



DEPARTMENT OF MANAGEMENT SCIENCE AND
TECHNOLOGY,
ATHENS UNIVERSITY OF ECONOMICS AND
BUSINESS

Master's Thesis in Business Analytics

**Prompt Caching Techniques for Optimizing Large
Language Models**

Date 15/07/2025

Christos Lyssas
P2822215

Supervisor's Name (AUEB):

Georgios Papastefanatos



Acknowledgements

I would like to express my sincere gratitude to my advisor, Dr. Georgios Papastefanatos, for his invaluable guidance and support throughout my research. His insights and encouragement have been instrumental in shaping this thesis.

I would also like to thank Dr. Manolis Terrovitis and Prof. Damianos Chatziantoniou for their valuable feedback and constructive comments as members of the examination committee.

Special thanks go to Stavros Maroulis for his support and involvement during this process.

I am grateful to the Athens University of Economics and Business for providing the necessary resources and environment that made this research possible.

Finally, I would like to thank my partner, who continuously pushed me to achieve my best, as well as my family and friends for their unwavering support and encouragement throughout this journey. Their belief in me has been a constant source of motivation.



Abstract

This thesis investigates prompt caching techniques as a method for optimizing large language models (LLMs), which have become increasingly prevalent in natural language processing tasks. Despite their impressive capabilities, LLMs often encounter challenges related to computational efficiency and response latency, particularly in real-time applications. This research proposes a framework for implementing prompt caching, which involves storing previously generated prompts and their outputs to minimize redundant computations and improve response times.

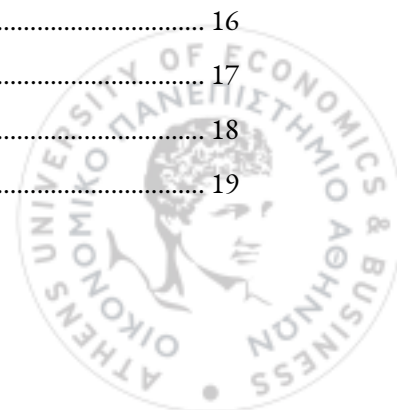
I conduct a comprehensive literature review to contextualize my work within the existing body of research, highlighting relevant studies that address optimization techniques for LLMs. My findings reveal that prompt caching can significantly enhance both the efficiency and accessibility of LLMs, paving the way for their broader application. This thesis contributes to the field by providing a detailed analysis of prompt caching strategies and their implications for LLM performance, alongside a discussion of other optimization techniques that can complement this approach.



Acknowledgements

Abstract

1.	Introduction.....	1
2.	Literature Review.....	2
2.1.	Overview of Large Language Models (LLMs).....	2
2.2.	General Optimization Techniques for LLMs.....	3
2.2.1.	Quantization.....	3
2.2.2.	Pruning.....	3
2.2.3.	Knowledge distillation.....	3
2.3.	Prompt Caching Techniques.....	3
2.3.1.	Definitions and methodologies.....	4
2.3.2.	Previous studies.....	4
2.3.3.	How prompt caching fits into the broader context of LLM optimization.....	4
2.4.	Related Work.....	4
2.4.1.	Semantic Prompt Caching.....	5
2.4.2.	Adaptive KV Cache Compression.....	5
2.4.3.	Cache Augmented Generation.....	6
3.	Methods and Algorithms.....	7
3.1.	Methodology.....	7
3.2.	Caching Strategies Implemented.....	8
3.2.1.	Semantic Similarity Caching.....	8
3.2.2.	Normalized Prompt Caching.....	9
3.3.	Algorithmic Workflows.....	9
3.4.	Concrete Examples from LongBench.....	10
4.	Experimental Design.....	15
4.1.	Data Sources.....	15
4.2.	Models Used.....	16
4.3.	Metrics for Evaluations.....	16
4.4.	Design Decisions.....	17
4.5.	Workload Description and Prompt Selection.....	18
4.5.1.	Cache Warm-Up and Prompt Variation.....	19



5.	Results.....	20
5.1.	Findings.....	20
5.1.1.	Performance improvements.....	21
5.1.2.	Comparisons with baseline models without caching.....	21
5.2.	Visualizations.....	23
6.	Discussion.....	34
6.1.	Interpretation of Results.....	35
6.2.	Limitations.....	35
6.3.	Future Work.....	36
6.4.	Threats to Validity.....	37
7.	Conclusion.....	39
	List of Figures.....	42
	List of Tables.....	42
	References.....	43



1. Introduction

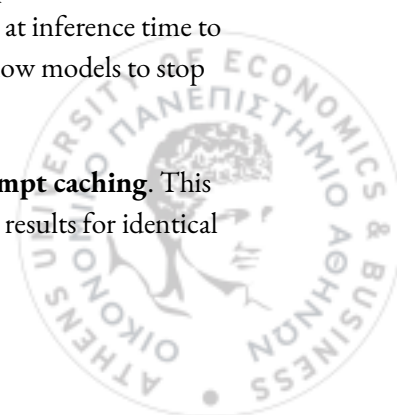
Large Language Models (LLMs) have revolutionized the field of Natural Language Processing (NLP) by enabling machines to understand, generate, and manipulate human language with unprecedented accuracy and fluency. These models, powered by deep learning techniques, are trained on vast datasets, allowing them to capture complex linguistic patterns and contextual nuances. The applications of LLMs span various domains, including chatbots, content generation, translation services, and even aiding in scientific research. Their ability to process and analyze large volumes of text data has made them invaluable tools in both commercial and academic settings, driving advancements in artificial intelligence and enhancing human-computer interaction.

Despite their remarkable capabilities, LLMs face a number of critical performance and computational challenges that limit their practicality and scalability. The foremost of these is their extremely high resource consumption. Modern LLMs often consist of billions—or even trillions—of parameters, requiring vast computational power and memory during both training and inference. Running these models in real-time applications demands high-end GPU or TPU infrastructure, which incurs significant costs in terms of hardware, energy consumption, and environmental impact. Furthermore, inference latency becomes a major bottleneck, especially for long-form inputs or when serving large numbers of users simultaneously. These demands make it difficult to deploy LLMs efficiently in low-resource or time-sensitive environments such as mobile devices, embedded systems, or real-time conversational agents.

Beyond compute constraints, LLMs also suffer from issues of factual inaccuracy, or “hallucinations,” where the models generate information that appears plausible but is incorrect or fabricated. Maintaining up-to-date knowledge is also a challenge, as static pre-trained models can quickly become outdated, especially in fast-moving domains. Additionally, the complex and opaque nature of LLM architectures makes interpretability difficult, raising concerns around fairness, accountability, and ethical deployment. These issues underline the urgent need for strategies that can make LLMs more efficient, reliable, and adaptable without compromising their core capabilities.

In response to these challenges, several research directions have emerged to optimize LLMs for practical use. One key area involves **model compression techniques**, such as pruning (removing non-essential weights), quantization (reducing numerical precision), and knowledge distillation (training smaller models to mimic larger ones), which aim to reduce model size and computational burden. **Efficient architecture design** is another area of focus, with models like **Mixture of Experts (MoE)** and **sparse transformers** activating only portions of the model per input, thereby saving resources. Additionally, **hardware-aware optimizations**—including tensor and pipeline parallelism, operator fusion, and improved scheduling algorithms—have enabled better performance across high-performance computing systems. Other promising techniques include **Retrieval-Augmented Generation (RAG)**, which integrates external knowledge sources at inference time to reduce the burden on the model's internal parameters, and **early-exit strategies**, which allow models to stop computation once a sufficient level of confidence is reached.

Among these emerging solutions, one particularly promising and practical direction is **prompt caching**. This technique addresses inefficiencies in inference by storing and reusing previously computed results for identical



or semantically similar prompts. In scenarios where the same or similar queries are issued repeatedly—such as in customer support systems, educational tools, or document summarization pipelines—prompt caching can dramatically reduce redundant computation, decrease latency, and lower energy consumption. It thus plays a crucial role in making LLMs more scalable and responsive in real-world deployments.

The objective of this paper is to investigate and analyze prompt caching techniques as a means to optimize Large Language Models (LLMs). Given the increasing demand for efficient and responsive AI systems, prompt caching emerges as a critical strategy to enhance the performance of LLMs during inference. This study aims to explore various prompt caching methodologies including their implementation, benefits, and potential drawbacks, as highlighted in recent literature. By examining techniques such as those presented in the works of Claude and FINCH, this paper seeks to provide a comprehensive understanding of how prompt caching can reduce latency, lower computational costs, and improve the overall user experience with LLMs. Furthermore, the research will evaluate the effectiveness of these techniques in maintaining the accuracy and reliability of LLM outputs, ensuring that optimizations do not compromise the quality of generated content. Ultimately, this paper aims to contribute valuable insights into the development of more efficient LLMs through the strategic application of prompt caching techniques.

2. Literature Review

2.1. Overview of Large Language Models (LLMs)

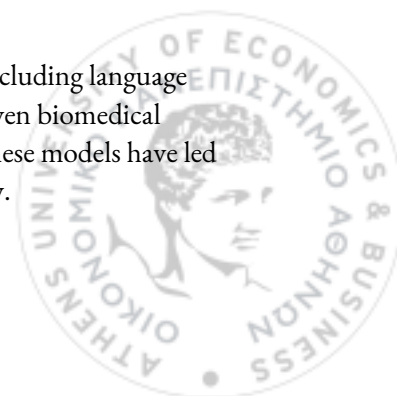
Large Language Models (LLMs) are advanced neural network-based architectures designed to understand and generate human-like text. They have evolved significantly over the past decade, transitioning from traditional n-gram models to sophisticated architectures like the Transformer, introduced by Ashish Vaswani (*Vaswani et al., 2017*). This architecture employs self-attention mechanisms, allowing LLMs to process and generate text with remarkable fluency and coherence.

The Transformer architecture relies on self-attention mechanisms, which allow models to efficiently capture long-range dependencies in text. Unlike older sequence models like RNNs (Recurrent Neural Networks) and LSTMs (Long Short-Term Memory networks), Transformers process entire sequences simultaneously rather than sequentially. This fundamental change has drastically improved performance in NLP tasks, paving the way for highly scalable and context-aware LLMs (*Vaswani et al., 2017*).

Notable examples of LLMs include BERT (Bidirectional Encoder Representations from Transformers), introduced by Devlin in 2018 (*Devlin & Chang, 2018*), which excels in understanding context, and GPT-3 (Generative Pre-trained Transformer), developed by OpenAI, is known for its impressive text generation capabilities. These models are typically trained on vast datasets using unsupervised learning techniques, followed by fine-tuning for specific tasks, such as sentiment analysis or question answering (*Luobe et al. 2024*).

LLMs have found applications across various domains, including conversational agents, language translation, and even code generation.

As LLMs continue to evolve, their applications have expanded across multiple domains, including language translation, customer support automation, content creation, software development, and even biomedical research. However, the increasing computational demands and high operational costs of these models have led researchers to explore various optimization techniques to improve efficiency and scalability.



2.2. General Optimization Techniques for LLMs

As LLMs have become more prevalent, the need for optimization techniques to enhance their performance and reduce operational costs has gained significant attention. Several key strategies have emerged in the literature.

2.2.1. Quantization

Quantization is a hardware-aware optimization technique that reduces the precision of model weights, allowing for smaller model sizes and faster inference times. By converting model parameters from high-precision floating-point representations (e.g., FP32) to lower-bit formats such as int8 (8-bit integers), quantization can significantly reduce memory consumption and computational requirements while maintaining near-original performance levels (Dettmers et al., 2022).

There are two types of quantization:

- **Post-training quantization (PTQ):** Converts model weights after training, making it a convenient approach for optimizing pre-trained models.
- **Quantization-aware training (QAT):** Integrated quantization into the training process, allowing the model to adjust to lower precision and minimize accuracy loss.

2.2.2. Pruning

Pruning involves removing less important parameters or connections within the model to streamline its architecture. This technique can lead to reduced model size and improved inference speed while maintaining accuracy. Recent studies have shown that structured pruning can effectively compress LLMs, making them more suitable for deployment in resource-constrained environments (Robinson et al., 2024).

There are two types of pruning methods:

- **Unstructured pruning:** Removes individual weights based on their significance.
- **Structured pruning:** Removes entire neurons, attention heads, or layers to create a more compact model.

2.2.3. Knowledge distillation

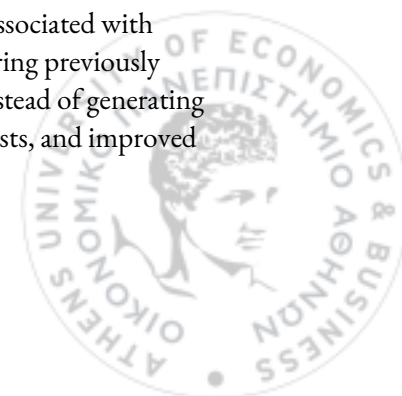
Knowledge distillation is a process where a smaller “student” model is trained to replicate the behavior of a larger “teacher” model. This approach allows for the transfer of knowledge from a high-performing model to a more efficient one, enabling the smaller model to achieve competitive performance on various tasks while requiring fewer resources (Robinson et al., 2024).

The process involves:

- Training a large, high-performance model on a task.
- Using the teacher model’s outputs as soft labels to guide the student model during training.
- The student model learns to mimic the teacher model’s decision-making, allowing it to achieve comparable accuracy with lower computational demands.

2.3. Prompt Caching Techniques

Prompt caching is an innovative optimization technique that addresses the inefficiencies associated with processing repetitive or similar prompts in LLM applications. This technique involves storing previously computed responses to specific prompts, allowing the model to retrieve these responses instead of generating them anew. The benefits of prompt caching include reduced latency, lower operational costs, and improved scalability (Gim et al., 2024).



2.3.1. Definitions and methodologies

Prompt caching is an innovative optimization technique that addresses the inefficiencies associated with processing repetitive or similar prompts in LLM applications. Although data management and web communities offer a rich array of caching strategies (Maroulis et al., 2024), the domain of caching specifically tailored to enhance the efficiency of large language models remains largely underexplored. When a similar prompt is encountered, the system can quickly retrieve the cached response, significantly speeding up the inference process (Chen et al., 2024). There are several types of prompt caching techniques, each with a unique approach to matching and retrieval:

- **Semantic Similarity Caching:** Uses embedding similarity to return cached responses for semantically similar prompts, allowing for more flexible reuse.
- **Key-value (KV) Cache Reuse:** Stores the internal attention key and value tensors during generation and reuses them for repeated tokens or prefixes, significantly accelerating time-to-first-token performance.

2.3.2. Previous studies

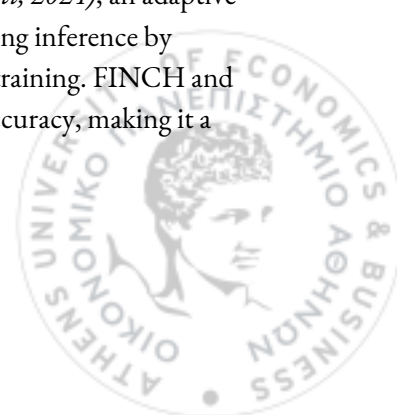
Recent studies have demonstrated the effectiveness of prompt caching in various contexts. For example, Anthropic's Claude model has reported up to 90% cost reduction and 85% latency reduction for long prompts through the implementation of prompt caching (Prompt Caching with Claude, 2024). Additionally, research on modular attention reuse has shown that reusing attention states across different prompts can lead to substantial improvements in time-to-first-token (TTFT) latency, with reductions ranging from 8x for GPU-based inference to 60x for CPU-based inference (Chen et al., 2024).

2.3.3. How prompt caching fits into the broader context of LLM optimization

Prompt caching complements other optimization techniques by addressing the specific challenge of repetitive computations in LLMs. While quantization, pruning and knowledge distillation focus on reducing model size and improving efficiency at the architectural level, prompt caching enhances the operational efficiency of LLMs during inference. By minimizing redundant processing, prompt caching allows LLMs to handle higher volumes of requests without sacrificing performance, making it a critical component of modern LLM deployment strategies (Gim et al., 2024)

2.4. Related Work

As more and more people adopt large language models (LLMs), it is becoming extremely important to optimize these models for computational cost, latency, and scalability. Some strategies are being developed to deal with these computational costs, latencies and scaling issues. One such strategy which is drawing attention is semantic prompt caching. It aims to minimize the repeated calculation by reusing outputs for semantically similar prompts. But that is just one piece of all optimizations. Preble, a distributed LLM serving platform (Srivatsa et al., 2024), further optimizes inference by reusing prompt caching computations across requests, significantly reducing latency (1.5x to 14.5x avg, 2x to 10x p99). Additionally, FINCH (Corallo & Papotti, 2024), an adaptive KV cache compression method, has been introduced to reduce memory consumption during inference by dynamically selecting and compressing stored attention values without requiring model retraining. FINCH and Preble together enable LLMs to process longer inputs efficiently while maintaining high accuracy, making it a key advancement in LLM scalability.



2.4.1. Semantic Prompt Caching

GPTCache is an open-source framework that caches semantic prompts to improve performance of large language model (LLM) queries (Bang, 2023). GPTCache stores prompts' embeddings with their responses, using vector similarity search to find previously cached responses for semantically similar queries. It minimizes needless requests to LLMs, saving on money and time. It begins with a **pre-processor** that formats the input requests, removes unnecessary prompt information, and compresses data to enhance cache effectiveness. The **adapter** converts LLM requests into a cache-compatible format, ensuring seamless integration. The **embedding generator** transforms user queries into vector embeddings, enabling similarity-based retrieval. At the core, the **cache manager** stores user requests and responses, manages vector storage, and clears outdated data using eviction policies like LRU (least recently used) and FIFO (first-in first-out). The similarity evaluator then identifies the **Top-K most relevant cached responses** using an **ALBERT-based** similarity function. Finally, the post processor either returns the closest cached response or generates a new one if no match is found, storing it for future use.

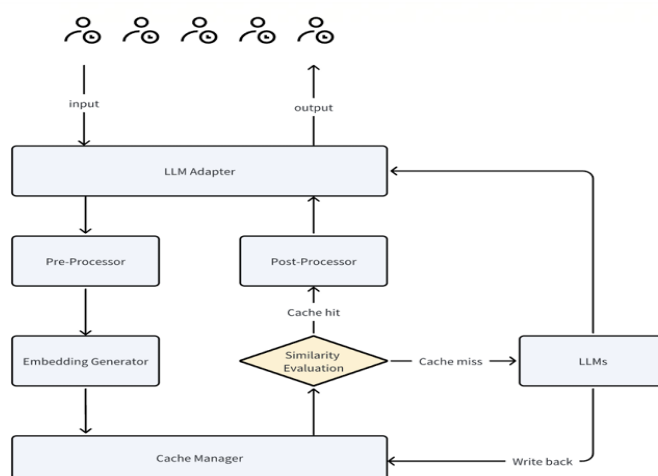


Figure 1 - GPTCache: The architecture comprises of six core components: adapter, pre-processor, embedding generator, cache manager, similarity evaluator, and post-processor.

2.4.2. Adaptive KV Cache Compression

Large Language Models (LLMs) are incredibly powerful, but they come with a big problem - they use a ton of memory, especially when dealing with long prompts. That is where FINCH (Corallo & Papotti, 2024) steps in. Instead of storing every detail from a prompt, FINCH figures out what is actually important and keeps only that, cutting down memory use while still making sure the model works just as well. It does this without any extra training or fine-tuning. Tests show that FINCH can compress memory use by up to 93x, all while keeping responses just as accurate. In fact, even at 3.76x compression, FINCH maintains 90% of the original model's accuracy. It is a huge step forward for making AI assistants, research tools, and other LLM applications faster, cheaper, and more efficient. In **Figure 2**, during the **Prefill stage**, each document chunk (represented as blue squares) is processed alongside the **input prompt** (yellow square). As the model analyzes the text, it selects the most relevant key-value pairs (solid squares) and stores them in the cache for future steps. These **cached values** (shown in violet) help process subsequent chunks efficiently. Once all chunks have been processed, the model enters the **Generation stage**, where it uses the compressed cached information to generate

a response. The white square represents the reserved space where output tokens are produced (Corallo & Papotti, 2024).

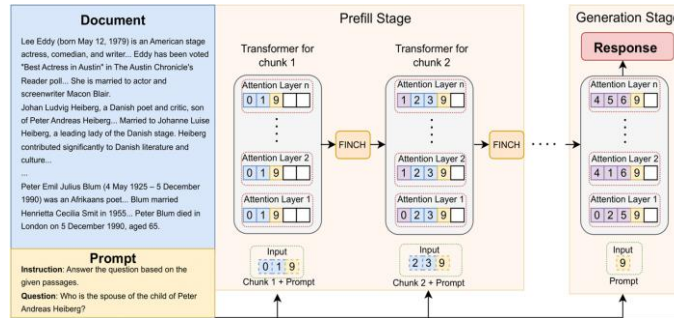


Figure 2 - FINCH processes input documents that are too large for the model's context by breaking them into chunks.

2.4.3. Cache Augmented Generation

Recent research has explored alternatives to Retrieval-Augmented Generation (RAG) by leveraging the growing capabilities of long-context large language models (LLMs). A notable contribution in this area is the work by Chan et al. (2024), which proposes Cache-Augmented Generation (CAG) as a retrieval-free alternative to traditional RAG pipelines. While RAG systems dynamically fetch and incorporate external documents during inference—often introducing latency, complexity, and the risk of retrieval errors—CAG circumvents these issues by preloading all relevant knowledge into the model's extended context window and caching its key-value (KV) states ahead of inference. This approach enables the model to generate accurate, context-rich responses without runtime retrieval, effectively transforming knowledge-intensive tasks into a pure generation problem. By offloading retrieval to a pre-processing step and utilizing the LLM's large context capacity (e.g., 128k tokens), CAG streamlines system design and improves inference efficiency, particularly when the knowledge base is bounded and manageable in size. Empirical evaluations using QA datasets like SQuAD and HotPotQA demonstrated that CAG outperforms or matches RAG baselines (BM25 and dense retrieval via OpenAI Indexes) in both accuracy and latency across small, medium, and large context configurations. These findings suggest that, for many real-world applications, especially those with fixed or static corpora, retrieval-free paradigms like CAG may offer a more scalable, reliable, and maintainable solution than traditional RAG pipelines.

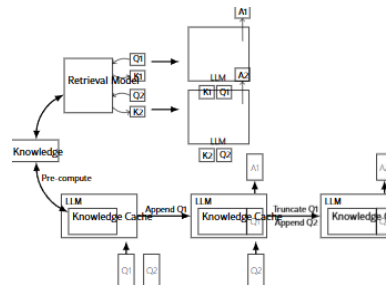


Figure 3 - Cache Augmented Generation architecture.



3. Methods and Algorithms

This section presents the contributions of this thesis, detailing the prompt caching methods implemented and evaluated to optimize large language model inference. Building upon the literature review, we consider caching strategies that address key challenges in identifying semantically similar prompts and maximizing cache hit rates through normalized prompt representations. Our approach combines semantic similarity analysis with efficient cache management to create a comprehensive framework that adapts to diverse prompt patterns. We describe the specific caching strategies implemented, detail the algorithmic workflows that govern cache operations, and provide concrete examples from the LongBench dataset to demonstrate practical applications. These methods form the technical foundation for the experimental evaluation presented in the following section.

3.1. Methodology

A practical way to reduce resource consumption and latency is by minimizing the number of LLM calls, which can be achieved through caching. This technique has long been used in various fields to enhance efficiency by storing and reusing previously computed results. In the context of LLMs, prompt caching involves storing prompt-response pairs in a limited-size cache, allowing the system to quickly retrieve responses for repeated or similar queries instead of recomputing them (Zhu et al., 2024). While various semantic search and embedding-based methods have been explored as potential solutions for improving cache hit rates, accurately representing the semantic meaning of a prompt remains a challenge. Additionally, semantically similar prompts do not always require identical responses, highlighting the need for a specialized vector embedding approach. This approach should not only improve the efficiency of finding similar prompts but also enhance the prediction accuracy of whether two prompts can be answered with the same response.

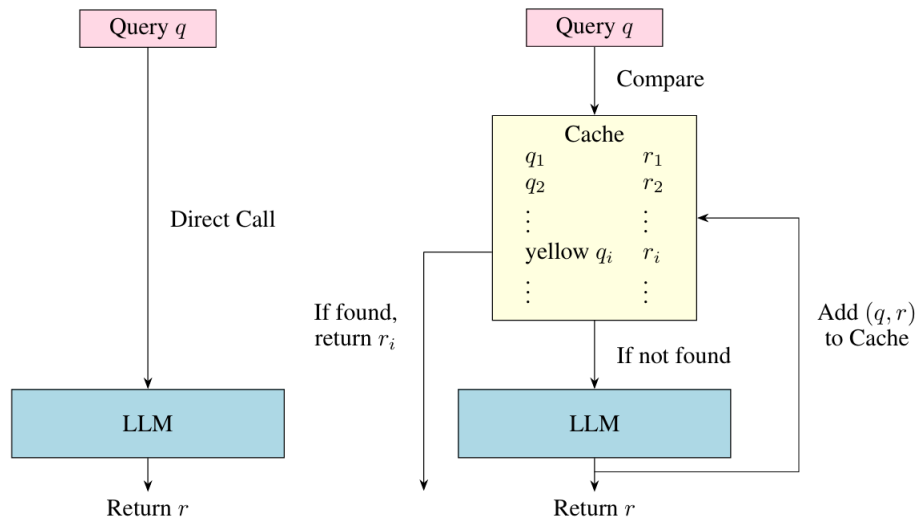


Figure 4 - The procedure of calling LLMs with or without cache.



With a cache in place, previously processed prompt-response pairs can be stored for future use. When a new prompt is received, the system can search the cache for a semantically similar prompt. If a match is found, the stored response can be retrieved and reused, eliminating the need for an additional LLM call.

3.2. Caching Strategies Implemented

This study will evaluate how caching strategies impact Llama 3.1's and Llama 3.2's performance, particularly in areas such as response time, memory consumption, and computational efficiency. It will examine No Cache Performance to measure query latency when caching is disabled and Normalized Prompt Cache (Based on Cache-Augmented Generation logic). Additionally, it will explore Semantic Similarity Caching, which retrieves stored responses based on semantic meaning rather than exact text matches, further optimizing memory usage and processing time.

3.2.1. Semantic Similarity Caching

Semantic Similarity Caching introduces an additional layer of optimization by focusing not on token-by-token repetition but on the semantic meaning of prompts. In this technique, prompts are embedded into a dense vector space using a sentence encoder, which transforms natural language text into high-dimensional numerical representations that capture semantic relationships and contextual meaning. Similarity metrics, most commonly cosine similarity, are then employed to determine the proximity of a new prompt to previously cached queries. Cosine similarity measures the cosine of the angle between two vectors in a multi-dimensional space, providing a metric that ranges from -1 to 1, where:

- **1** indicates identical semantic direction (perfect similarity)
- **0** indicates orthogonal vectors (no similarity)
- **-1** indicates opposite semantic direction (completely dissimilar)

The cosine similarity between two vectors A and B is calculated as:

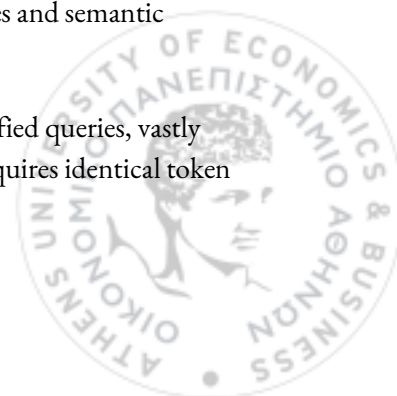
$$\text{cosine_similarity}(A, B) = (A \cdot B) / (\|A\| \times \|B\|)$$

Where:

- $A \cdot B$ is the dot product of vectors A and B : $\sum(A_i \times B_i)$
- $\|A\|$ is the magnitude (Euclidean norm) of vector A : $\sqrt{\sum(A_i^2)}$
- $\|B\|$ is the magnitude (Euclidean norm) of vector B : $\sqrt{\sum(B_i^2)}$

When a new input is found to be semantically similar to an existing cached query beyond a defined threshold (80% similarity), the model can skip full inference and return the cached response. This threshold represents the minimum cosine similarity score required for cache reuse, balancing between cache hit rates and semantic accuracy.

This method allows the system to effectively generalize across paraphrased or slightly modified queries, vastly improving efficiency in realistic, noisy usage scenarios. Unlike exact-match caching that requires identical token sequences, Semantic Similarity Caching can recognize that prompts like:



- "What is the capital of France?"
- "Can you tell me France's capital city?"
- "What city serves as the capital of France?"

All represent the same semantic intent, despite different surface-level expressions. The embedding vectors for these prompts would have high cosine similarity scores, enabling cache reuse and significant performance improvements. The sentence encoder transforms each prompt into a dense vector (typically 384, 512, or 768 dimensions depending on the model), where semantically similar prompts are positioned closer together in the vector space. The cosine similarity calculation then provides a normalized measure of this semantic proximity, making it robust to variations in sentence length and structure while focusing purely on semantic content and meaning.

3.2.2. Normalized Prompt Caching

A second technique, **Normalized Prompt Caching**, is a simple approach implemented as part of this work, where the system caches entire prompt-response pairs during inference. Upon receiving a new prompt, the system first applies basic normalization—such as lowercasing and removing extraneous punctuation—to reduce trivial variations. It then checks whether the normalized prompt has an exact match in the cache. If a match is found, the cached response is returned directly, avoiding further computation. If no match exists, the model performs full inference and stores the resulting response for future reuse. Unlike semantic caching, this method does not generalize across similar but non-identical inputs; its efficiency benefits are limited to scenarios involving repeated prompts that normalize to the same form.

3.3. Algorithmic Workflows

Caching mechanisms were implemented via three core workflows: Semantic Cache Similarity and Normalized Prompt Caching. These are formalized below:

Algorithm 1: Semantic Cache Lookup Process

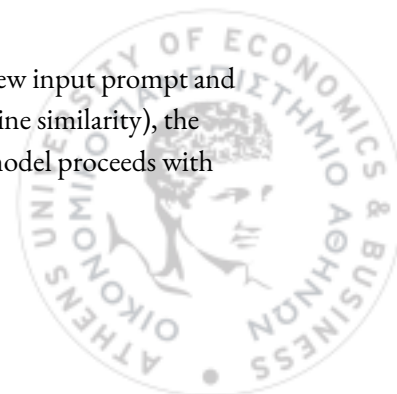
Input: Prompt (P), threshold

Output: Cached response or None

- 1: Retrieve all stored prompts P_i from Semantic Cache (S)
- 2: For each P_i in S:
- 3: Compute cosine similarity between P and P_i
- 4: If similarity \geq threshold:
- 5: Return corresponding cached response R_i
- 6: If no match is found:
- 7: Return None

Explanation:

This algorithm retrieves semantically similar responses from the cache by embedding the new input prompt and comparing it to cached prompts. If a prompt with sufficient similarity is found ($\geq 80\%$ cosine similarity), the associated response is immediately returned. Otherwise, a cache miss is recorded, and the model proceeds with normal inference.



Algorithm 2: Normalized Prompt Caching Workflow

Input: Prompt (P)

Output: Model response R or Cached response

- 1: Normalize prompt P to obtain clean key K
- 2: If cache contains key K:
- 3: Return cached response R_k
- 4: Else:
- 5: Run model inference with prompt P
- 6: Store new response R in cache under key K
- 7: Return response R

Explanation:

Normalized Prompt Cache first normalizes incoming prompts (e.g., by lowercasing and stripping punctuation) to generate consistent cache keys. It checks whether a response for this key already exists in the cache. If found, the cached response is returned immediately. If not, the system performs full model inference and stores the result in the cache for future reuse.

3.4. Concrete Examples from LongBench

To assess how **Normalized Prompt Caching** performs, we present a general example using a small set of diverse prompts from the LongBench dataset. These were evaluated over a baseline run (no cache) and four subsequent cached runs. Below, we show the prompt list along with their cache status, followed by execution time comparisons.

The following five prompts were executed in the **baseline run**, where caching was disabled and the prompts were encountered for the first time. As expected, all resulted in full inference and initial cache population.



Baseline - No Cache

Queries	Prompt	From Cache
1	quick question which case was brought to court first miller v california or gates v collier	False
2	hey the actor that plays phileas fogg in around the world in 80 days costarred with gary cooper in a 1939 goldwyn productions film...	False
3	hey i was wondering prior to playing for michigan state keith nichol played football for a school situated inwards what city	False
4	i need help with this gary l bennett was a part of the space missions that get a primary destination of what celestial body	False
5	quick question was atom egoyans biggest commercial success on stage or on film	False

Table 1 - LongBench examples, Normalized Prompt Caching, Baseline run (No cache)



Normalized Prompt Caching Hits with Variation Types

Queries	Variation Type	Prompt	From Cache
1	Exact Repetition	quick question which case was brought to court first miller v california or gates v collier	True
2	Exact Repetition	hey the actor that plays phileas fogg in around the world in 80 days costarred with gary cooper...	True
3	Exact Repetition	hey i was wondering prior to playing for michigan state keith nichol played football for a school situated inwards...	True
4	Exact Repetition	i need help with this gary l bennett was a part of the space missions that get a primary destination...	True
5	Exact Repetition	quick question was atom egoyans biggest commercial success on stage or on film	True

Table 2 - LongBench examples, Normalized Prompt Caching, Hits



To explore how a **Semantic Similarity Caching** strategy performs, we present a general example using a small set of base prompts from the LongBench dataset. These were executed in an initial baseline run (no cache) and four subsequent cached runs. Semantic caching uses embedding-based similarity (e.g., cosine similarity) to match incoming prompts with existing cache entries, with a similarity threshold of 0.80 used to determine cache hits.

Below, we present the full list of prompts, their cosine similarity scores against matched entries, and whether caching was successful. This is followed by a summary of execution time for each run.

Baseline - No Cache

The following five prompts were executed in the baseline run, where caching was disabled. All resulted in full inference and population of the semantic cache.

Query	Prompt	From Cache
1	Quick question, Which case was brought to court offset Miller v. California or Gates v. Collier ?	False
2	Quick question, The actor that plays Phileas Fogg in "Around the World in 80 Days", co-starred with Gary Cooper in a 1939 Goldwyn Productions film found on a novel by what author?	False
3	Quick question, Prior to playing for Michigan State, Keith Nichol played football for a school located in what city?	False
4	Hey, atomic number 53 was wondering, Gary L. Bennett was a part of the space missions that have a primary destination of what celestial body?	False
5	Quick question, Was Atom Egoyans biggest commercial success on stage or on film?	False

Table 3 - LongBench examples, Semantic Similarity Caching, Baseline run (No cache)



Semantic Similarity Cache Hits (Cosine Similarity Matching)

Query	Prompt	Matched With	Cosine Similarity	From Cache
6	Quick question, Which case was brought to court first Miller v. California or Gates v. Collier ?	Query 1	0.9179	True
7	I need help with this... novel by what author?	Query 2	0.9659	True
8	Prior to playing for Michigan State... placed in what city?	Query 3	0.9644	True
9	Gary L. Bennett was a component of the space missions...	Missed Caching	0.6471	False
10	Was Atom Egoyans biggest commercial success along stage...	Query 5	0.9212	True

Table 4 - LongBench examples, Semantic Similarity Caching, Hits



4. Experimental Design

This section outlines the experimental framework used to evaluate the effectiveness of the prompt caching methods and algorithms presented in the previous section. We describe the datasets selected for testing, the large language models employed in our experiments, and the evaluation metrics used to assess caching performance. The experimental setup details our testing methodology, including the rationale behind key design decisions and the specific workload characteristics that guided our evaluation approach. This comprehensive experimental framework enables rigorous assessment of our caching strategies across diverse scenarios and provides the foundation for the results and analysis presented in subsequent sections.

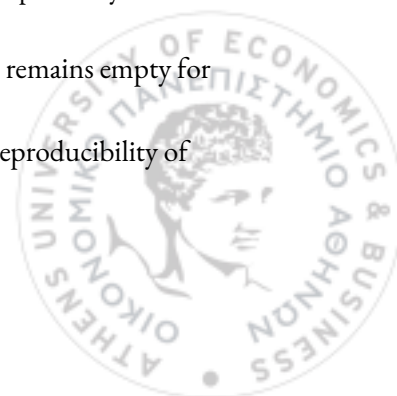
4.1. Data Sources

In this study, we utilize the LongBench benchmark (*Bai et al., 2024*) as the primary dataset for evaluating the performance of various LLM caching strategies. LongBench is a comprehensive and high-quality collection of long-context tasks, specifically designed to stress-test the capabilities of Large Language Models (LLMs) across a wide variety of domains, including open-domain question answering, document summarization, code reasoning, and multi-turn interactions.

The dataset consists of 503 challenging multiple-choice questions, each featuring a long-form input context and a task-oriented instruction or query. This diversity and realism in task design make LongBench particularly well-suited for analyzing caching behavior in real-world LLM workloads. The long-context nature of many queries within LongBench allows for meaningful evaluation of prompt caching especially in scenarios where repeated or semantically similar prompts occur.

Each entry in the LongBench dataset is structured with the following components:

- **Input:** A concise instruction, question, or command that initiates the task. This could be a single-sentence prompt or a short query requiring the model to generate an informed response.
- **Context:** A long passage or collection of information needed to respond to the input. Contexts can include scientific articles, Wikipedia entries, news stories, legal documents, or code files. The richness and diversity of this field make it ideal for testing memory-intensive capabilities of LLMs.
- **Answers:** A list of valid answers or expected outputs corresponding to the input and context. These are used to evaluate model performance objectively across different types of tasks, including extractive, generative, and multi-choice formats.
- **Length:** The total length of the input, context, and answers combined, measured in tokens for English-language samples. This value is useful for analyzing how performance scales with prompt size.
- **Dataset Name:** The specific sub-dataset from which the prompt originates (e.g., GovReport, NarrativeQA, Qasper). This metadata allows researchers to isolate results by domain or task type.
- **Language:** The natural language in which the sample is written. While LongBench primarily features English data, its format allows for multilingual extensions.
- **All Classes:** For classification tasks, this field lists all the possible output classes. It remains empty for generative or non-classification tasks.
- **ID:** A unique identifier assigned to each data entry. This ensures traceability and reproducibility of experimental results.



This structured design provides both breadth and depth in task representation, making LongBench an ideal testbed for exploring the effects of different caching algorithms under realistic LLM usage patterns.

In addition to LongBench, supplementary datasets may also be introduced to further validate results and test specific hypotheses related to caching hit rates and cache utility. These additional datasets would be chosen based on their relevance, complexity, and ability to enrich the prompt variability landscape, thereby enhancing the comprehensiveness of this study.

4.2. Models Used

In this study, I will utilize Llama 3.1 with 8 billion parameters and Llama 3.2 with 3 billion parameters, both advanced open-weight Large Language Models (LLMs) designed for natural language understanding, generation, and reasoning tasks. As successors to previous Llama (Large Language Model Meta AI) versions, these models incorporate several architectural improvements, including enhanced attention mechanisms, better token efficiency, and longer context handling, making them ideal candidates for evaluating caching strategies. They are built on the Transformer (*Vaswani et al., 2017*) architecture, leveraging self-attention mechanisms to process and generate human-like text efficiently. Trained on diverse datasets, these models perform well across various NLP tasks, including text summarization, question-answering, and conversational AI.

Llama 3.1 and Llama 3.2 were chosen for this study due to several key factors. First, their scalability and efficiency allow for extensive testing of real-time caching strategies while maintaining a balance between computational cost and accuracy. Second, their extended context processing capabilities make them well-suited for evaluating prompt caching, where memory efficiency plays a crucial role. Furthermore, their strong semantic understanding enhances the reliability of cache hit predictions, as caching strategies often depend on identifying semantically similar prompts rather than exact text matches.

Additionally, for semantic caching, we utilize **all-MiniLM-L6-v2**, a sentence transformer designed for efficient and accurate sentence-level semantic similarity calculations. This model plays a crucial role in determining cache hits based on meaning rather than exact wording, improving the efficiency of caching mechanisms in this study.

4.3. Metrics for Evaluations

To ensure a thorough evaluation, this study will analyze execution time, cache hit rates, speedup ratios, cache efficiency scores, and latency distribution across different caching strategies using both Llama 3.1 and Llama 3.2. Specifically:

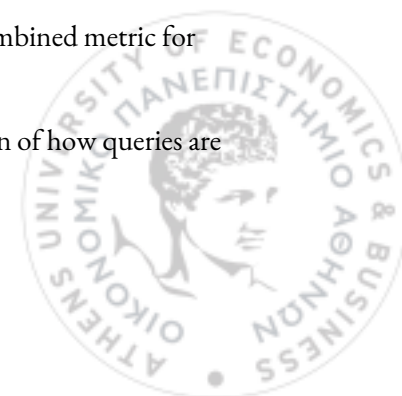
Execution Time: Measures the time taken to process queries with and without caching.

Cache Hit Rate: Calculates the percentage of queries served from cache.

Speedup Ratio: Defined as the ratio of non-cached time to cached time.

Cache Efficiency Score: Computed as the product of hit rate and speedup, offering a combined metric for cache performance.

Query Time Over Time: Visualized using a line plot to provide a graphical representation of how queries are fetched by Cache Type.



By assessing the trade-offs between computational savings and response quality, this research aims to improve the inference efficiency of both Llama 3.1 and 3.2, making them more adaptable for real-world AI applications that demand fast, cost-effective, and scalable language modeling.

4.4. Design Decisions

The design of the experimental framework was guided by a focus on balancing rigor, realism, and reproducibility. Two models were selected for benchmarking: **Llama 3.2 3B** and **Llama 3.1 8B**. The former was chosen as a lightweight model suitable for real-time inference on consumer-grade hardware, while the latter represented a more capable and compute-intensive configuration. This contrast allowed for evaluating caching performance across different scales, helping to identify how model capacity affects both responsiveness and cache efficiency.

To simulate realistic variation in user behavior, controlled perturbations were applied to prompts across **the two caching methods**, ensuring consistency in evaluation conditions. These modifications were designed to reflect the kind of minor linguistic or structural differences commonly seen in natural language inputs, while maintaining semantic coherence.

- **Synonym replacement** was applied incrementally across runs: 0% in the initial run to maintain identical queries with the baseline, then increased to 5%, 10%, 15%, and 20% in the subsequent cached runs to gradually introduce lexical variation.
- **Random prefix injection** added conversational openers (e.g., “Hey, I was wondering” or “I need help with this...”) to simulate natural dialogue variability.

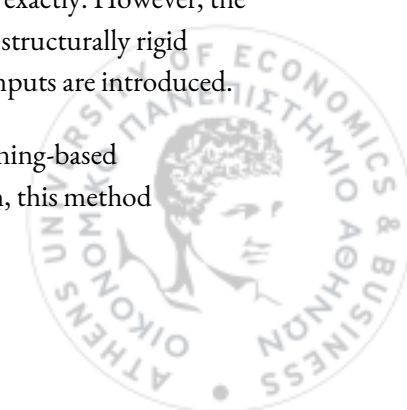
These perturbation strategies were applied dynamically between runs for both **Normalized Prompt Caching** and **Semantic Similarity Caching**. This helped assess the robustness of each caching strategy under varying, yet realistic, prompt formulations.

In addition to the shared perturbation mechanisms above, **Normalized Prompt Caching** applied further preprocessing to enhance string match reliability. Since this method relies on exact or near-exact textual matching, it is especially sensitive to minor formatting differences that don’t affect meaning but do affect literal comparison. To mitigate this, prompts were normalized using two specific transformations:

- **Lowercasing** all characters to eliminate case sensitivity as a mismatch factor,
- **Punctuation removal**, such as stripping commas, periods, and other surface-level symbols.

These normalization steps were exclusive to the Normalized Prompt Caching pipeline. By reducing superficial lexical variation, they increased the chance of repeated prompts matching previous entries exactly. However, the method did not attempt to generalize across reworded or paraphrased inputs. It remains a structurally rigid approach that performs well under consistent formatting but suffers when more diverse inputs are introduced.

In contrast, **Semantic Similarity Caching** was designed to be more flexible, using a meaning-based comparison rather than literal string matching. Instead of relying on textual normalization, this method



transformed each prompt into a semantic vector using a pre-trained sentence transformer. The incoming vector was then compared to previously cached prompts using **cosine similarity**, a common metric for measuring semantic closeness in vector space. If the similarity between the new and cached prompts exceeded a defined threshold, the system treated it as a cache hit and reused the corresponding response.

- The **similarity threshold** was set at **80%**, offering a balance between flexibility and precision. This allowed the system to capture paraphrased or structurally different prompts without being too permissive.

This threshold was carefully chosen. A lower cutoff (e.g., 70%) could result in false positives, incorrectly matching prompts with different meanings. Conversely, a higher threshold (e.g., 90%) would be more conservative, potentially overlooking viable semantic matches and diminishing caching benefits.

Unlike Normalized Prompt Caching, the semantic approach **did not require any punctuation removal or lowercasing**. The embedding process inherently tolerates such variations, making the method well-suited for environments where input diversity is the norm. This embedding-based flexibility allowed Semantic Similarity Caching to perform effectively even when prompts were rephrased, shuffled, or otherwise transformed — without relying on hand-crafted normalization rules.

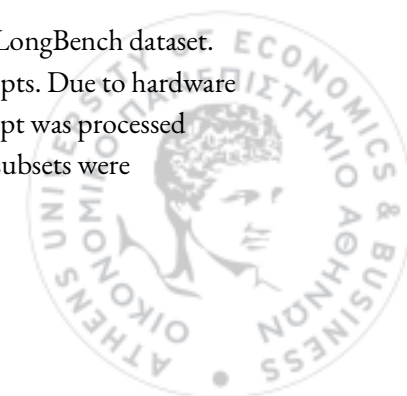
4.5. Workload Description and Prompt Selection

All experiments were conducted on a single desktop machine with the following specifications: a **six-core Intel(R) Core(TM) i5-9400F** processor operating at a base frequency of **2.90GHz**, paired with **16 GB of RAM** and an **NVIDIA GeForce GTX 1660 GPU** equipped with **6 GB of VRAM**. The system utilized a **500GB solid-state drive (SSD)** to ensure high-speed data access and loading of model weights. This hardware configuration reflects a moderately powerful consumer setup, representative of systems used in many research and development contexts.

The experimental workload was designed to systematically evaluate and compare the performance of two distinct caching strategies: **Normalized Prompt Caching** and **Semantic Similarity Cache**. Each strategy was tested across six runs — one baseline (uncached) run followed by five cached runs — to capture both “cold” and progressively “warmer” cache states and assess system behavior under repeated usage.

The prompts were drawn from the **LongBench** dataset, a benchmark designed for long-form language model tasks. Prompt selection was **deterministic**: when the number of prompts to be evaluated was specified (e.g., 5, 10, 20), the system selected the **first n prompts in the dataset** based on their original ordering. This deterministic approach ensured consistency and repeatability across experiments, enabling fair comparison across different caching methods under identical workload conditions.

The experiments evaluated caching strategies using subsets of prompts sampled from the LongBench dataset. For the Llama 3.2 model, three dataset sizes were tested: 50, 100, and 200 individual prompts. Due to hardware limitations, the larger Llama 3.1 model was evaluated using only 200 prompts. Each prompt was processed sequentially (i.e., one at a time), simulating a typical user-querying scenario. The selected subsets were representative of the broader LongBench dataset in terms of structure and content.



4.5.1. Cache Warm-Up and Prompt Variation

To realistically evaluate the robustness and effectiveness of caching strategies in real-world conditions, the experiments deliberately avoided using **verbatim prompt reuse** between cached runs. Instead, prompt inputs were subjected to a series of **controlled linguistic perturbations**. These transformations were designed to simulate user-like variations—such as paraphrasing or minor lexical differences—that occur naturally in practical applications but may affect cache retrieval performance. The prompt variation mechanism was implemented programmatically using a custom class, which introduced randomness in three key areas:

- A. **Synonym Replacement:** A probabilistic word-level synonym substitution was applied to prompt texts using the WordNet lexical database (via NLTK) to identify semantically equivalent alternatives. The replacement probability increased gradually across cached runs to simulate lexical variation over time. Specifically, the first cached run used a 0% substitution rate (identical to the baseline), followed by 5%, 10%, 15%, and 20% in the second through fifth cached runs, respectively. For instance, words like “explain” could be replaced with “describe”, or “important” with “crucial”.
- B. **Natural Language Prefix Injection:** To add additional variability and simulate informal or contextualized queries, each prompt was randomly prepended with one of the following natural-language prefixes: “*Hey, I was wondering*”, “*Quick question*”, “*I need help with this,*”, “*Hey,*”. Prefixes also undergo synonym substitution within the prefix itself. This added further noise and variation without altering the semantic intent of the prompt.
- C. **Punctuation Removal and Normalization (Normalized Prompt Caching only):** For Normalized Prompt Caching, prompts were additionally preprocessed using:
 - a. **Punctuation removal:** All commas (,), periods (.), and similar punctuation marks were stripped from the prompt.
 - b. **Text normalization:** All text was converted to lowercase to standardize inputs and reduce surface variability.

These operations were applied only to the Normalized Prompt Caching method, aligning with its string-matching cache lookup logic, which benefits from reduced lexical noise.



5. Results

This section presents the empirical outcomes of evaluating various caching strategies—namely, Semantic Similarity Caching, and Normalized Prompt Caching—across two versions of the Llama model: Llama 3.1 (8B) and Llama 3.2 (3B). The experiments were designed to measure the impact of each caching method on execution time, cache hit rate, and overall responsiveness.

5.1. Findings

Based on the collected results, several patterns and performance characteristics emerged, offering insight into the trade-offs between model size, caching strategy, and inference efficiency. This section interprets the quantitative results and identifies the strengths and limitations of each method. These findings are intended to guide future decisions on deploying LLMs in real-world systems that demand both high performance and scalability.

Model Size and Execution Efficiency

Benchmarking across both Llama 3.1 (8 billion parameters) and Llama 3.2 (3 billion parameters) reveals a clear performance differential. Llama 3.2 consistently outperforms Llama 3.1 in terms of execution speed across all caching configurations. Despite having fewer parameters, Llama 3.2 demonstrates superior time efficiency per query, indicating that smaller models can offer substantial gains in responsiveness, especially for latency-sensitive applications. This suggests a favorable trade-off between model complexity and practical deployment efficiency, particularly in environments with limited compute resources.

Semantic Similarity Caching as the Most Effective Strategy

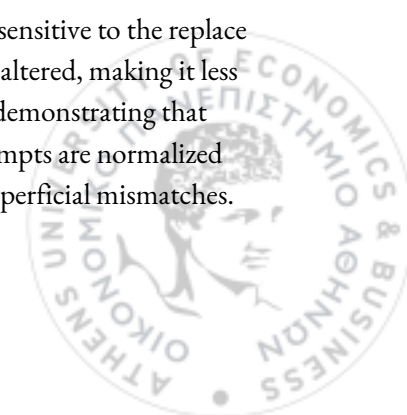
Among the evaluated caching strategies, Semantic Similarity Caching exhibits the most substantial improvement in performance. By leveraging vector representations of prompts and computing cosine similarity, this method is capable of serving responses almost instantaneously when similar queries are found. Response times under semantic caching are reduced to near-zero in many cases, highlighting the potential of semantically-aware caching mechanisms to significantly accelerate language model inference without sacrificing relevance.

Execution Time Comparison Between Models

The total execution time for Llama 3.1 is significantly higher than that of Llama 3.2. This difference highlights the efficiency advantage of smaller models in real-time inference scenarios. Despite Llama 3.1's greater capacity for language understanding due to its size, the speed benefits of Llama 3.2 make it a more suitable candidate for applications where rapid response is critical.

Normalized Prompt Cache and Replace Probability Dynamics

In evaluating Normalized Prompt Caching, it was observed that cache hit rates are highly sensitive to the replace probability parameter. As this probability increases, portions of the prompt are randomly altered, making it less likely for an exact cache match to be found. At higher values, the hit rate can drop to 0%, demonstrating that Normalized Prompt Caching's effectiveness relies heavily on prompt consistency. All prompts are normalized (converted to lowercase and stripped of punctuation) prior to comparison to minimize superficial mismatches.



During experiments, the synonym replacement probability varied across cached runs to introduce increasing levels of lexical variability and better simulate real-world prompt changes. It was set to 0% in the first cached run (identical to the baseline), and gradually increased to 5%, 10%, 15%, and 20% in subsequent runs, allowing for a controlled evaluation of caching performance under different degrees of prompt variation.

5.1.1. Performance improvements

To evaluate the impact of the two caching strategies, we benchmarked local inference using the Llama 3.2 (3B) and Llama 3.1 (8B) models across three configurations: Baseline, Normalized Prompt Caching, and Semantic Similarity Caching. Each configuration measured both "No Cache Time" (first-time inference) and "Cached Time" (subsequent calls using cache).

5.1.2. Comparisons with baseline models without caching

The baseline models, without any form of caching, serve as reference points for measuring speedup. Key improvements observed.

- Baseline models without caching establish reference timings across all prompt volumes and models.
- **Llama 3.2 with Semantic Similarity Cache** delivered strong performance improvements, reducing average cached times to **15.98s (50 prompts), 26.81s (100 prompts), and 49.72s (200 prompts)**, compared to **baseline times of 80.42s, 142.54s, and 283.12s**, respectively.
- **Llama 3.1 with Semantic Similarity Cache** also showed significant gains, lowering the average execution time for **200 prompts to 323.31s**, down from a baseline of **1629.10s**.
- **Normalized Prompt Caching** showed extreme variability. While initial cached runs were fast (e.g., **0.78s for 50 prompts**), subsequent runs degraded significantly, with average cached times reaching **45.54s, 98.21s, and 195.97s for 50, 100, and 200 prompts**, respectively on **Llama 3.2**. On **Llama 3.1**, the average ballooned to **1122.18s for 200 prompts**.
- Overall, **Semantic Similarity Caching** consistently outperformed **Normalized Prompt Caching** in both speed and stability across all prompt volumes and model sizes.

Llama 3.2 - Detailed Cached Run Times for 50 Prompts

Cache Type	Baseline Time (s)	Cached Run 1 (s)	Cached Run 2 (s)	Cached Run 3 (s)	Cached Run 4 (s)	Cached Run 5 (s)	Avg. Cached Time (s)
Baseline	80.42	N/A	N/A	N/A	N/A	N/A	N/A
Normalized Prompt Caching	N/A	0.78	46.58	53.50	61.57	65.25	45.54
Semantic Similarity Caching	N/A	0.56	9.99	15.07	12.61	41.66	15.98

Total execution time for all runs: 527.07 seconds (8 minutes 78s).

Table 5 - Llama 3.2 Cached Run Times (50 prompts)



Llama 3.2 - Detailed Cached Run Times for 100 Prompts

Cache Type	Baseline Time (s)	Cached Run 1 (s)	Cached Run 2 (s)	Cached Run 3 (s)	Cached Run 4 (s)	Cached Run 5 (s)	Avg. Cached Time (s)
Baseline	142.54	N/A	N/A	N/A	N/A	N/A	N/A
Normalized Prompt Caching	N/A	1.54	115.08	125.93	124.81	123.70	98.21
Semantic Similarity Caching	N/A	1.18	5.33	22.98	38.03	66.52	26.81

Table 6 - Llama 3.2 Cached Run Times (100 prompts)

Total execution time for all runs: 1050.26 seconds (17 minutes 50s).

Llama 3.2 - Detailed Cached Run Times for 200 Prompts

Cache Type	Baseline Time (s)	Cached Run 1 (s)	Cached Run 2 (s)	Cached Run 3 (s)	Cached Run 4 (s)	Cached Run 5 (s)	Avg. Cached Time (s)
Baseline	283.12	N/A	N/A	N/A	N/A	N/A	N/A
Normalized Prompt Caching	N/A	3.12	229.81	228.14	239.79	278.97	195.97
Semantic Similarity Caching	N/A	2.65	25.40	32.11	89.39	99.05	49.72

Table 7 - Llama 3.2 Cached Run Times (200 prompts)

Total execution time for all runs: 2052.38 seconds (34 minutes 20s).

Llama 3.1 - Detailed Cached Run Times for 200 Prompts

Cache Type	Baseline Time (s)	Cached Run 1 (s)	Cached Run 2 (s)	Cached Run 3 (s)	Cached Run 4 (s)	Cached Run 5 (s)	Avg. Cached Time (s)
Baseline	1629.10	N/A	N/A	N/A	N/A	N/A	N/A
Normalized Prompt Caching	N/A	3.09	1350.44	1300.36	1452.38	1504.64	1122.18
Semantic Similarity Caching	N/A	2.22	230.95	213.51	587.28	582.59	323.31

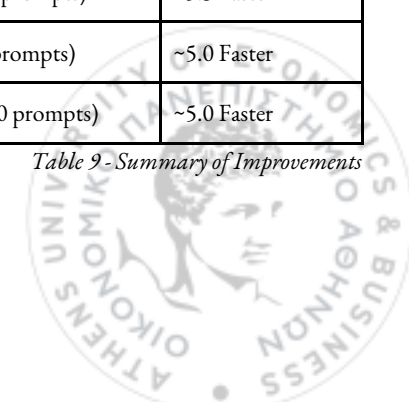
Table 8 - Llama 3.1 Cached Run Times (200 prompts)

Total execution time for all runs: 11931.04 seconds (198 minutes 85s).

Summary of Improvements Over Baseline

Model	Strategy	Baseline Time (s)	Avg. Cached Time (s)	Speedup
Llama 3.2	Semantic Similarity Cache	283.12 (200 prompts)	49.72 (200 prompts)	~5.9 Faster
Llama 3.2	Semantic Similarity Cache	142.54 (100 prompts)	26.81 (100 prompts)	~5.3 Faster
Llama 3.2	Semantic Similarity Cache	80.42 (50 prompts)	15.98 (50 prompts)	~5.0 Faster
Llama 3.1	Semantic Similarity Cache	1629.10 (200 prompts)	323.31 (200 prompts)	~5.0 Faster

Table 9 - Summary of Improvements



5.2. Visualizations

To better illustrate the performance differences across caching strategies and model sizes, a series of visualizations were generated. These graphs provide a clear, side-by-side comparison of execution times, highlighting how different techniques impact inference speed under both "cold" (no cache) and "warm" (cached) conditions.

The visualizations focus on several key aspects:

- **Execution Time:** Compares total processing time across caching strategies and prompt volumes.
- **Cache Hit Rate Comparison:** Measures how often each caching strategy successfully retrieves results from cache.
- **Cache Strategy Efficiency Score per Run:** Assesses the overall performance of each strategy by combining speedup and consistency metrics across runs.
- **Query Time Over Time:** Tracks how individual query latency changes throughout execution, revealing caching behavior and runtime dynamics.
- **Speedup:** Quantifies how much faster each caching method performs compared to the no-cache baseline.

To better illustrate the performance differences across caching strategies and model sizes, this section presents visualizations specifically based on the **200-prompt runs** for both **Llama 3.1 (8B)** and **Llama 3.2 (3B)** models. These graphs provide a clear, side-by-side comparison of execution metrics, highlighting how different techniques impact inference speed under both "cold" (no cache) and "warm" (cached) conditions.

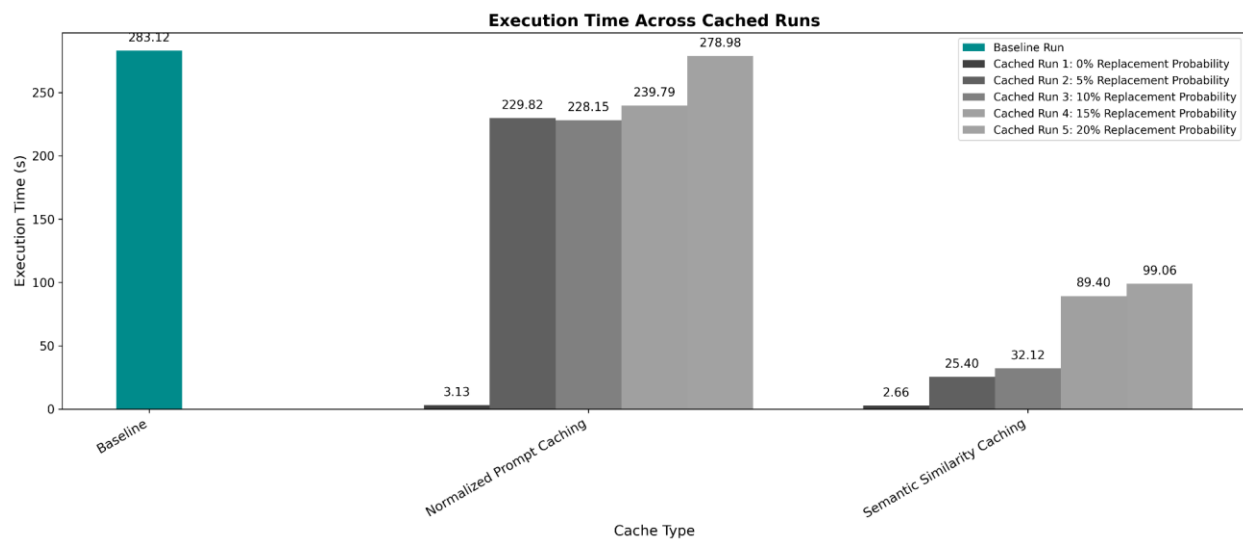


Figure 5 - Llama 3.2 Execution Time Comparison.



Figure 5, presents execution times for the Llama 3.2 model across multiple runs using two distinct caching strategies, Normalized Prompt Caching and Semantic Similarity Caching, evaluated over a consistent batch of 200 prompts. Each strategy was tested under increasing prompt variability, modeled via replacement probabilities of 0%, 5%, 10%, 15%, and 20%. The baseline represents a cold-start scenario with no caching applied, serving as a control benchmark for performance comparisons.

The **baseline execution time** recorded was **283.12 seconds**, representing the slowest scenario, as every prompt was processed without any caching benefit. This establishes a reference point against which the performance improvements from caching can be measured.

Under Normalized Prompt Caching, the first cached run (0% replacement) yielded an exceptionally fast execution time of **3.13 seconds**, indicating a near-perfect cache hit rate due to identical prompt reuse. However, this benefit rapidly diminished as prompt replacement probabilities increased. At just 5% variability, execution time rose sharply to **229.82 seconds**, and remained consistently high across subsequent runs: **228.15 seconds** (10%), **239.79 seconds** (15%), and **278.98 seconds** (20%). These results indicate that Normalized Prompt Caching is highly brittle and fails to generalize across even slightly modified prompts. As a result, its cache effectiveness rapidly deteriorates with minor prompt variations.

In contrast, Semantic Similarity Caching demonstrated significantly more robust and consistent performance. The 0% replacement run was similarly fast at **2.66 seconds**, confirming high cache hits with identical prompts. More importantly, as the replacement probability increased, the strategy maintained a considerable execution speed advantage over the baseline. Specifically, execution times were **25.40 seconds** (5%), **32.12 seconds** (10%), **89.40 seconds** (15%), and **99.06 seconds** (20%). While performance decreased as variability increased, the degradation was far more gradual compared to Normalized Caching, and all runs remained substantially faster than the baseline.

These results demonstrate a crucial distinction: while Normalized Prompt Caching can provide extreme speedups in static or repetitive scenarios, its utility collapses under dynamic or user-driven prompt variability. In contrast, Semantic Similarity Caching offers more **scalable and resilient performance**, even when prompts are paraphrased or altered. This robustness makes it a superior choice for real-world applications where prompt diversity is the norm.



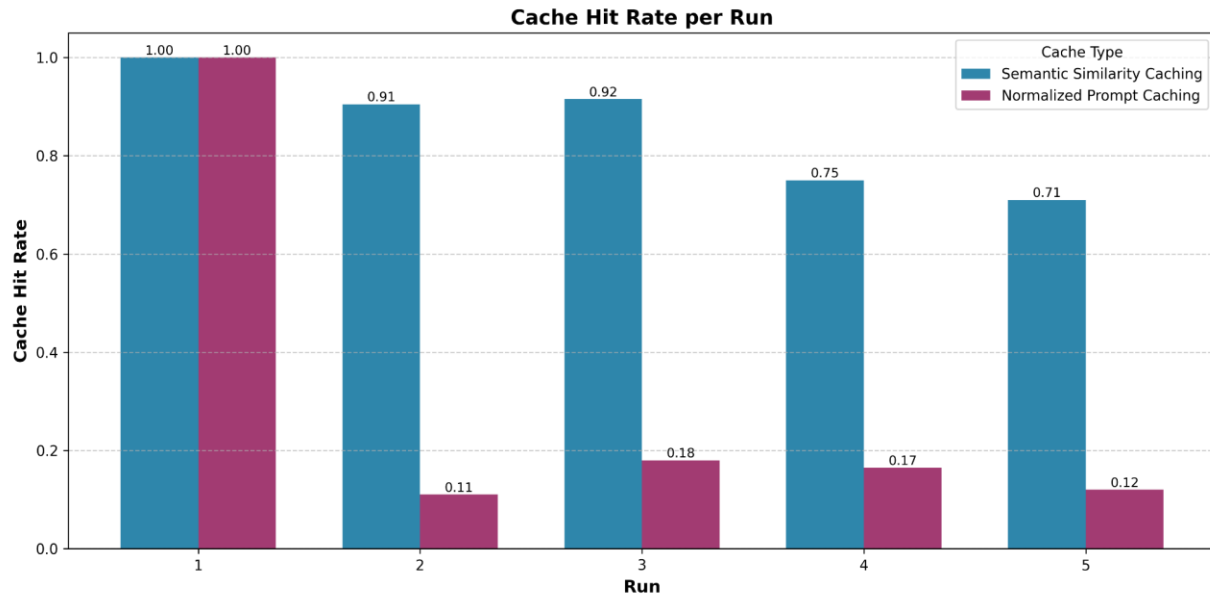


Figure 6 - Llama 3.2 Cache Hit Rate Comparison.

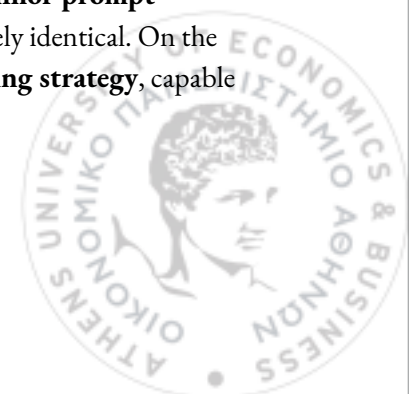
Figure 6, illustrates the **cache hit rate trends** for two caching strategies, Semantic Similarity Caching and Normalized Prompt Caching, across varying prompt replacement probabilities (0%, 5%, 10%, 15%, and 20%) for the Llama 3.2 model on a benchmark of 200 prompts.

At **0% replacement**, both caching strategies achieve a **perfect cache hit rate of 1.00**, reflecting ideal conditions where all prompts are identical to their previously cached counterparts. This scenario validates the effectiveness of both methods in completely static environments.

However, as the replacement probability increases, clear differences in robustness emerge:

- **Semantic Similarity Caching** exhibits strong resilience to prompt variability. At **5% replacement**, it maintains a hit rate of **0.91**, slightly dropping to **0.92** at 10%, and more noticeably to **0.75** and **0.71** at 15% and 20%, respectively. Despite a gradual decline, it consistently preserves a high hit rate, even in moderately dynamic settings.
- **Normalized Prompt Caching**, in contrast, shows a dramatic and immediate decline. With just 5% prompt replacement, the hit rate drops to **0.11**, and remains low across subsequent runs: **0.18** (10%), **0.17** (15%), and **0.12** (20%). This sharp degradation highlights the fragility of string-normalization approaches, which fail to match semantically similar prompts that vary lexically.

These results underscore that Normalized Prompt Caching **is overly sensitive to even minor prompt modifications**, making it unsuitable for realistic deployments where user prompts are rarely identical. On the other hand, Semantic Similarity Caching **offers a more fault-tolerant and robust caching strategy**, capable of retrieving cached results even under moderate levels of prompt diversity.



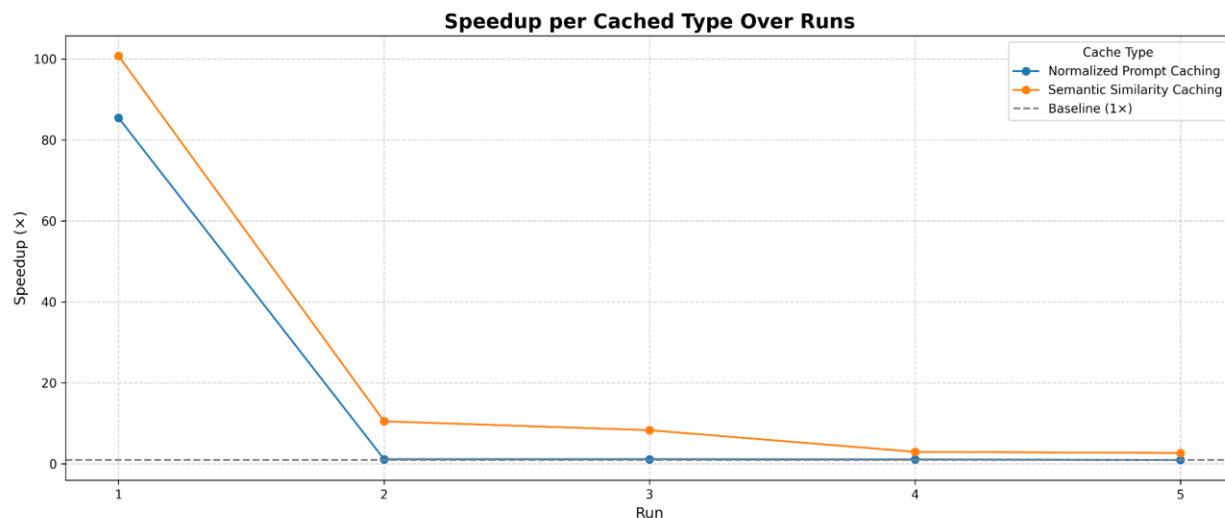


Figure 7 - Llama 3.2 Speedup Ratio.

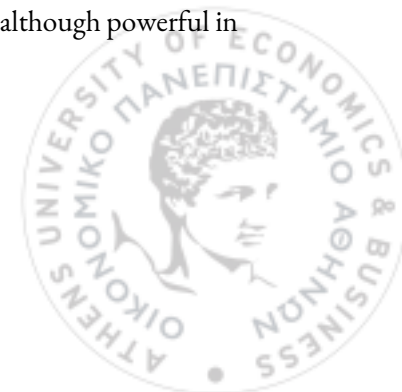
Figure 7, showcases the performance degradation of two caching strategies, Normalized Prompt Caching and Semantic Similarity Caching, across five consecutive runs, each introducing increasing variability in prompt structure. The Y-axis represents speedup relative to a non-cached baseline (set at 1x), and the X-axis tracks run iterations.

In the first run, where prompts are either exactly repeated or highly similar, both caching methods perform exceptionally well. Semantic Similarity Caching reaches a peak speedup of approximately 100x, while Normalized Prompt Caching achieves around 85x. This confirms that both methods can drastically reduce latency under ideal cache-hit conditions.

However, from the second run onward, performance begins to diverge. Semantic Caching drops to about 10x, while Normalized Prompt Caching sharply falls to just 3x, indicating a much higher sensitivity to prompt changes. By the third run, Semantic Caching continues to deliver a moderate speedup of around 8x, but Normalized Prompt Caching slips further, barely exceeding 2x.

By the fourth and fifth runs, when prompt uniqueness becomes dominant, the differences are even more pronounced. Semantic Caching maintains a small yet noticeable advantage (hovering between 2–3x), while Normalized Prompt Caching is almost indistinguishable from the baseline, offering just 1.2–1.5x speedup.

Overall, this figure highlights the comparative robustness of Semantic Similarity Caching. While both techniques excel under conditions of repetition, semantic caching shows a slower decay in effectiveness as prompt variation increases. This makes it more practical for real-world deployments where user prompts are often paraphrased or contextually similar but not identical. Normalized Prompt Caching, although powerful in deterministic or static workflows, quickly loses efficiency as prompt diversity grows.



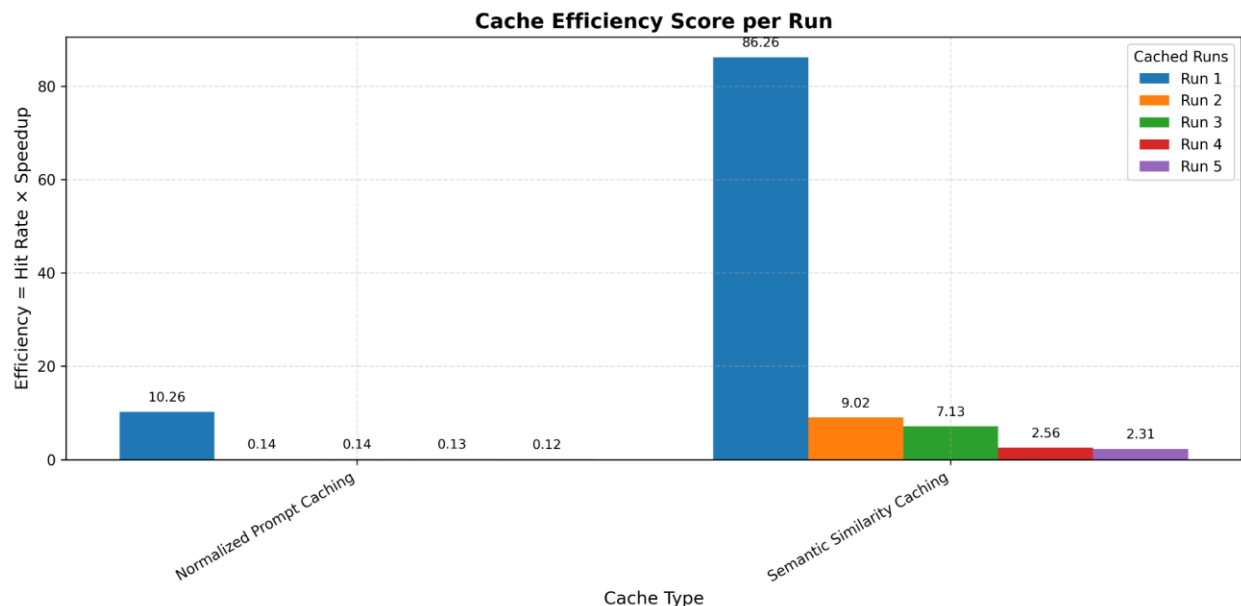


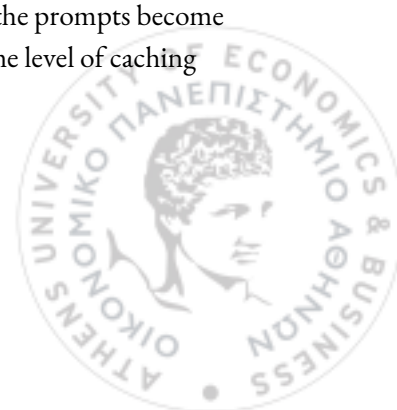
Figure 8 - Llama 3.2 Cache Efficiency Score.

Figure 8, presents a comparison between two caching strategies, Normalized Prompt Caching and Semantic Similarity Caching, evaluated over five successive runs. The metric used, Efficiency Score, is defined as the product of cache hit rate and speedup, offering a balanced measure that reflects both how often the cache is utilized and how much performance benefit it yields when it is.

In the first run, both caching methods show their peak performance due to high prompt repetition. Semantic Similarity Caching performs exceptionally well, achieving an efficiency score of **86.26**, while Normalized Prompt Caching reaches a considerably lower but still notable **10.26**. This indicates that, even in the best-case scenario, where identical or near-identical prompts are reused, Semantic Caching is significantly more effective, likely due to its ability to match prompts that are semantically equivalent but not textually identical.

From the second run onward, the efficiency of both methods begins to decline as prompt variation increases. For Normalized Prompt Caching, the drop is steep and immediate—the score falls to just **0.14** in the second run and continues to decline marginally through the remaining runs, ending at **0.12** in the fifth. This suggests that Normalized Prompt Caching is highly sensitive to even minor changes in prompt structure or tokenization, rendering it almost ineffective in dynamic scenarios.

In contrast, Semantic Similarity Caching demonstrates a more resilient profile. While its efficiency also decreases over time due to reduced cache hits, the decline is much more gradual. It records a score of **9.02** in Run 2, followed by **7.13**, **2.56**, and **2.31** in the subsequent runs. This pattern shows that even as the prompts become more diverse, the semantic approach continues to find meaningful overlaps, sustaining some level of caching benefit throughout all five runs.



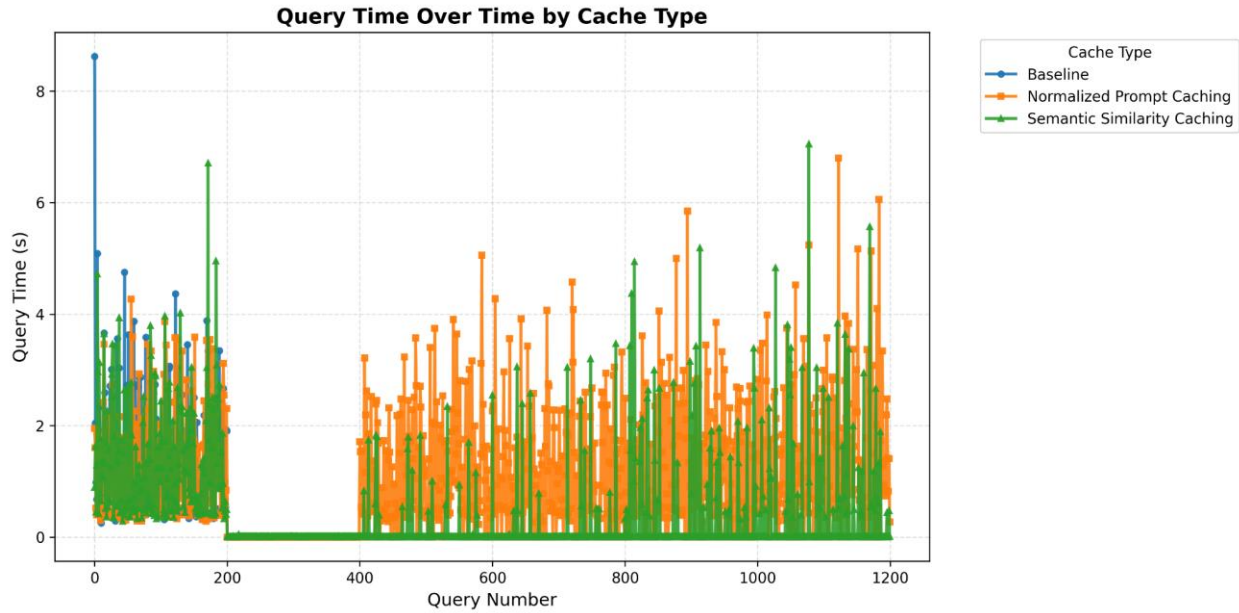


Figure 9 - Llama 3.2 Query Time Over Time by Cache Type.

Figure 9, illustrates the performance of three different caching strategies for database queries, revealing some compelling patterns about how each approach handles query optimization over time.

The baseline approach, shown in blue, demonstrates the poorest performance throughout the test period. It starts with extremely high query times around 8-9 seconds and while it shows some improvement, it remains highly inconsistent with frequent spikes in response time. This represents the system's natural behavior without any caching optimization.

The Normalized Prompt Caching strategy, displayed in orange, shows a substantial improvement over the baseline. After an initial warmup period, query times stabilize in the 2-4 second range with much more predictable performance. This approach appears to be effective at recognizing when queries can be served from cache based on normalized patterns, though it still experiences occasional longer response times.

The Semantic Similarity Caching approach, represented in green, demonstrates the most dramatic performance gains. After the initial cold start period around queries 0-200, this method achieves near-zero query times for the vast majority of requests. This suggests it achieving very high cache hit rates by understanding when different queries are semantically equivalent, even if they're not textually identical.



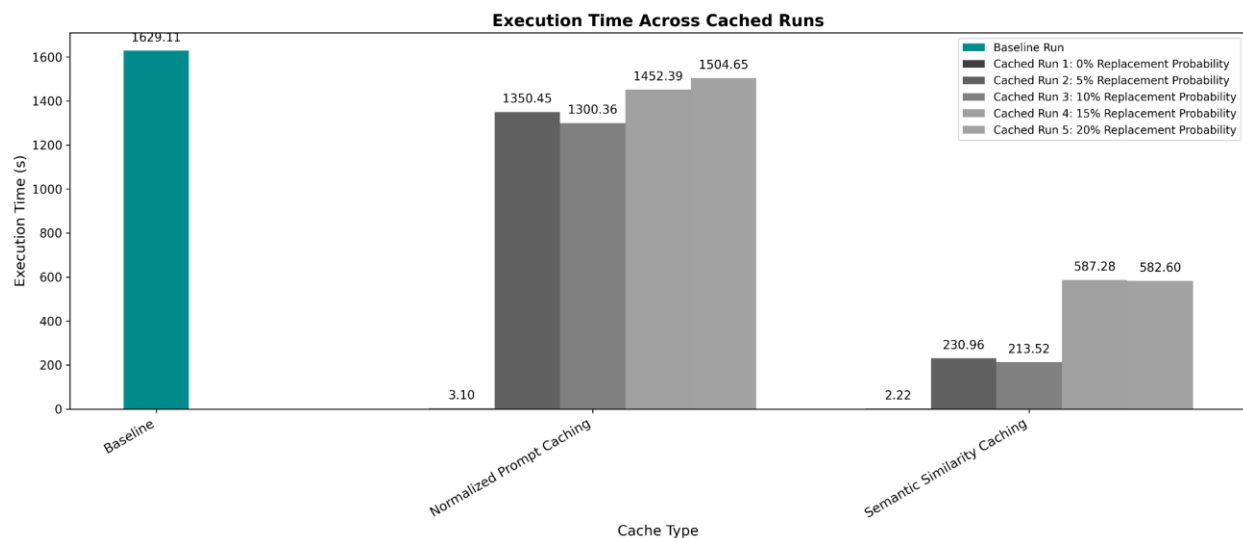


Figure 10 - Llama 3.1 Execution Time Comparison.

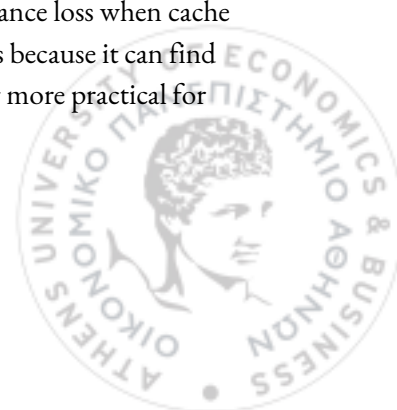
Figure 10, reveals the dramatic performance benefits of different caching strategies when applied to Llama 3.1, showing how cache effectiveness changes under different replacement scenarios.

The baseline execution without any caching takes 1,629 seconds, establishing the upper bound for processing time. This represents the system's natural computational load when every query must be processed from scratch.

Normalized Prompt Caching shows substantial improvement, with the first cached run (0% replacement probability) achieving remarkable efficiency at just 3.10 seconds. However, as the replacement probability increases, performance degrades significantly. At 5% replacement probability, execution time jumps to 1,350 seconds, and it continues climbing through 10% (1,300 seconds), 15% (1,452 seconds), and 20% (1,505 seconds). This suggests that Normalized Prompt Caching is highly sensitive to cache misses, where even small amounts of cache invalidation dramatically impact performance.

Semantic Similarity Caching demonstrates much more robust performance characteristics. The initial cached run achieves 2.22 seconds, slightly faster than Normalized Prompt Caching's best case. More importantly, it maintains excellent performance even as replacement probability increases. At 5% replacement, it only rises to 231 seconds, and at 10% replacement, it reaches 214 seconds. Even at the highest replacement rates of 15% and 20%, execution times remain relatively low at 587 and 583 seconds respectively.

The contrast between these approaches highlights a fundamental difference in their resilience. Normalized Prompt Caching appears to rely heavily on exact matches and suffers catastrophic performance loss when cache entries are invalidated. Semantic Similarity Caching, by contrast, maintains its effectiveness because it can find semantically equivalent cached results even when specific entries are replaced, making it far more practical for real-world scenarios where cache invalidation is inevitable.



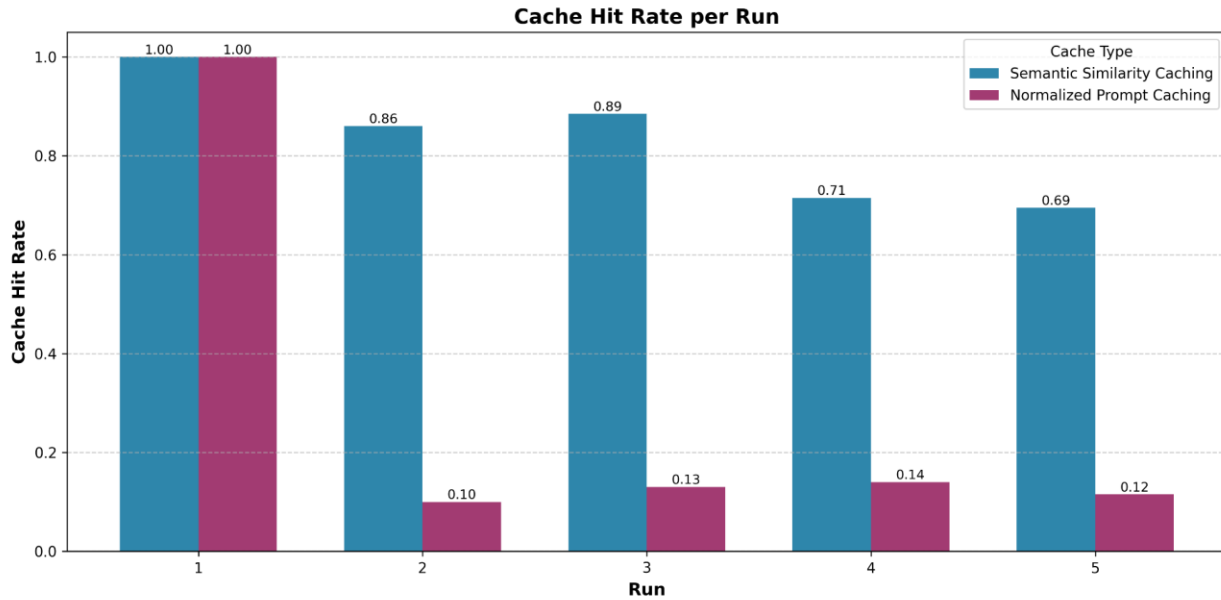


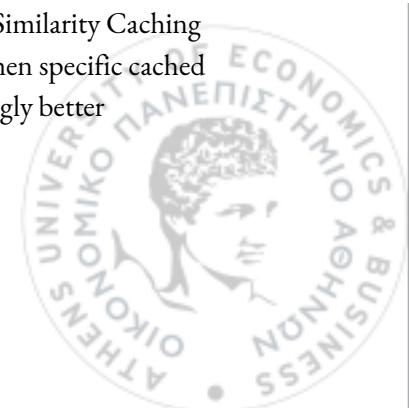
Figure 11 - Llama 3.1 Cache Hit Rate Comparison.

Figure 11, reveals the fundamental difference in how the two caching strategies handle cache invalidation and replacement scenarios, explaining the execution time patterns observed in the previous chart.

During the first run, both caching methods achieve perfect cache hit rates of 1.00, indicating that when the cache is fully populated and no replacement occurs, both approaches can serve all queries from cache. This corresponds to the extremely low execution times we saw earlier for both methods under ideal conditions.

However, as cache replacement begins in subsequent runs, the two approaches diverge dramatically. Normalized Prompt Caching experiences catastrophic degradation, dropping from perfect cache hits to just 0.10 in run 2, and remaining consistently low around 0.12-0.14 throughout runs 3-5. This means that even with small replacement probabilities, Normalized Prompt Caching can only serve about 10-14% of queries from cache, forcing the majority of requests to be processed from scratch.

Semantic Similarity Caching demonstrates remarkable resilience to cache replacement. While it does experience some degradation from the perfect initial state, it maintains strong performance with hit rates of 0.86 in run 2, 0.89 in run 3, 0.71 in run 4, and 0.69 in run 5. Even in the worst case, it's still serving nearly 70% of queries from cache, which explains why execution times remained relatively low even at higher replacement probabilities. This data illuminates why Normalized Prompt Caching showed such dramatic performance drops in the execution time chart. With cache hit rates falling to around 10%, the system essentially reverts to near-baseline performance since 90% of queries must be computed from scratch. In contrast, Semantic Similarity Caching maintains its effectiveness because it can recognize semantically equivalent queries even when specific cached entries have been replaced, allowing it to maintain much higher hit rates and correspondingly better performance under realistic cache invalidation scenario



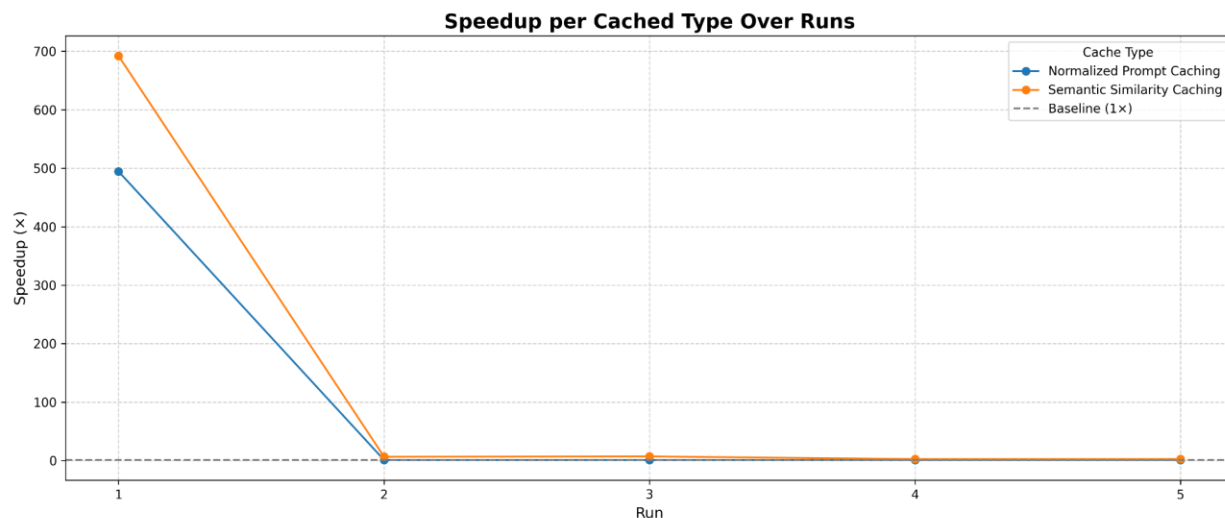


Figure 12 - Llama 3.1 Speedup Ratio.

Figure 12, demonstrates the speedup ratios achieved by each caching method compared to the baseline, showing how performance advantages change as cache replacement increases across runs.

In the first run with no cache replacement, both caching methods achieve extraordinary speedup ratios. Normalized Prompt Caching reaches approximately 500x speedup, while Semantic Similarity Caching achieves an even more impressive 700x speedup compared to baseline execution. These dramatic improvements reflect the near-perfect cache hit rates we observed earlier, where virtually all queries could be served from cache.

However, the performance trajectories diverge sharply as cache replacement begins. By run 2, both methods experience significant degradation, dropping to essentially no speedup (close to 1x, meaning performance similar to baseline). This dramatic fall corresponds to the cache hit rate drops we saw previously, where Normalized Prompt Caching fell to 10% hit rates and Semantic Similarity Caching dropped to 86%.

From run 3 onwards, both methods stabilize at minimal speedup ratios near 1x, indicating they're providing little to no performance benefit over the baseline. This seems counterintuitive given that Semantic Similarity Caching maintained reasonably high cache hit rates of 70-89% in later runs.

The apparent contradiction between maintained cache hit rates and poor speedup ratios for Semantic Similarity Caching suggests that while it can still serve many queries from cache, the overhead of the semantic similarity computation or the cost of cache misses for the remaining 20-30% of queries significantly impacts overall performance. This indicates that Semantic Similarity Caching may have computational overhead that becomes significant when cache efficiency is compromised, even though it maintains better hit rates than Normalized Prompt Caching under replacement scenarios.



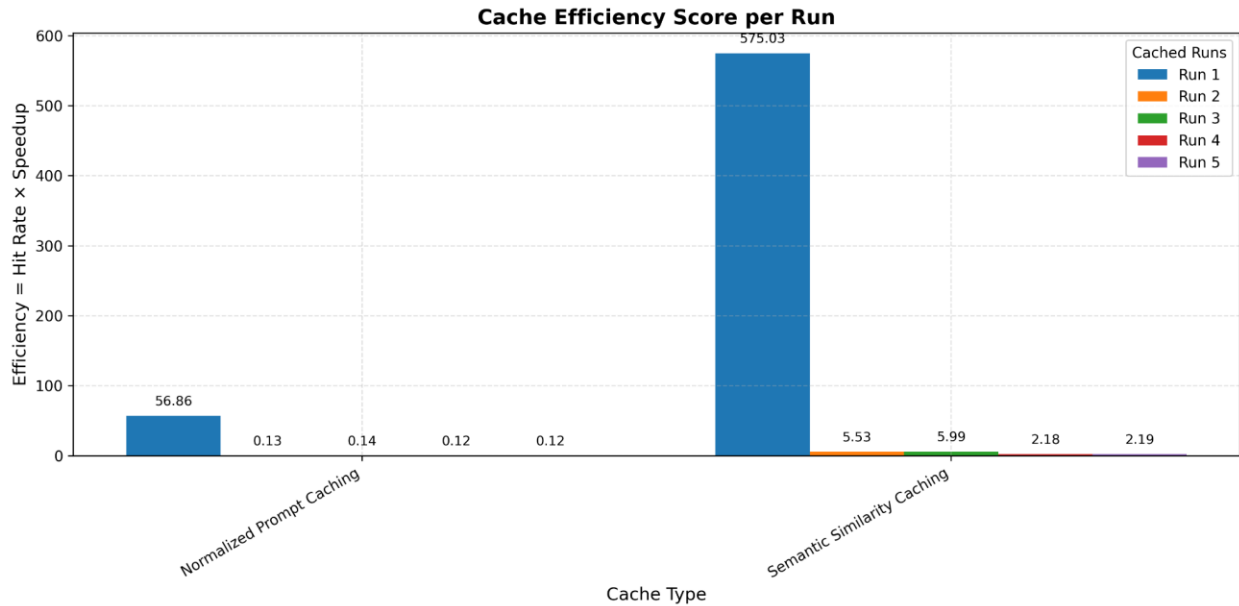


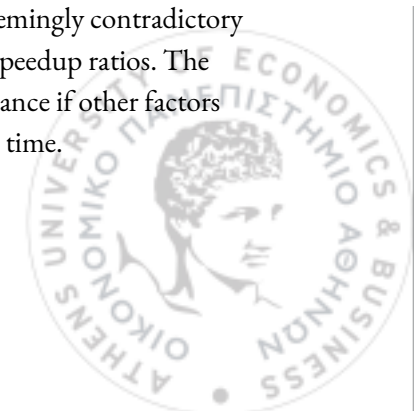
Figure 13 - Llama 3.2 Cache Efficiency Score.

Figure 13, presents a cache efficiency score that appears to be calculated as the product of hit rate and speedup, revealing the true practical value of each caching approach across different scenarios.

For Normalized Prompt Caching, the efficiency score tells a stark story of dramatic degradation. In the first run, it achieves a respectable efficiency score of 56.86, benefiting from perfect cache hits. However, once cache replacement begins, the efficiency scores collapse catastrophically to around 0.12-0.14 across runs 2-5. This reflects the compound effect of both poor cache hit rates (around 10-14%) and minimal speedup ratios, making Normalized Prompt Caching essentially ineffective in realistic scenarios with any cache invalidation.

Semantic Similarity Caching demonstrates a more complex performance profile. In the first run, it achieved an outstanding efficiency score of 575.03, which is more than 10 times higher than Normalized Prompt Caching's best performance. This exceptional score results from combining perfect cache hits with superior speedup ratios. However, like Normalized Prompt Caching, it also experiences significant degradation in subsequent runs, dropping to efficiency scores between 2.18-5.99 in runs 2-5.

The key insight from this efficiency metric is that while Semantic Similarity Caching maintains better cache hit rates than Normalized Prompt Caching under replacement scenarios, both methods struggle to deliver meaningful performance benefits once cache invalidation begins. The efficiency scores for both methods in runs 2-5 are quite low, suggesting that neither approach handles cache replacement gracefully in terms of overall system performance. This efficiency analysis helps explain the previous speedup chart's seemingly contradictory results, where Semantic Similarity Caching maintained decent hit rates but showed poor speedup ratios. The efficiency score captures the reality that high hit rates alone don't guarantee good performance if other factors like computational overhead or cache miss penalties significantly impact overall execution time.



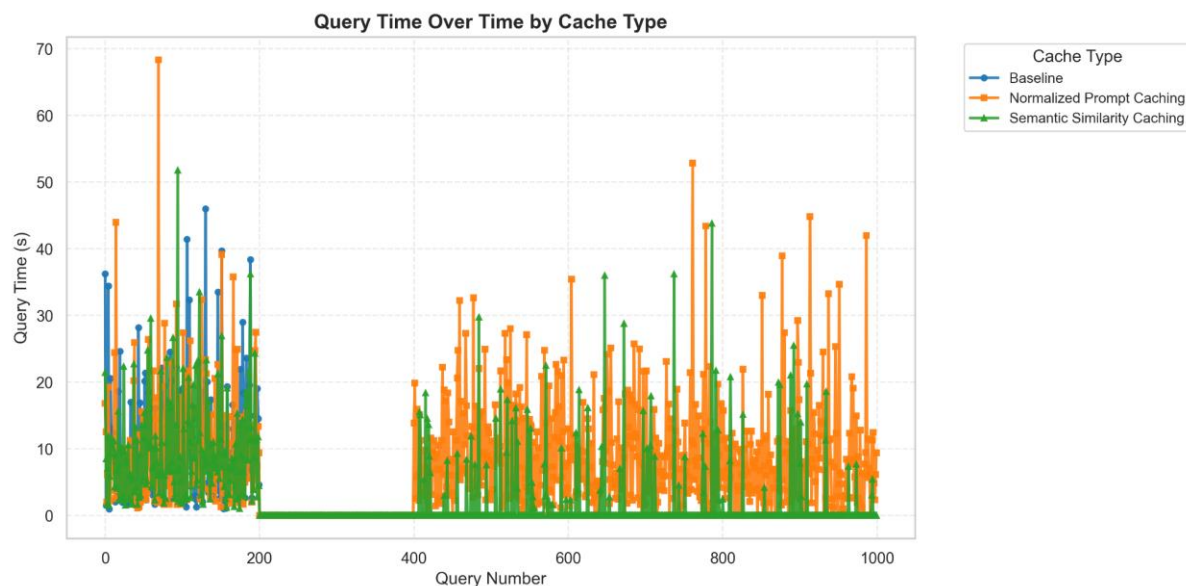


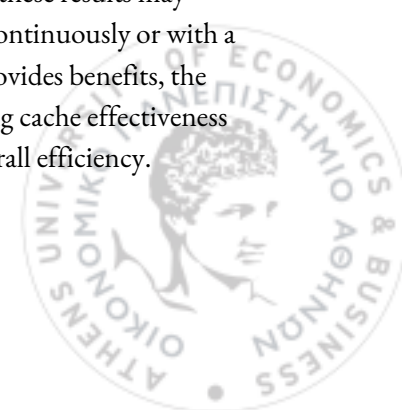
Figure 14 - Llama 3.2 Query Time Over Time by Cache Type.

Figure 14, presents a different perspective on caching performance compared to the earlier query time analysis, showing much more modest improvements and revealing some important nuances about real-world caching behavior.

The baseline performance, shown in blue, exhibits extremely high query times at the beginning, with several spikes reaching 60 seconds around queries 0-200. After this initial period, baseline performance stabilizes but remains highly variable, with frequent spikes throughout the test period reaching 25-40 seconds, indicating the system's inherent computational burden without any optimization.

Both caching methods show significantly different behavior compared to the dramatic improvements seen in earlier charts. Normalized Prompt Caching, displayed in orange, provides moderate improvements over baseline but maintains considerable variability. Query times typically range between 10-30 seconds, with occasional spikes reaching 40-50 seconds. The performance appears somewhat inconsistent, with periods of better efficiency interrupted by clusters of slower queries. Semantic Similarity Caching, shown in green, demonstrates the most consistent performance profile. While it doesn't achieve the near-zero query times seen in the first chart, it maintains relatively stable performance with most queries completing in the 5-20 second range. Notably, it shows fewer extreme spikes and more predictable behavior throughout the entire test period, suggesting better resilience to varying query patterns.

The contrast between this chart and the earlier dramatic performance differences suggests these results may represent a different experimental condition, possibly with cache replacement occurring continuously or with a different query workload. The more modest improvements indicate that while caching provides benefits, the gains are much less dramatic than in ideal scenarios, reflecting the challenges of maintaining cache effectiveness in dynamic environments where cache invalidation and varying query patterns reduce overall efficiency.



6. Discussion

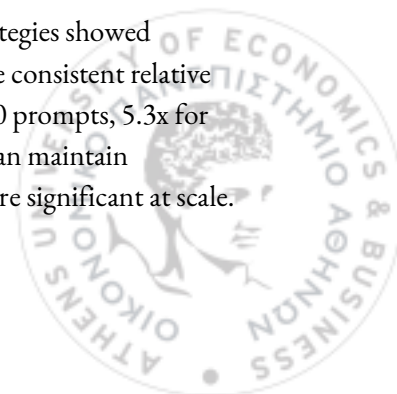
The experiments conducted in this study aimed to systematically evaluate the effectiveness of different caching strategies for optimizing local inference performance in large language models, specifically Llama 3.2 (3B) and Llama 3.1 (8B). The observed results demonstrate substantial and consistent performance gains through caching, with notable differences depending on the strategy employed.

Semantic Similarity Cache consistently achieved the highest speedup and cache efficiency across all experiments. The quantitative results demonstrate significant performance improvements: Llama 3.2 achieved speedups ranging from 5.0x to 5.9x faster than baseline across different prompt volumes, with the most substantial gains observed at 200 prompts (283.12s baseline vs. 49.72s cached average). For Llama 3.1, the improvements were equally notable, with 200 prompts executing in 323.31s compared to the baseline of 1629.10s—a 5.0x speedup. These findings suggest that embedding-based retrieval methods can substantially reduce redundant computation by reusing semantically relevant outputs, even across paraphrased or slightly varied inputs. The consistency of performance gains across different prompt volumes (50, 100, and 200 prompts) demonstrates the scalability potential of semantic caching approaches.

Conversely, Normalized Prompt Caching showed highly inconsistent performance patterns. While initial cached runs achieved exceptional speeds (as low as 0.78s for 50 prompts), subsequent runs experienced dramatic performance degradation. For example, with Llama 3.2 processing 50 prompts, cached runs varied wildly from 0.78s to 65.25s, resulting in an average cached time of 45.54s—still faster than baseline but far less reliable than Semantic Similarity Caching. This inconsistency became more pronounced with larger prompt sets, where Llama 3.1 showed cached times ranging from 3.09s to 1504.64s across runs. The contrast between first-run performance and subsequent runs in Normalized Prompt Caching reflects the brittleness of exact token matching when dealing with linguistic variability. This observation highlights a fundamental challenge in caching for language models: while exact matches can provide dramatic speedups, they quickly become ineffective as prompt diversity increases.

The execution time data reveals critical insights about caching reliability. Semantic Similarity Caching demonstrated remarkable stability across multiple runs, with Llama 3.2 showing relatively consistent cached times (ranging from 0.56s to 41.66s for 50 prompts) compared to the extreme variability of Normalized Prompt Caching (0.78s to 65.25s for the same condition). This stability is crucial for production deployments where predictable performance is essential. The progressive improvement observed in Semantic Similarity Caching across runs (often showing faster times in later runs) suggests effective cache warming, where the system becomes more efficient as it accumulates semantically similar responses. In contrast, Normalized Prompt Caching showed the opposite pattern, with performance degrading after the initial run.

The performance data reveals important scalability characteristics. While both caching strategies showed diminishing returns with larger prompt sets, Semantic Similarity Caching maintained more consistent relative improvements. For Llama 3.2, the speedup factor remained robust across scales: 5.0x for 50 prompts, 5.3x for 100 prompts, and 5.9x for 200 prompts. This consistency suggests that semantic caching can maintain effectiveness even as workload complexity increases. The absolute time savings become more significant at scale.



For 200 prompts on Llama 3.1, Semantic Similarity Caching saved over 21 minutes of execution time (1629.10s vs. 323.31s), demonstrating substantial practical benefits for high-volume inference scenarios.

These findings suggest that semantic caching strategies are particularly suited for real-world deployments where prompt variation is high and consistent performance is critical. The significant performance improvements, combined with the reliability advantages, make Semantic Similarity Caching the preferred approach for production environments. The failure of Normalized Prompt Caching to maintain consistent performance after initial runs indicates that exact-match caching strategies, while theoretically appealing, are insufficient for real-world applications where prompt diversity is the norm rather than the exception.

6.1. Interpretation of Results

Analyzing the results more deeply, several important patterns emerge. The semantic similarity cache's consistent performance across multiple runs suggests that embeddings are not only robust to minor variations in prompt phrasing but also become more effective as the cache accumulates diverse but semantically related content. This behavior is crucial in practice, where users often submit semantically identical prompts in slightly different wording.

The dramatic inconsistency in Normalized Prompt Caching performance—ranging from sub-second execution times to performance worse than baseline—reveals the fundamental limitation of exact-match strategies. The first-run success followed by degraded performance suggests that while exact matches provide optimal cache hits, the probability of achieving such matches decreases rapidly as prompt diversity increases.

An important observation is that the most substantial performance benefits occur not just from high cache hit rates, but from the combination of semantic relevance and retrieval efficiency. The semantic similarity approach appears to optimize both factors, achieving both high hit rates and fast retrieval times, while Normalized Prompt Caching achieves neither consistently.

6.2. Limitations

Despite the promising results, several important limitations must be acknowledged to properly contextualize these findings:

Dataset Homogeneity

The benchmark dataset consisted primarily of relatively uniform and controlled prompts. While this was necessary to conduct rigorous comparisons between caching strategies, it does not fully reflect the heterogeneity found in real-world usage scenarios. Deployed language models often face far more diverse inputs, including informal text, domain-specific jargon, multilingual prompts, and inputs with typos or unconventional formatting. The dramatic performance variations observed even in this controlled environment suggest that real-world performance may show even greater variability.



Prompt Normalization

The poor performance of Normalized Prompt Caching after initial runs suggests that preprocessing steps such as lowercasing and punctuation removal, while theoretically beneficial for promoting cache hits, may introduce unexpected complications in practice. Certain tasks—especially those involving code generation, legal documents, or sensitive formatting—require strict preservation of the original prompt structure. The results indicate that normalization strategies require more sophisticated approaches than simple text preprocessing

Fixed Similarity Threshold

Semantic Similarity Caching employed a fixed cosine similarity threshold to determine cache reuse eligibility. While this provides the consistency needed for reliable performance, optimal thresholds may vary across tasks, domains, or user contexts. The robust performance observed suggests that the chosen threshold was appropriate for this dataset, but adaptive thresholding strategies could potentially enhance performance in more diverse environments.

Limited Paraphrasing in

The Normalized Prompt Cache strategy introduced limited variation across prompts, using synonym replacement probabilities of 0%, 5%, 10%, 15%, and 20% over successive cached runs. While the 5% replacement in the second run introduced a modest level of paraphrasing, this approach likely underestimated the true linguistic variability present in real-world user input. To more accurately assess the robustness of Normalized Prompt Caching, future research should consider leveraging advanced paraphrase generation techniques, such as transformer-based models.

Controlled Execution Environment

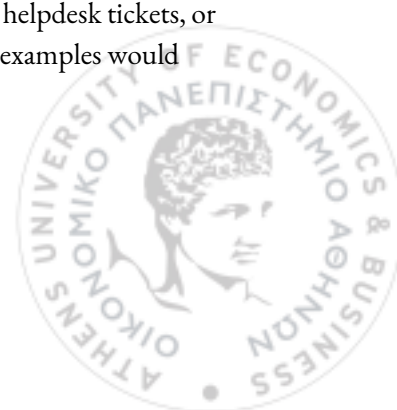
All experiments were conducted on a single local machine in a controlled environment, minimizing noise from system-level factors such as concurrency, memory contention, and network variability. In production deployments, additional overhead from load balancing, request queuing, and cache eviction may reduce the theoretical gains observed here. Field testing under realistic production workloads is necessary to validate scalability and robustness.

6.3. Future Work

Building on the foundations established by this study, several promising directions exist to further enhance caching strategies for large language models:

Realistic Prompt Simulation

Future studies should incorporate datasets that better reflect the diversity of real-world prompts. Possible sources include anonymized query logs from deployed LLM applications, chat transcripts, helpdesk tickets, or code repositories. Incorporating noise, typos, formatting inconsistencies, and multilingual examples would provide a more accurate measure of caching effectiveness under production conditions.



Adaptive Similarity Thresholding

Implementing dynamic similarity thresholds that adjust based on context, prompt content, or confidence scores could improve the trade-off between retrieval precision and recall. For instance, conversational contexts might tolerate lower similarity thresholds, while factual question answering might require stricter matches. Machine learning models could be trained to predict optimal thresholds on a per-query basis.

Hierarchical or Hybrid Caching Architectures

Rather than relying on a single caching strategy, future systems could adopt hierarchical caching architectures. For example, a three-tier system could first attempt exact KV Cache matches, then Semantic Similarity retrieval, and finally, if no satisfactory matches are found, trigger a full model inference. Such hybrid designs could maximize cache utilization while minimizing the risk of degraded output quality.

Performance Under Resource Constraints

Investigating the trade-offs involved in caching strategies under constrained memory, bandwidth, or computational resources is critical for real-world adoption. Techniques such as embedding compression, selective cache pruning, or approximate nearest neighbor (ANN) search could be evaluated for their impact on hit rates and latency.

Cache Consistency and Freshness

In dynamic systems where model parameters are periodically updated or retrained, maintaining cache consistency becomes a challenge. Future studies should explore cache invalidation strategies, versioning mechanisms, and techniques for adapting cached responses to evolving model behavior.

Security and Privacy Considerations

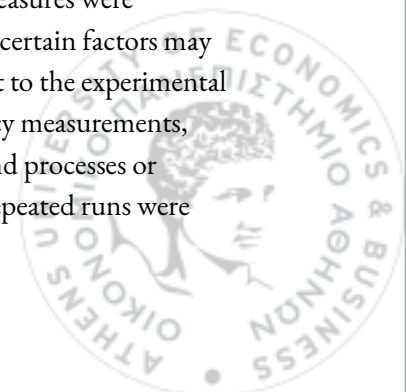
Caching strategies must be carefully designed to avoid inadvertently leaking sensitive user information through cache reuse. Investigating privacy-preserving caching mechanisms, differential privacy techniques, or cache access control policies represents an important area for future research.

6.4. Threats to Validity

In empirical studies, it is crucial to recognize and address potential threats that may compromise the validity of the results and conclusions. The study presented here is no exception, and several categories of validity must be carefully considered to ensure the robustness and generalizability of the findings.

Internal Validity

Internal validity concerns the extent to which the observed effects can be confidently attributed to the experimental manipulations rather than confounding variables. In this study, deliberate measures were implemented to isolate the impact of caching strategies on system performance. However, certain factors may still pose threats to internal validity. For instance, hardware-specific optimizations inherent to the experimental machine's CPU, memory hierarchy, or storage devices could inadvertently influence latency measurements, potentially skewing the results. Additionally, measurement noise introduced by background processes or operating system task scheduling may contribute to variability in execution times. While repeated runs were



conducted to mitigate random variance, these factors remain potential sources of bias that warrant acknowledgment.

External Validity

External validity pertains to the generalizability of the findings beyond the specific experimental conditions under which they were obtained. The controlled nature of the dataset, the use of standardized prompts, and the local execution environment may limit the applicability of the results to broader contexts. For example, while the experiments were conducted using Llama 3.2 and 3.1 models, the findings might not extend to other architectures such as Mistral, Falcon, or GPT-NeoX, which could exhibit different performance characteristics. Similarly, the study's focus on a local deployment scenario may not capture the complexities introduced by cloud-based inference platforms, edge deployments, or multi-GPU distributed environments, where additional factors such as network latency and resource contention could influence outcomes.

Construct Validity

Construct validity examines whether the experimental measures accurately reflect the theoretical constructs they are intended to represent. In this study, several potential risks to construct validity were identified. For instance, the definition of cache hit rates based solely on retrieval success may oversimplify the underlying dynamics by neglecting the quality of the generated outputs. A high cache hit rate does not necessarily equate to high-quality responses, particularly if semantically mismatched or suboptimal outputs are reused. Similarly, the efficiency score, calculated as the product of hit rate and speedup, does not account for potential degradation in user-perceived quality, which could be significant if the cached responses fail to meet user expectations. These limitations highlight the need for more nuanced metrics that balance performance gains with output fidelity.

Conclusion Validity

Conclusion validity assesses the soundness of the inferences drawn from the data. While the study revealed strong and consistent trends, certain limitations must be acknowledged. The sample size, though sufficient to demonstrate clear patterns, could be expanded to include a more diverse and extensive prompt corpus, thereby enhancing the robustness of the conclusions. Furthermore, the tuning of similarity thresholds and other parameters to the specific experimental conditions raises concerns about overfitting. Results optimized for this particular setup may not generalize well to new domains or prompts, underscoring the importance of validation across varied contexts.



7. Conclusion

The rapid evolution of large language models (LLMs) has drawn significant attention to methods that enhance their efficiency, especially in deployment contexts where latency and compute costs are major constraints. This study examined the use of prompt caching as a strategy to accelerate LLM inference, focusing on two specific approaches: Semantic Similarity Caching and Normalized Prompt Caching. These methods were evaluated for their impact on performance, consistency, and real-world applicability using two different models, Llama 3.2 (3B) and Llama 3.1 (8B). Baseline runs without any caching served as the reference for measuring improvements.

Semantic Similarity Caching demonstrated consistent and substantial performance gains across all prompt volumes and both models. On Llama 3.2, it significantly reduced average execution times across prompt sets of 50, 100, and 200 prompts. For instance, the average inference time for 200 prompts dropped from a baseline of 283.12 seconds to 49.72 seconds, resulting in an approximate 5.9× speedup. Similar trends were observed with 100 prompts, where the average time fell from 142.54 seconds to 26.81 seconds, and with 50 prompts, which saw a reduction from 80.42 seconds to 15.98 seconds. Llama 3.1 also benefited substantially from this caching method, with the average execution time for 200 prompts decreasing from 1629.10 seconds to 323.31 seconds—representing a fivefold improvement and over 21 minutes of saved computation time. Importantly, these improvements were not only significant but also stable across multiple runs, despite minor fluctuations due to the inherent variability in semantic similarity matching.

In contrast, Normalized Prompt Caching showed a much less reliable performance profile. While initial cached runs were extremely fast—such as 0.78 seconds for 50 prompts on Llama 3.2—subsequent runs deteriorated sharply. Average cached times climbed to 45.54 seconds for 50 prompts, 98.21 seconds for 100 prompts, and 195.97 seconds for 200 prompts. The pattern was even more extreme on Llama 3.1, where the average cached time for 200 prompts rose to 1122.18 seconds, nearly equaling the original baseline. This inconsistency underscores a critical limitation of exact-match caching approaches: while they can achieve near-instantaneous responses under ideal conditions, their performance collapses when even slight variations in input prompt phrasing occur. This makes them unsuitable for realistic settings where user inputs are rarely repeated verbatim.

Overall, Semantic Similarity Caching outperformed Normalized Prompt Caching both in terms of raw speed and consistency. Its ability to handle input variation robustly makes it a far more practical solution for real-world LLM optimization. The findings suggest that future work should continue to explore semantic-aware caching techniques, possibly integrating more advanced paraphrase generation models or hybrid strategies, to further improve reliability and performance under diverse usage conditions.



Contributions and Implications

This study contributes to the growing body of research on LLM optimization by:

1. **Providing comprehensive empirical validation** of caching strategies with detailed quantitative analysis across multiple models, prompt volumes, and execution runs, revealing both performance potential and reliability characteristics.
2. **Identifying critical reliability factors** that distinguish practical caching solutions from theoretical approaches, demonstrating that consistency of performance is as important as peak performance for real-world deployment.
3. **Establishing scalability benchmarks** showing how different caching strategies perform across varying workload sizes, with Semantic Similarity Caching maintaining effectiveness even as prompt volumes increase from 50 to 200.
4. **Revealing the limitations of exact-match approaches** through detailed analysis of Normalized Prompt Caching performance degradation, providing important insights for future caching system design.



Final Remarks

While this study advances understanding of prompt caching in LLMs, several challenges remain. The dynamic nature of user queries, evolving model architectures, and the need for real-time adaptability necessitate ongoing innovation in caching mechanisms. Future work should explore adaptive thresholding, integration with distributed inference systems, and methods to mitigate quality degradation in cached responses.

Ultimately, as LLMs continue to permeate diverse domains, optimizing their efficiency without sacrificing accuracy will remain a critical research frontier. Caching, when implemented judiciously, represents a powerful tool in this endeavor—balancing computational savings with the demand for responsive, high-quality AI interactions.



List of Figures

Figure 1 - GPTCache: The architecture comprises of six core components: adapter, pre-processor, embedding generator, cache manager, similarity evaluator, and post-processor.	5
Figure 2 - FINCH processes input documents that are too large for the model's context by breaking them into chunks.	6
Figure 3 - Cache Augmented Generation architecture.	6
Figure 4 - The procedure of calling LLMs with or without cache.	7
Figure 5 - Llama 3.2 Execution Time Comparison.	23
Figure 6 - Llama 3.2 Cache Hit Rate Comparison.	25
Figure 7 - Llama 3.2 Speedup Ratio.	26
Figure 8 - Llama 3.2 Cache Efficiency Score.	27
Figure 9 - Llama 3.2 Query Time Over Time by Cache Type.	28
Figure 10 - Llama 3.1 Execution Time Comparison.	29
Figure 11 - Llama 3.1 Cache Hit Rate Comparison.	30
Figure 12 - Llama 3.1 Speedup Ratio.	31
Figure 13 - Llama 3.2 Cache Efficiency Score.	32
Figure 14 - Llama 3.2 Query Time Over Time By Cache Type.	33

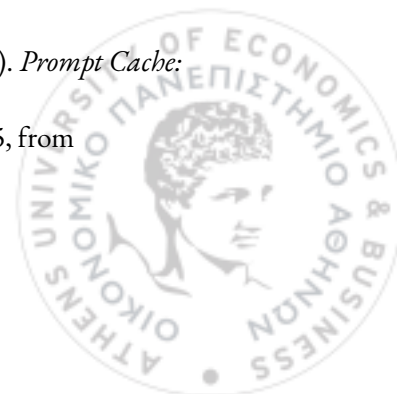
List of Tables

Table 1 - LongBench examples, Normalized Prompt Caching, Baseline run (No cache)	11
Table 2 - LongBench examples, Normalized Prompt Caching, Hits	12
Table 3 - LongBench examples, Semantic Similarity Caching, Baseline run (No cache)	13
Table 4 - LongBench examples, Semantic Similarity Caching, Hits	14
Table 5 - Llama 3.2 Cached Run Times (50 prompts)	21
Table 6 - Llama 3.2 Cached Run Times (100 prompts)	22
Table 7 - Llama 3.2 Cached Run Times (200 prompts)	22
Table 8 - Llama 3.1 Cached Run Times (200 prompts)	22
Table 9 - Summary of Improvements	22



References

- Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., Dong, Y., Tang, J., & Juanzi Li. (2024). *LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding*. ACL Anthology. <https://aclanthology.org/2024.acl-long.172/>
- Bang, F. (2023). *GPTCache: An Open-Source Semantic Cache for LLM Applications Enabling Faster Answers and Cost Savings*. Association for Computational Linguistics. <https://aclanthology.org/2023.nlposs-1.24/>
- Chan, B. J., Chen, C.-T., Cheng, J.-H., & Huang, H.-H. (2024, Dec 20). *Don't Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks*. arxiv. <https://arxiv.org/html/2412.15605v1>
- Corallo, G., & Papotti, P. (2024, 12). *FINCH: Prompt-guided Key-Value Cache Compression for Large Language Models*. Retrieved 01 10, 2025, from https://direct.mit.edu/tacl/article/doi/10.1162/tacl_a_00716/125280
- Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. OpenReview. Retrieved January 10, 2025, from https://proceedings.neurips.cc/paper_files/paper/2022/file/c3ba4962c05c49636d4c6206a97e9c8a-Paper-Conference.pdf
- Devlin, J., & Chang, M.-W. (2018, Nov. 2). *Open Sourcing BERT: State-of-the-Art Pre-training for Natural Language Processing*. Google Research. <https://research.google/blog/open-sourcing-bert-state-of-the-art-pre-training-for-natural-language-processing/>
- Gim, I., Chen, G., Lee, S., Sarda, N., Khandelwal, A., & Zhong, L. (2024, Apr. 25). *Prompt Cache: Modular Attention Reuse for Low-Latency Inference*. arxiv. Retrieved Jan. 10, 2025, from <https://arxiv.org/pdf/2311.04934v2>



Joshi, A., & Alake, R. (2024, Aug. 13). *Adding Semantic Caching and Memory to Your RAG Application Using MongoDB and LangChain*. MongoDB.

<https://www.mongodb.com/developer/products/atlas/advanced-rag-langchain-mongodb/>

Kwon, W., Li, Z., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023).

Efficient Memory Management for Large Language Model Serving with Paged Attention.

<https://arxiv.org/abs/2309.06180>

Luohe, S., Hongyi, Z., Yao, Y., Zuchao, L., & Hai, Z. (2024). *Keep the Cost Down: A Review on*

Methods to Optimize LLM's KV-Cache Consumption. <https://arxiv.org/html/2407.18003v1>

Markjbrown. (2024, December 3). *Semantic cache for large language models - Azure Cosmos DB*.

Microsoft Learn. Retrieved February 1, 2025, from [https://learn.microsoft.com/en-us/azure/cosmos-](https://learn.microsoft.com/en-us/azure/cosmos-db/gen-ai/semantic-cache)

[db/gen-ai/semantic-cache](https://learn.microsoft.com/en-us/azure/cosmos-db/gen-ai/semantic-cache)

Maroulis, S., Stamatopoulos, V., Papastefanatos, G., & Terrovitis, M. Visualization-aware Time Series

Min-Max Caching with Error Bound Guarantees. *Proc. VLDB Endow.* 17(8): 2091-2103 (2024).

<https://www.vldb.org/pvldb/vol17/p2091-maroulis.pdf>

Prompt caching with Claude. (2024, August 14). Anthropic. Retrieved January 10, 2025, from

<https://www.anthropic.com/news/prompt-caching>

Robinson, M., Duncan, F., Worthington, A., Wilson, J., & Harris, S. (2024, 10 22). *Optimizing Large*

Language Models: A Novel Approach Through Dynamic Token Pruning. Retrieved 01 11, 2025, from

[https://assets-eu.researchsquare.com/files/rs-5293588/v1_covered_52b30393-790e-4ab2-8bee-](https://assets-eu.researchsquare.com/files/rs-5293588/v1_covered_52b30393-790e-4ab2-8bee-e6e7d4b16895.pdf?c=1729565713)

[e6e7d4b16895.pdf?c=1729565713](https://assets-eu.researchsquare.com/files/rs-5293588/v1_covered_52b30393-790e-4ab2-8bee-e6e7d4b16895.pdf?c=1729565713)

Romero, F., Li, Q., Yadwadkar, N. J., & Kozyrakis, C. (2021). *INFaaS: Automated Model-less Inference*

Serving. chrome-



extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.usenix.org/system/files/atc21-romero.pdf

Srivatsa, V., He, Z., Abhyankar, R., Li, D., & Yiyang Zhang. (2024). *Preble: Efficient Distributed Prompt Scheduling for LLM Serving*. University of California, San Diego.

<https://escholarship.org/uc/item/1bm0k1w0#main>

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). *Attention Is All You Need*. <https://arxiv.org/abs/1706.03762>

Yu, G.-I., Jeong, S., Kim, G.-W., Kim, S., & Chun, B.-G. (2022). *Orca: A Distributed Serving System for Transformer-Based Generative Models*. chrome-

extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.usenix.org/system/files/osdi22-yu.pdf

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., & Sheng, Y. (2023). *SGLang: Efficient Execution of Structured Language Model Programs*. <https://arxiv.org/abs/2312.07104>

Zhu, H., Zhu, B., & Jiao, J. (2024). *Efficient Prompt Caching via Embedding Similarity*. Department of Electrical Engineering and Computer Sciences, UC Berkeley ‡Department of Statistics, UC Berkeley. <https://arxiv.org/pdf/2402.01173>

