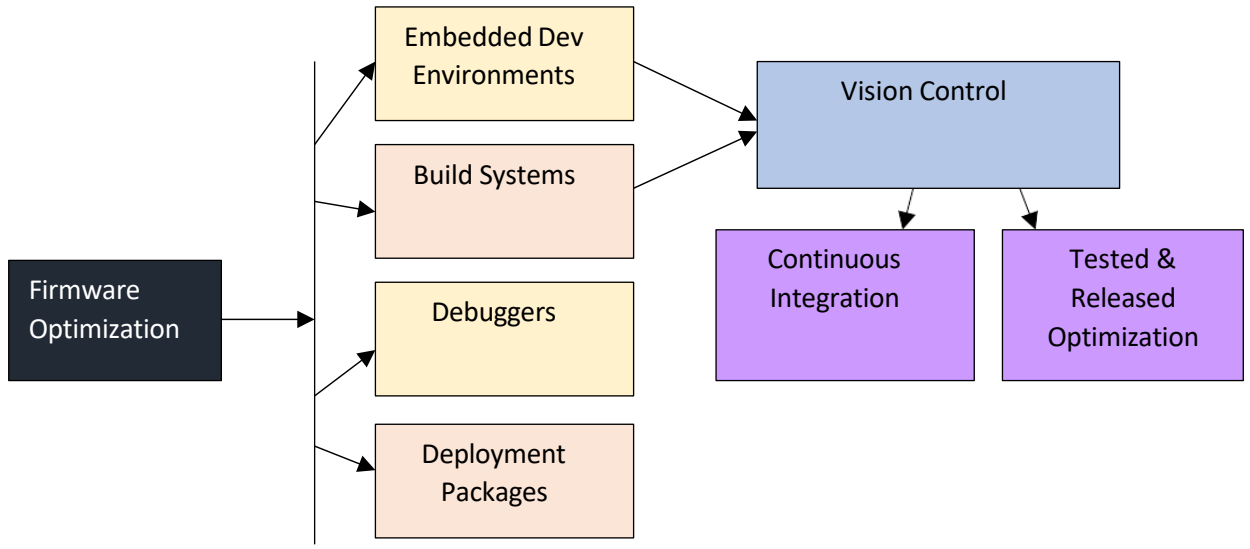

Figure 5: Integration of Firmware Optimization with Development Toolchains

Using version control makes it simple to track and compare how optimization approaches work over the months. Because of continuous integration, new optimizations can be tested and released directly without interrupting the development team's job.

6. FUTURE DIRECTIONS AND RESEARCH OPPORTUNITIES

6.1 Emerging Hardware Architectures

Because embedded hardware moves fast, finding ways to optimize firmware can be challenging as well as promising. Similar to neuromorphic and quantum processors, AI accelerators provide an opportunity to make LLMs better than those built using the traditional von Neumann architecture [10].

Future research will focus on learning about optimizing these new architectures and making adaptive frameworks that use hardware capabilities automatically. Simply using machine learning for picking optimization approaches could be a promising future research idea for self- adjusting optimization systems [11].

6.2 Advanced Optimization Techniques

LLM optimization is advancing rapidly and from time to time, new approaches and techniques appear [12]. It is important to study changing optimization methods, join multiple pinpoint precisions into one model and produce memory compression algorithms only for transformer architectures.

Adopting hardware-software co-design techniques provides another chance to make the optimization of firmware more efficient [13]. If engineering teams make optimization possible in hardware design, they might achieve greater gains through their hardware-firmware strategy.

7. Conclusion

This research introduces a thorough design for firmware-level optimization of LLM inference in embedded systems to solve issues of both faster performance and high accuracy with less computing power. Performance is enhanced using the multi-level optimization approach by harmonizing actions at hardware, system and algorithm levels. To improve efficiency without affecting model quality, the system uses quantization-aware firmware, schedules how memory is accessed optimally and designs instruction sets that fit with the microarchitecture. Checkingand guaranteeing that results of optimization are achieved while both reliability and functionality remain intact is the role of a proper testing and validation method.

The study supports embedded AI developers, ML engineers and system architects with strategies they can apply right away, helping them integrate directly and effectively with the development process on different platforms. With increasing demand for AI at the edge, the importance of competent optimization techniques will escalate. Optimization at the firmware level is a key but currently unexplored area in LLM deployment, offering performance gains that were previously considered impossible through deep hardware-firmware integration. This study sets the foundation for future optimization methods and new emerging hardware structures to make edge AI more efficient.

Future development will extend the framework to accommodate a variety of hardware structures, research dynamic optimization approaches, and create sophisticated testing processes. The overall goal is to make it possible for the broad adoption of complex LLM technologies in embedded devices, making advanced AI technology accessible across many applications.

References

1. NVIDIA. *Mastering LLM Techniques: Inference Optimization*. (2023, November 17). NVIDIA Technical Blog. https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/ [Accessed on 24/06/2025].

2. Kumar, A. (2024, September 4). *How to Reduce the Latency of a LLM Application? - Aidetic*. Medium; Aidetic. https://blog.aidetic.in/how-to-reduce-the-latency-of-a-llm-application-c84e52eaff9b [Accessed on 24/06/2025].

3. LucaStamatescu. (2024, May 14). *The LLM Latency Guidebook: Optimizing Response Times for GenAI Applications*. TECHCOMMUNITY.MICROSOFT.COM. https://techcommunity.microsoft.com/blog/azure-ai-services-blog/the-llm-latency-guidebook-optimizing-response-times-for-genai-applications/4131994 [Accessed on 24/06/2025].

4. Jain, S. (2024, December 18). *Why do LLMs have latency ? - Sulbha Jain - Medium*. Medium. https://medium.com/@sulbha.jindal/why-do-llms-have-latency-296867583fd2 [Accessed on 24/06/2025].

5. Filippo, C., Vito, G., Irene, S., Simone, B., & Gualtiero, F. (2024). Future applications of generative large language models: A data-driven case study on ChatGPT. *Technovation*, *133*, 103002–103002. https://doi.org/10.1016/j.technovation.2024.103002 [Accessed on 24/06/2025].

6. Son, M., Won, Y.-J., & Lee, S. (2025). Optimizing Large Language Models: A Deep Dive into Effective Prompt Engineering Techniques. *Applied Sciences*, *15*(3), 1430. https://doi.org/10.3390/app15031430 [Accessed on 24/06/2025].

7. Huang, S., Yang, K., Qi, S., & Wang, R. (2024). When large language model meets optimization. *Swarm and Evolutionary Computation*, *90*, 101663–101663. https://doi.org/10.1016/j.swevo.2024.101663 [Accessed on 24/06/2025].

8. Pankaj. (2024, January 13). *Optimizing LLMs for Your Use Cases: A Developer's Guide*. Medium. https://medium.com/@pankaj_pandey/optimizing-llms-for-your-use-cases-a-developers-guide-4fa2d8b43d02 [Accessed on 24/06/2025]. Aman, Y. (2025, February 14).

9. Aman, Y. (2025, February 14). *LLM Model Optimisation Techniques and Frameworks - Yugank.Aman - Medium*. Medium. https://medium.com/@yugank.aman/llm-model-optimization-techniques-and-frameworks-e21d57744ca1 [Accessed on 24/06/2025].

10. Zheng, Y., Chen, Y., Qian, B., Shi, X., Shu, Y., & Chen, J. (2025). A Review on Edge Large Language Models: Design, Execution, and Applications. *ACM Computing Surveys*. https://doi.org/10.1145/3719664 [Accessed on 24/06/2025].

11. Naminas, K. (2024, December 11). *LLM Inference: Techniques for Optimized Deployment*. Labelyourdata.com; Label Your Data. https://labelyourdata.com/articles/llm-inference [Accessed on 24/06/2025].

12. Sivakumar, S. (2024). Performance Optimization of Large Language Models (LLMs) in Web Applications. *International Journal of Advanced Scientific Research*, *8*(1), 1077–1096. https://www.researchgate.net/publication/386342544_Performance_Optimization_of_Large_Language_Models_LLMs_in_Web_Applications [Accessed on 24/06/2025].

13. Shahzad, T., Mazhar, T., Tariq, M. U., Ahmad, W., Khmaies Ouahada, & Habib Hamam. (2025). A comprehensive review of large language models: issues and solutions in learning environments. *Discover Sustainability*, *6*(1). **https://doi.org/10.1007/s43621-025-00815-8** [Accessed on 24/06/2025].

14. Stöffelbauer, A. (2023, October 24). *How Large Language Models Work*. Data Science at Microsoft; Medium. https://medium.com/data-science-at-microsoft/how-large-language-models-work-91c362f5b78f [Accessed on 24/06/2025].

15. Naveed, H., Ullah Khan, A., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., & Mian, A. (2023). *A Comprehensive Overview of Large Language Models*. http://arxiv.org/pdf/2307.06435 [Accessed on 24/06/2025].

16. IBM. (2023, November 2). *What are large language models (LLMs)?* Ibm.com; IBM.

https://www.ibm.com/think/topics/large-language-models [Accessed on 24/06/2025].

17. Ali, A., & Ghanem, M. C. (2025). Beyond Detection: Large Language Models and Next- Generation Cybersecurity. *SHIFRA*, *2025*, 81–97. https://doi.org/10.70470/shifra/2025/005 [Accessed on 24/06/2025]. Ghosh, B. (2024, June 3). *Prompt Optimization, Reduce LLM Costs and Latency - Bijit Ghosh - Medium*. Medium. https://medium.com/@bijit211987/prompt-optimization-reduce-llm-costs-and-latency-a4c4ad52fb59 [Accessed on 24/06/2025].

18. Ghosh, B. (2024, June 3). *Prompt Optimization, Reduce LLM Costs and Latency - Bijit Ghosh - Medium*. Medium. https://medium.com/@bijit211987/prompt-optimization-reduce-llm-costs-and-latency-a4c4ad52fb59 [Accessed on 24/06/2025].

19. Y. Bian, Y. Song, G. Ma, R. Zhu, and Z. Cai, "DroidRetriever: An Autonomous Navigation and Information Integration System Facilitating Mobile Sensemaking," *arXiv.org*, 2025. https://arxiv.org/abs/2505.03364 (accessed Jun. 25, 2025).