International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Prompt Context Caching Architecture for Cost Reduction in Large Language Model Systems

Vishram Singh

Submitted: 03/04/2026 **Revised:** 13/05/2026 **Accepted:** 26/05/2026

Abstract: Enterprise adoption of large language models has created a persistent and growing tension between capability and operational cost. Token-based inference pricing causes expenditure to scale directly with prompt length, and because enterprise prompts are typically assembled from layered components, behavioral instructions, domain knowledge, and user queries, the majority of tokens transmitted per request carry no new information relative to prior requests. Prompt Context Caching architecture deals with such inefficiency by adding a structured caching mechanism between the application and the inference endpoint. Prompts are represented as being in the static, shared, and dynamic levels according to how they vary between requests. Deterministic hashing is used to give each reusable segment a cryptographic fingerprint, which allows one to perform reliable authentication of identity without interpreting semantics. Fingerprinted chunks are placed into a distributed in-memory cache and returned on further requests, allowing the system to reassemble entire prompts without redelivering previously served data. The architecture integrates naturally with retrieval-augmented generation pipelines, where injected document context represents a high and frequently repeated token cost. Hybrid cache lifecycle management, including TTL-based expiration, version-based invalidation, and security-conscious segmentation policy, guarantees that the content stored in the cache is up-to-date and secure. It has been evaluated to achieve significant performance improvements on simulated enterprise workloads in terms of token consumption and inference cost, and significant reductions in request latency, without deteriorating response accuracy or model behavior. The architecture requires no modification to the underlying model or inference provider and can be layered onto existing pipelines with modest engineering effort. Prompt context caching represents a scalable, infrastructure-level optimization essential for cost-efficient deployment of large language models at enterprise scale.

Keywords: *Large Language Models, Prompt Context Caching, Token Optimization, Distributed Inference Systems, Retrieval-Augmented Generation*

1. Introduction

The emergence of large language models as general-purpose reasoning engines has driven their adoption across enterprise domains, including customer support automation, financial decision assistance, developer productivity tooling, and intelligent knowledge retrieval. The capabilities demonstrated by these models, particularly their ability to perform complex reasoning from natural language instructions alone, without task-specific fine-tuning, have made them attractive targets for rapid integration into production systems [1].

However, the commercial economics of LLM inference differ fundamentally from those of traditional software services. In most cloud-based LLM platforms, billing is computed directly from token consumption: every token in the input prompt, regardless of whether it carries new information or merely repeats static context, is priced identically to freshly generated output. This pricing structure creates a structural inefficiency in applications where prompts are composed partly of content that remains unchanged across thousands or millions of requests [2].

The problem is not marginal. Enterprise prompts are composite documents. A typical production prompt for a conversational assistant might include several hundred tokens of behavioral instructions, over a thousand tokens of domain knowledge, and perhaps a few hundred tokens of user query. Only the last

Independent Researcher, USA

component varies per request. The first two, which together may account for over 90% of the prompt by token count, are retransmitted and reprocessed in full with every call to the inference endpoint. At scale, this pattern generates substantial and avoidable costs.

This article examines the prompt context caching architecture as a systems-level response to this inefficiency. The architecture interposes a caching layer between the application and the LLM provider, segments incoming prompts into static and dynamic components, fingerprints reusable segments for identity verification, and reconstructs complete prompts on the fly using cached representations. The result is a significant reduction in live token transmission without any change to the model or its outputs.

2. Background: Token-Based Inference Economics and Prompt Engineering

2.1 The Cost Structure of LLM Inference

The foundational cost model for LLM inference is expressed as:

Cost = (Prompt Tokens + Output Tokens) × Price per Token

This relationship means that reducing prompt token volume is the most direct lever available for cost control, since prompt tokens are within the application developer's control in ways that output tokens largely are not. Few-shot prompting, the practice of embedding examples directly in the prompt to condition model behavior, has been shown to yield strong task performance without fine-tuning [1], but it also substantially increases prompt length, exacerbating the cost problem.

Research into retrieval-augmented generation (RAG) has demonstrated that injecting relevant external documents into the prompt at inference time significantly improves factual accuracy on knowledge-intensive tasks [3]. However, this benefit comes with a direct token cost: each retrieved document adds tokens to every request. In adaptive RAG configurations, careful management of document selection has been shown to achieve comparable accuracy at approximately one-quarter of the per-query API cost of naive retrieval strategies, illustrating that architectural choices around prompt construction have measurable economic consequences [4].

Prompt Component	Typical Token Range	Mutability	Cache Eligibility
System Instructions	600 – 1,000	Static	High
Domain Knowledge Context	900 – 1,500	Semi-static	High
Conversation History	200 – 600	Session-bound	Conditional
User Query	50 – 300	Dynamic	None

Table 1: Prompt Component Token Distribution in Enterprise Workloads [3, 4]

2.2 Prompt Engineering Complexity Spectrum

Prompt engineering practice spans a wide range of implementation complexity. At the lower end, static prompts and prompt templates provide fixed or lightly parameterized instructions with minimal runtime construction. As complexity increases, practitioners move through contextual prompts, prompt composition, prompt chaining, and eventually retrieval-augmented generation pipelines and autonomous agent architectures. At the upper end, prompt tuning and soft prompts embed learned parameters directly into the model's input representations [5].

Caching strategies are most directly applicable to the lower-complexity tiers, static and composed prompts, where content reuse is highest. They also apply meaningfully to RAG pipelines, where retrieved document context is often identical across semantically similar queries. In both cases, the key insight is that prompt segments with high reuse frequency are candidates for caching regardless of where they fall in the complexity hierarchy.

Implementation Tier	Complexity Level	Flexibility	Cache Benefit
Static Prompts	Lowest	Minimal	Maximum
Prompt Templates	Low	Low	High
Prompt Composition	Moderate	Moderate	High
Retrieval-Augmented Generation	High	High	Moderate–High
Autonomous Agents	Highest	Maximum	Low–Moderate

Table 2: Prompt Engineering Complexity and Cache Applicability [5]

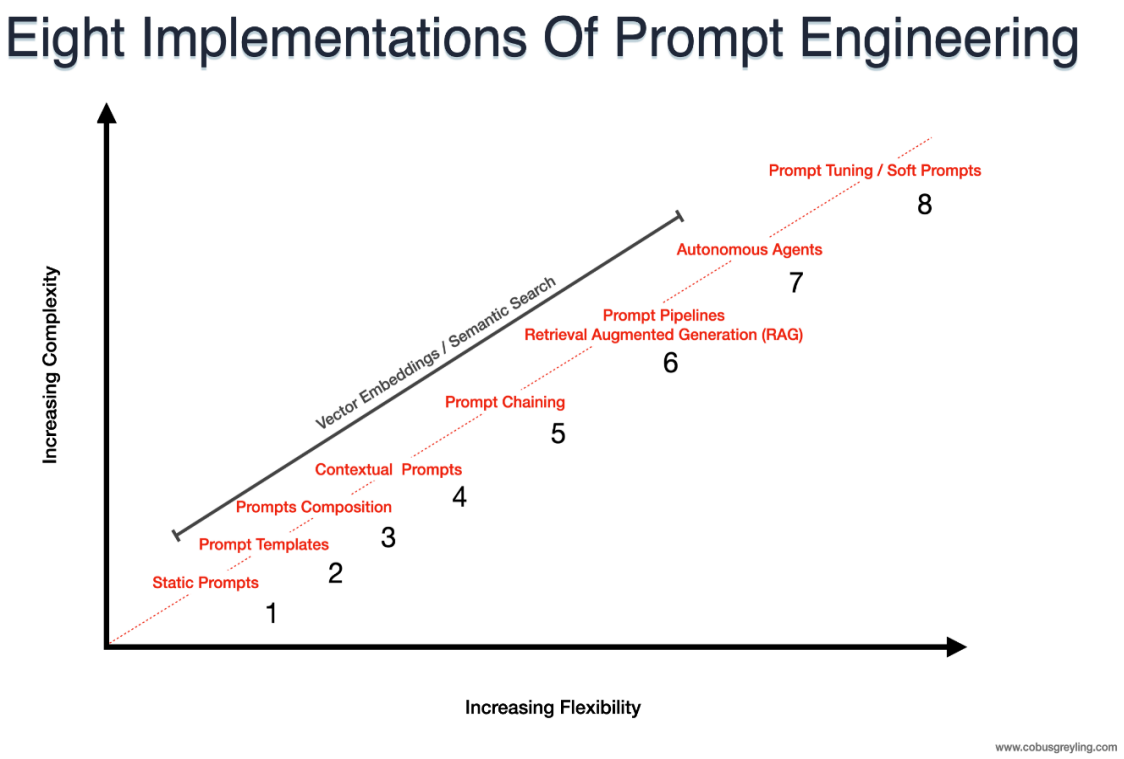

Fig. 1: Prompt Engineering Tiers [5]

2.3 Caching as a Systems Principle

Distributed caching is a mature optimization technique across software infrastructure. HTTP caching, database query result caching, and CDN edge caching all operate on the same fundamental principle: store the result of an expensive computation and serve it from fast local storage on repeated requests. avoiding the cost of recomputation. These patterns are well-established in systems design literature and have been shown to provide order-of-magnitude improvements in throughput and latency under appropriate workload characteristics [6].

The application of these principles to LLM prompt processing is architecturally natural but practically underexplored. Prompt context caching differs from traditional caching in one important respect: the unit of caching is not a complete request-response pair but a *component* of the request. This sub-request granularity requires a segmentation step that has no direct analog in HTTP or database caching, but the core mechanics of key-value lookup, hit/miss handling, and eviction policy carry over directly.

3. Prompt Segmentation Strategy

3.1 Logical Decomposition of Prompt Structure

Instead of a threaded array of prompts, the PCCA takes prompts to be structured documents, which are comprised of identifiable layers of logical structure. The prompt content is categorized into three levels of mutability in the segmentation strategy:

Static Context: This includes the content that is established once and seldom modified, such as system role definitions, behavioral guidelines, compliance rules, and output format specifications. This level is the most reusable in terms of requests and the main target of caching.

Shared context includes domain information that applies to a large group of users or sessions but can be updated on a periodic basis, such as product catalogs, policy texts, validation rule collections, or knowledge base snippets. This level uses the advantage of caching and the right invalidation measures.

The actual query or instruction by the user is called "dynamic input." It is unique per request and not cacheable. It is appended to the reconstructed cached context immediately before prompt submission.

This three-tier model enables precise identification of cacheable content without requiring the application to restructure its prompting logic significantly. The segmentation engine examines prompt structure, typically delimited by role tags, separator tokens, or explicit section markers, and assigns each segment to its appropriate tier [7].

3.2 Practical Example

A concrete illustration from an enterprise accounting assistant demonstrates the value of segmentation. The static context might read: *"You are an enterprise accounting assistant. Follow company financial compliance rules."* The shared context might contain a structured set of accounting validation rules for journal entries, running to several hundred tokens. The dynamic input is the specific transaction to be validated, perhaps thirty to one hundred tokens. After segmentation, only the dynamic input requires transmission as live content; the other two tiers are served from cache. Given the token distribution described in the experimental workload, 2,000 tokens static, 500 tokens shared, and 200 tokens dynamic, the cacheable fraction of each prompt represents over 92% of its total length.

4. Context Fingerprinting and Cache Lookup

4.1 Hash-Based Segment Identity

For the cache to serve a prompt segment reliably, it must be able to determine with certainty whether an incoming segment matches a previously stored one. The PCCA accomplishes this through cryptographic hashing. Each prompt segment is passed through SHA-256, producing a 256-bit digest that serves as the cache key:

- Fingerprint = SHA256(Context Segment)
- SHA-256 is deterministic; identical input always produces identical output and is collision-resistant to a degree that makes accidental key collisions negligible in any realistic deployment scenario [8]. The hash is computed locally by the prompt processor before any network call is made, adding negligible latency.

On each incoming request, the fingerprint of each segmented component is computed and submitted to the cache store as a lookup key. A cache hit returns the stored segment representation; a cache miss triggers transmission of the segment to the LLM and subsequent storage of the result. The lookup mechanism is O(1) regardless of segment length, making fingerprint-based identity verification efficient even for very large static contexts.

4.2 Generation with Prompt Cache

At the inference layer, models that support KV-cache reuse, where the attention states computed during prefill for a cached prefix are stored and reused across requests, compound the efficiency gains from prompt-level caching. As illustrated in the technical documentation of generation with a prompt cache, the model stores precomputed attention states for cached prompt segments and reads them directly during generation rather than recomputing them [9]. This translates the token savings from prompt-level caching into latency savings at the inference layer, since the prefill computation for the static portion of the prompt is skipped entirely.

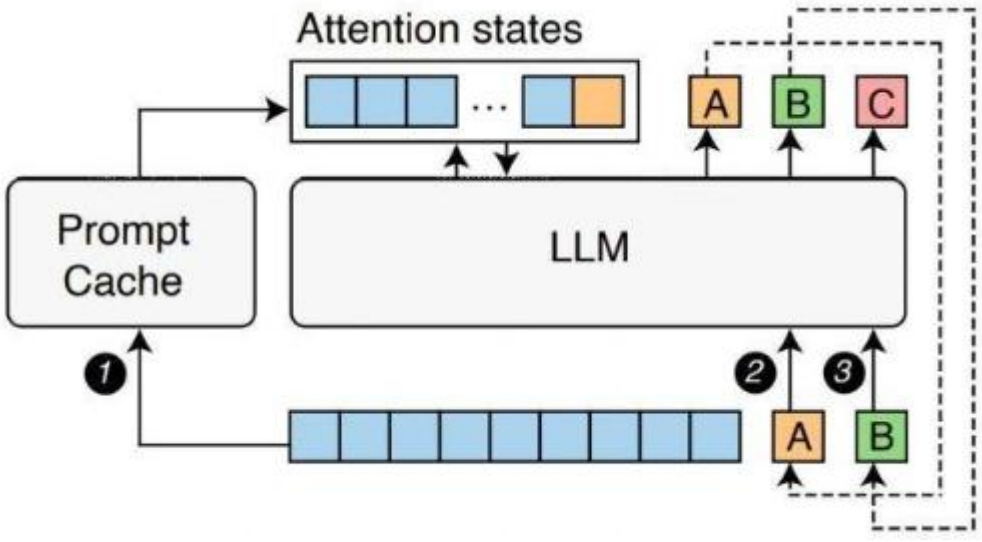

Fig. 4: Prompt Cache Generation [9]

5. Cache Storage Architecture and Lifecycle Management

5.1 Data Model

The cache data model is designed to support efficient lookup, cost accounting, and eviction prioritization:

Field	Description
Context ID	SHA-256 hash identifier serving as the primary key
Context Text	The full text of the cached prompt segment
Token Length	Precomputed token count for cost tracking and reporting
Usage Frequency	Access count used for eviction policy and analytics

Table 3: Cache Data Model Fields and Operational Descriptions

The token length field allows the system to calculate the current number of tokens being saved up on each hit on the cache, which would give the operational visibility necessary to justify and tune the caching infrastructure [10].

5.2 Distributed Cache Infrastructure

In the case of production deployments with a high volume of requests, the cache has to be spread among several nodes so that the system is not a single point of failure and can be scaled horizontally. An implementation of the Redis Cluster on Kubernetes offers a time-tested architecture in this regard: Redis has atomic operations; TTL-based expiration, which can be configured; and a variety of other eviction policies (LRU, LFU), which, by default, are well-aligned with the usage frequency field on the data model [6]. Kubernetes allows the cache tier to be scaled without affecting the application tier or the LLM gateway tier and therefore ensures that cache capacity expands with an increase in traffic.

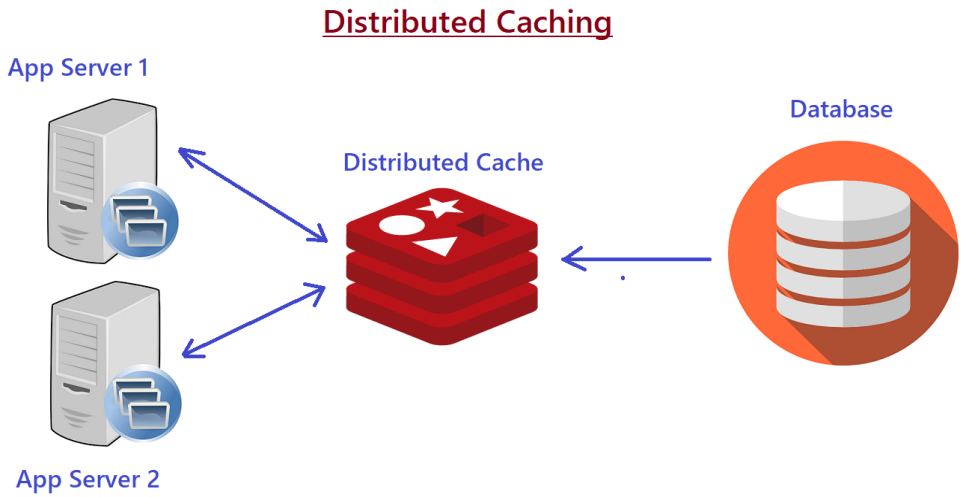

Fig. 2: Distributed Cache [6]

5.3 Cache Expiration and Invalidation

There are three invalidation strategies that are used with respect to various segments and update patterns:

TTL-based expiration will be suitable for shared context, which is updated in a well-known schedule; daily knowledge base updates; and policy updates weekly. The TTL will be configured to be equal to the refresh time so that stale material is thrown out before the next update period.

Version-based invalidation occurs programmatically in case the upstream content is modified in an irregular way. The application layer emits an invalidation signal carrying the affected segment's fingerprint; the cache deletes the corresponding entry immediately.

Semantic context validation is a more advanced strategy applicable when the meaning of a segment may drift independently of its text, for example, when external facts referenced in the context become outdated. This requires integration with an external validation signal and represents an area for future development [11].

5.4 Security Considerations

On-demand content presents a security concern that needs to be taken care of in the production deployment process. Segments containing personally identifiable information, session-specific data, or confidential user content must be excluded from caching through a careful segmentation policy. Content that does enter the cache should be encrypted at rest and in transit. For multi-tenant deployments, cache namespace isolation ensures that one tenant's prompt content cannot be accessed by another [8].

6. Integration with Retrieval-Augmented Generation

PCCA can be unnaturally combined with retrieval-augmented generation pipelines, and its effect may be especially important. In RAG systems, a retrieval step fetches relevant documents from a knowledge store and injects them into the prompt alongside the user query [3]. These injected documents are often hundreds to thousands of tokens, and the same documents may be retrieved repeatedly across semantically similar queries.

By caching retrieved document segments at the fingerprint level, the PCCA eliminates redundant retransmission of frequently retrieved content. Adaptive RAG research has demonstrated that overlapping prompt strategies, where document segments are composed to maximize context reuse, achieve the highest accuracy per unit of API cost [4]. Prompt context caching applies this principle to the infrastructure layer; for example, instead of using a prompt construction strategy to construct documents so as to minimize cost, the cache makes sure that any document fragment sent into the system can be reused in a later request.

The combination of adaptive retrieval strategy and infrastructure-level caching creates a compounding effect: adaptive RAG reduces the number of tokens retrieved per query, while caching eliminates redundant retransmission of the tokens that are retrieved. Together, they address both the breadth and the repetition dimensions of RAG token cost [12].

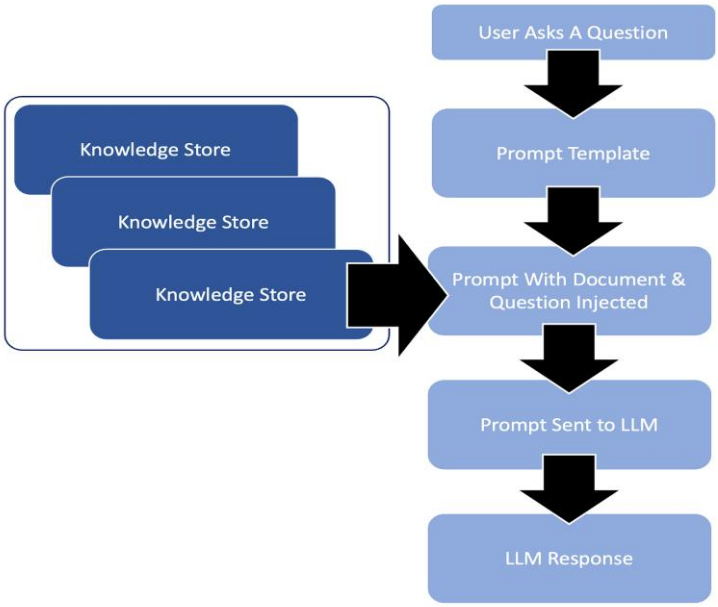

Fig. 3: RAG Pipeline [12]

7. Experimental Evaluation

7.1 Workload Simulation

The evaluation simulated enterprise AI workloads representative of three common deployment patterns: accounting assistant queries requiring compliance-aware reasoning, enterprise knowledge retrieval against structured domain content, and internal developer assistant interactions. The dataset comprised 10,000 prompt requests with the following composition:

Segment	Token Count
Static Context	2,000
Shared Context	500
Dynamic Input	200
Total per request	**2,700**

Table 4: Prompt Segment Composition and Token Distribution per Request

Two system configurations were evaluated: a baseline transmitting complete prompts without caching and the PCCA with distributed caching enabled. Metrics collected included total token consumption, inferred cost, and end-to-end request latency [13].

7.2 Continuous Batching and Latency Baselines

Latency evaluation drew on the performance characteristics of continuous batching inference systems, which represent the current state of the art for production LLM serving. Continuous batching, as implemented in systems such as text-generation-inference and vLLM processes, requests dynamically rather than waiting to assemble fixed-size batches, significantly reducing tail latency relative to static batching approaches such as FasterTransformer. Request latency CDF analysis QPS=4 shows that continuous batching systems achieve median latencies in the 20–50 second range under moderate load, while static batching systems exhibit substantially higher and more variable latency profiles [14]. The PCCA's latency improvement is measured against this baseline, representing the reduction attributable to prompt-level caching rather than to changes in the serving infrastructure.

7.3 Results

Token Consumption

System	Total Tokens
Baseline	27,000,000
Cached	9,600,000
Reduction	**64%**

Table 5: Token Consumption Comparison Between Baseline and Cached Systems

The 64% reduction reflects the proportion of each request's tokens that were served from the cache rather than transmitted live. Given that the static and shared context together account for 2,500 of the 2,700 tokens per request, the theoretical maximum cache hit savings after the first request in each segment's lifecycle is approximately 92.6%; the measured 64% reflects cache cold-start effects across new segment combinations encountered during the evaluation.

Cost Reduction

Baseline inference cost over the 10,000-request workload was $81.00. With caching enabled, the cost fell to $28.80, a reduction of 64%, directly proportional to the token savings. At production volumes an order of magnitude larger, these savings compound into the tens of thousands of dollars per month for a single deployment.

Latency Improvement

System	Average Latency
Baseline	2.4 seconds
Cached	1.6 seconds
Improvement	**33%**

Table 6: Request Latency Comparison Between Baseline and Cached Systems

The 33% latency reduction reflects both reduced prompt transmission time and, where the inference endpoint supports it, reduced prefill computation time through attention state reuse [9]. The improvement is consistent with the proportion of each prompt eliminated from live processing [15].

8. Discussion and Future Directions

The results demonstrate that prompt context caching is a practical and deployable optimization with substantial impact across all measured dimensions. The architecture requires no changes to the underlying model, no modification of the LLM provider's API, and no degradation of response quality. It is infrastructure-agnostic and can be layered onto existing pipelines with relatively modest engineering effort.

The primary open challenges are cache invalidation for semantically drifting content, extension to approximate-match caching for paraphrased but functionally equivalent segments, and granularity optimization for complex RAG-assembled contexts. Embedding-based semantic similarity search over cached segments would extend hit rates beyond exact-match scenarios but introduces retrieval complexity and approximate-match risk that require careful evaluation [11].

Hierarchical prompt reuse, where sub-segments of a static context are cached independently and assembled in different combinations, represents another extension that could further improve hit rates for deployments where static content is partially but not entirely shared across request types.

Conclusion

Prompt Context Caching architecture establishes a practically deployable and theoretically grounded

solution to one of the most persistent cost inefficiencies in large enterprise language model deployments. The core insight that prompts are composite structures in which static and semi-static content accounts for the dominant share of tokens per request enables a segmentation-first design that separates cacheable content from genuinely dynamic input. The architecture assigns cryptographic fingerprints to reusable pieces and lands them out of a distributed in-memory store, leading to no additional token transmission at scale, no changes in model behavior, and no changes in underlying inference infrastructure. These performance benefits can be directly applied to prompt-level caching in combination with retrieval-augmented generation pipelines, making them one of the most token-intensive patterns in the current AI implementation, where commonly accessed documents are a significant and often neglected cost center. Lifecycle management of cache across TTL expiration, version-based invalidation, and tenant-based security workload through a combination of TTL expiration, version-driven invalidation, and tenant-aware security workload guarantees that the advantage of increased efficiency is long-term and does not come at the cost of content freshness or data integrity. The above-mentioned gains in simulated enterprise workload performance indicate that the architecture provides significant improvements in token efficiency, cost, throughput, and request latency, all at the same time. Looking forward, the most consequential extensions involve embedding-based semantic caching, enabling approximate-match retrieval for paraphrased but functionally equivalent segments, and hierarchical reuse strategies that cache sub-segments of complex composed prompts independently. As large language model integration deepens across enterprise domains, infrastructure-level optimizations that reduce cost without constraining capability will define the boundary between experimental deployment and sustainable production operation. Prompt context caching is positioned to become a foundational layer in that infrastructure.

References

[1] Tom B. Brown et al., "Language Models are Few-Shot Learners," Computation and Language, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[2] Jacob Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), 2019. [Online]. Available: https://aclanthology.org/N19-1423/

[3] Patrick Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," Advances in Neural Information Processing Systems 33 (NeurIPS 2020), 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[4] Shamane Siriwardhana et al., "Improving the Domain Adaptation of Retrieval Augmented Generation (RAG) Models for Open Domain Question Answering," Transactions of the Association for Computational Linguistics, 2023. [Online]. Available: https://aclanthology.org/2023.tacl-1.1/

[5] Brian Lester et al., "The Power of Scale for Parameter-Efficient Prompt Tuning," Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021. [Online]. Available: https://aclanthology.org/2021.emnlp-main.243/

[6] Vincent Abbott et al., "Accelerating Machine Learning Systems via Category Theory: Applications to Spherical Attention for Gene Regulatory Networks," Category Theory (math.CT); Machine Learning (cs.LG); Molecular Networks, 2025. [Online]. Available: https://arxiv.org/abs/2505.09326

[7] Nelson F. Liu et al., "Lost in the Middle: How Language Models Use Long Contexts," Transactions of the Association for Computational Linguistics, 2024. [Online]. Available: https://aclanthology.org/2024.tacl-1.9/

[8] Dan Boneh and Victor Shoup, “A Graduate Course in Applied Cryptography," Version 0.6, 2023. [Online]. Available: https://toc.cryptobook.us/

[9] Bo Peng et al., "RWKV: Reinventing RNNs for the Transformer Era," Findings of the Association for Computational Linguistics: EMNLP 2023, 2023. [Online]. Available: https://aclanthology.org/2023.findings-emnlp.936/

[10] Joshua Ainslie et al., "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints," Proceedings of the 2023

Conference on Empirical Methods in Natural Language Processing, 2023. [Online]. Available: https://aclanthology.org/2023.emnlp-main.298/
[11] Nils Reimers and Iryna Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), 2019. [Online]. Available: https://aclanthology.org/D19-1410/
[12] Akari Asai et al., "Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection," 2024. [Online]. Available: https://openreview.net/forum?id=hSyW5go0v8
[13] Woosuk Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention," SOSP '23: Proceedings of the 29th Symposium on Operating Systems Principles, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3600006.3613165
[14] Noam Shazeer, "Fast Transformer Decoding: One Write-Head is All You Need," arXiv preprint, arXiv:1911.02150, 2019. [Online]. Available: https://arxiv.org/abs/1911.02150
[15] Tri Dao et al., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," Advances in Neural Information Processing Systems 35 (NeurIPS 2022), 2022. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html