האוניברסיטה העברית בירושלים
THE HEBREW UNIVERSITY OF JERUSALEM

Semantic Caching for LLM-Based Applications

By Aviv Gelfand

Submitted to: Adva Liberman and Lihi Nahum

Practical Workshop 55507

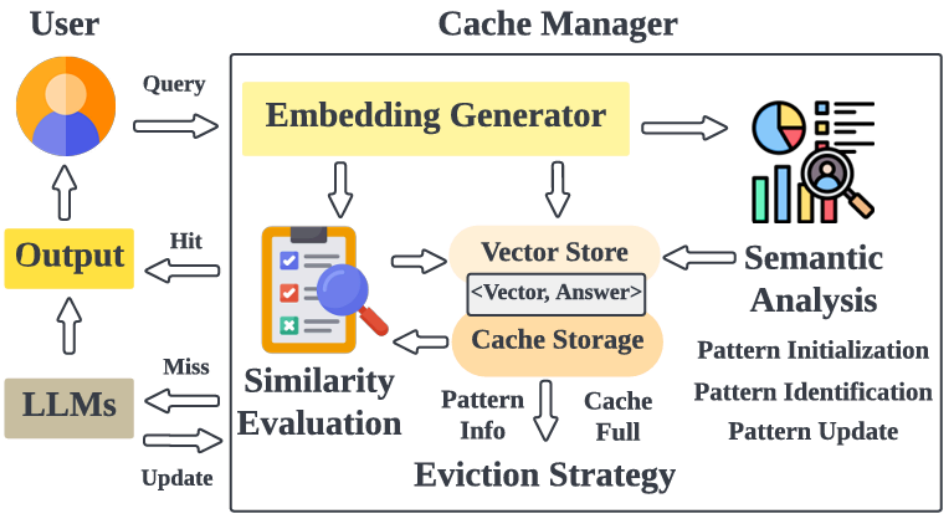


Table of Contents

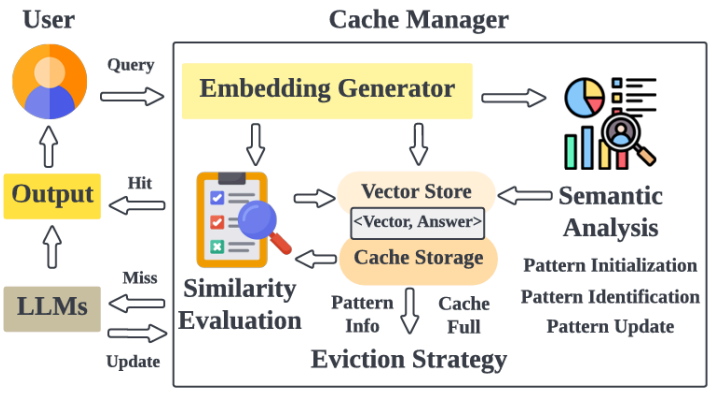
1. The Research Project: Motivation & Goal

During my internship at MemorAI, I was focused on coding, searching, and retrieving large volumes of text and applying Large Language Models (LLMs) and Language Models as a Service (LMaaS):

LLMs excel in NLP tasks[Yang et al., 2023] but are typically accessed via API due to heavy computational requirements. API access allows novice users to solve complex tasks without extensive resources [Yu et al., 2023]. Costs include components of input (prompt length) and output (generated sequence length). User experience is affected by performance, response time, and costs.

Effective LLM Invocation: Wang et al. (2023) frame it as a multi-objective optimization problem with four components: input abstraction, *semantic cache*, solution design, and output enhancement.

Semantic Caching Improves efficiency by storing and retrieving semantic information. **Workflow:** Query submission, embedding generation, similarity evaluation, cache hit/miss, LLM processing, semantic analysis, and output.



Benefits: Reduced computation and data transmission costs, support for high concurrent requests, low-cost and low-latency performance. **Project requirements**: Research, self-directed learning, experimentation, methodological thinking, and statistical analysis for model accuracy assessment.

---

2. Experiment Design and Planning

This chapter details an experiment designed to evaluate the effectiveness of semantic caching in an LLM-based system, contrasting its performance with a baseline that does not use caching.

Objective:

To evaluate the impact of semantic caching on the performance and cost-efficiency of an LLM-based system.

Hypotheses:

1. Performance Hypothesis: Semantic caching reduces the response time for repeated queries.
2. Cost Hypothesis: Semantic caching reduces the overall cost of query processing.

Tools and Components:

- ☑ **Python** code and libraries for simulations, data manipulation and analysis.
- ☑ **Transformers** (Hugging Face): For tokenization and embeddings using "microsoft/MiniLM-L12-H384-uncased".
- ☑ **FAISS (Facebook AI Similarity Search)**: Efficient similarity search code library.
- ☑ **Groq**: API client for LLM interactions.
- ☑ **OrderedDict**: Implements ordered cache with efficient item eviction

Initial Dataset

Quora Question Pairs Dataset - A benchmark for natural language understanding tasks with over 400,000 question pairs, each labeled to indicate if the questions are duplicates, providing a rich source of varied and semantically diverse queries.

Methodology:

1. Data Collection: Queries: A diverse set of queries will be collected to simulate real-world usage. Responses: For each query, responses will be generated using the LLM, both with and without caching.

2. Experimental Setup: Control Group (No Cache): Queries will be processed by the LLM without using a cache. Experimental Group (With Cache): Queries will be processed by the LLM with semantic caching enabled.

3. Variables: Independent Variable: Use of semantic caching (True or False). Dependent Variables: Response time, cost of API calls, cache maintenance cost, and cache hit rate.

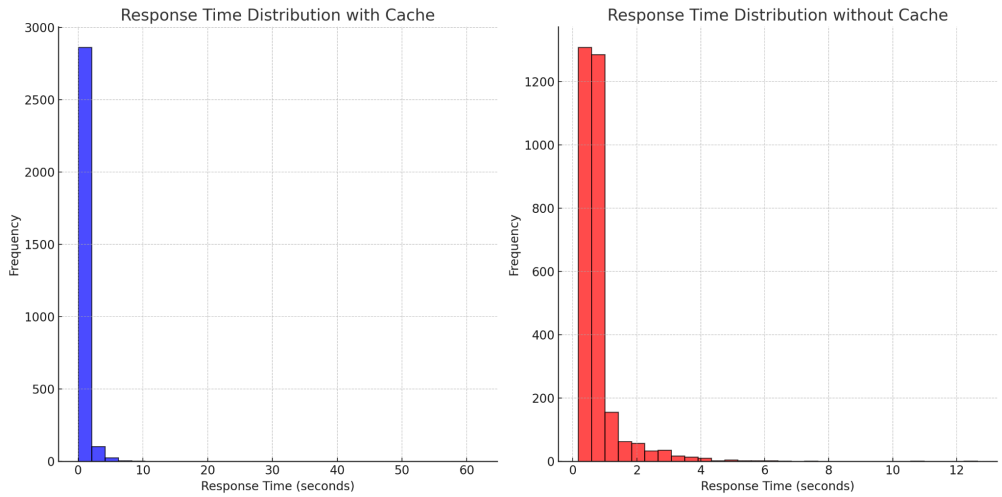
4. Data Analysis: Performance Analysis: Compare response times between the control and experimental groups, statistical tests, and Cost-Benefit Analysis.

3. Experiments and Data Analysis

I ran the experiment 30 times with different random states to ensure the robustness and generalizability of the results. For each run, queries are processed with and without caching, and the results are saved to a combined CSV file.

By comparing response times and cache hit rates across different experimental runs and conditions, researchers can draw meaningful conclusions about the benefits and limitations of semantic caching in real-world applications.

The Impact of Semantic Cache on Running Time Resources



With Cache: The response times are more spread out, with a noticeable number of longer response times. There are some extreme values (outliers) with very high response times.

Without Cache: The response times are more concentrated and have a shorter range than the cached responses. The distribution is more uniform without extreme outliers.

Statistical Test Results

To further analyze the differences, we'll perform statistical tests to determine if the differences in response times between the two groups are significant.

1. **Kolmogorov-Smirnov Test**: KS Statistic: 0.169, P-value: approaches to 0. The very low p-value indicates a significant difference between the distributions of response times with and without caching.
2. **T-test**: T Statistic: -1.999, P-value: 0.0457. The p-value is less than 0.05, indicating a statistically significant difference in the mean response times between the two groups.

Cost-Benefit Analysis

Assumptions: LLM API cost: $0.01/call, Cache maintenance: $0.0001/query

Results:

1. Cost with cache: $30.60

- API calls: $30.00 (503 hits, 2497 misses)

- Cache maintenance: $0.60

2. Cost without cache: $60.00 (6000 queries)

Net savings: $29.40

Conclusion and Recommendations

This research assessed the impact of semantic caching on an LLM-based application, focusing on response times and cost-effectiveness. The savings can be substantial, highlighting the cost-effectiveness of implementing a cache system. In conclusion, semantic caching offers significant cost savings and performance improvements. Practical implications include significantly reducing API call expenses, allowing budget reallocation, and improving user satisfaction due to faster and more consistent response times, although managing variability remains crucial.

I recommend the company optimize its caching strategy by enhancing caching algorithms, dynamically adjusting cache sizes, implementing real-time cost tracking, planning for scalability, investigating high response time outliers, and integrating user feedback to refine and improve overall performance and user satisfaction.

Future research should use actual cost data for more accurate analysis, closely control and analyze factors like query characteristics and system load, and explore advanced caching techniques to improve efficiency.

By addressing the identified limitations and implementing these recommendations, the organization can optimize its caching strategy, enhance user experience, and achieve significant cost savings.

Bibliography

[1] L. Zheng et al., "Lmsys-chat-1m: A large-scale real-world llm conversation dataset," arXiv preprint arXiv:2309.11998, 2023.

[2] T. Sun et al., "Moss: Training conversational language models from synthetic data," arXiv preprint arXiv:2307.15020, 2023.

[3] L. Chen, M. Zaharia, and J. Zou, "Frugalgpt: How to use large language models while reducing cost and improving performance," arXiv preprint arXiv:2305.05176, 2023.

[4] D. Wu, X. Wang, Y. Qiao, Z. Wang, J. Jiang, S. Cui, and F. Wang, "Large language model adaptation for networking," arXiv preprint arXiv:2402.02338, 2024.

Appendix

A) Extended Version of Motivation for Semantic Caching Project

LLMs are essential for various NLP tasks [Yang et al., 2023], demonstrating abilities like in-context learning, multi-step reasoning, and instruction following. Due to commercial reasons, potential misuse risks, and high tuning costs, LLMs like GPT-3, GPT-4, and Claude are typically provided via API rather than open-sourced, known as Language Models as a Service (LMaaS) [Yu et al., 2023].

Accessing these LLMs through APIs allows novice users to solve tasks without extensive computational resources or deep learning expertise. However, using LLM services can be expensive, especially for high-throughput applications. Costs consist of two components: (1) input cost, based on input prompt length, and (2) output cost, based on generated sequence length (see table 1 in the appendix).

Cost considerations and factors like performance and response time impact the user experience of LLM services. High latency harms user experience, while rising costs hinder scalability. Different languages, prompt methods, and enhancements can significantly alter performance [Ahia et al., 2023; Lai et al., 2023]. Affordable LLMs often complement expensive ones; for example, GPT-4 mistakes 11% of questions on the CoQA dataset, whereas the cheaper GPT-J provides correct answers [Chen et al., 2023].

Since pricing doesn't always correlate with user experience, exploring effective invocation methods for LLM services is crucial. We aim to use various LLM services to develop effective invocation strategies for different scenarios.

Wang et al. (2023) analyze effective invocation methods in LMaaS, framing it as a multi-objective optimization problem considering latency, performance, and cost. Their taxonomy categorizes methods into four components: (1) input abstraction, (2)

semantic cache, (3) solution design, and (4) output enhancement. These components can be combined into a cohesive framework, as shown in the following figure.

B) About Semantic Caching

The chatbot algorithm being developed can benefit significantly from the integration of a **Semantic Caching** feature. Yahli, the CEO, explained the necessity of this project, and I took the lead.

A semantic cache improves LLM invocation efficiency by storing and quickly retrieving semantic information, focusing on meaning and context rather than raw data. Before invoking the service, the semantic cache is checked; if there's a hit, the cached output is returned directly.

The following figure shows the general workflow of how LLM-based systems, like chatbots, use Semantic Similarity Cache.

Step-by-Step Breakdown:

1. **User Query Submission**: User submits a query.
2. **Embedding Generation**: The Cache Manager converts the query into a vector for efficient comparison.
3. **Similarity Evaluation**: The query vector is compared against vectors in the Vector Store.
4. **Cache Hit or Miss**:
    - **Hit**: If a similar vector is found, the corresponding answer is retrieved and returned.

    - **Miss**: If no similar vector is found, the query is forwarded to LLMs for processing.
5. **LLM Processing and Update**: LLMs generate a response, which is added to the Vector Store and Cache Storage.
6. **Semantic Analysis**:
    - **Pattern Initialization**: Identifying new patterns in user interactions.
    - **Pattern Identification**: Recognizing recurring patterns to improve cache efficiency.
    - **Pattern Update**: Updating patterns with the latest data.
7. **Eviction Strategy**: When cache storage is full, less relevant data is removed based on pattern information.
8. **Output to User**: The response is delivered back to the user.

This workflow leverages semantic similarity caching to enhance response times and optimize resources. Cache technology reduces computation and data transmission costs, supports high concurrent requests, and provides low-cost, low-latency performance [Miao et al., 2023].

This project requires research, self-directed learning, experimentation, methodological thinking, and statistical analysis to assess model accuracy. I am enthusiastic about its relevance to any AI-using organization.

Provider	**LLM**	**Input Cost**	**Output Cost**
OpenAI	gpt-4	$30.00	$60.00
	gpt-4-turbo	$10.00	$30.00

Provider	LLM	Input Cost	Output Cost
	gpt-3.5-turbo-1106	$1.00	$2.00
Anthropic	Claude-2.0	$11.02	$32.68
	Claude-instant-1.2	$1.63	$5.51
AI21	Jurassic-2 Ultra	$15.00	$15.00
	Jurassic-2 Mid	$10.00	$10.00
	Jurassic-2 Light	$3.00	$3.00

Table 1: Price list of different LMaaS. The cost is priced per 1 million tokens. The data was updated to 24 January 2024.

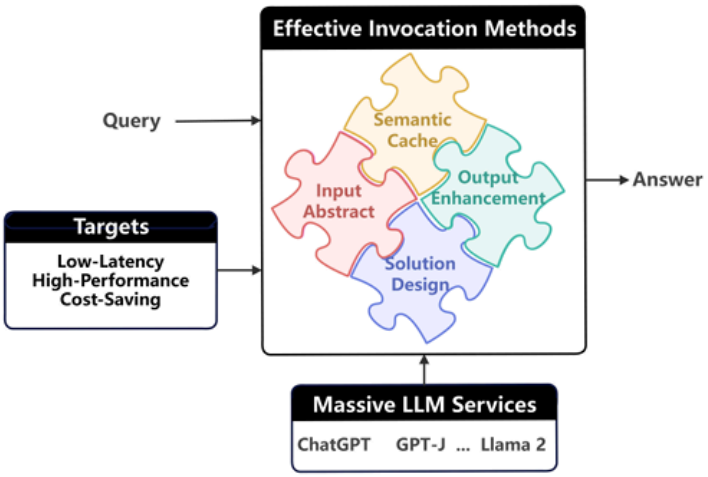


Figure 1: Vision of efficient invocation strategy construction for massive LLM services.

The final DataFrame, `all_results_df`, contains detailed information about each query processed during the experiment. The columns in this DataFrame include:

1. `qid`: The unique identifier for each query.

2. `question`: The text of the query.

3. `response`: The generated response to the query.

4. `response_time`: The time taken to generate the response.

5. `cache_hit`: A boolean indicating whether the response was retrieved from the cache (`True` or `False`).

6. `similarity_score`: The similarity score between the query and the closest cached query, if a cache hit occurred. This will be `None` if no cache hit occurred.

7. `similar_text`: The text of the most similar cached query, if a cache hit occurred. This will be `None` if no cache hit occurred.

8. `random_state`: The random state used when sampling the dataset for the experiment, helping to identify different runs of the experiment.

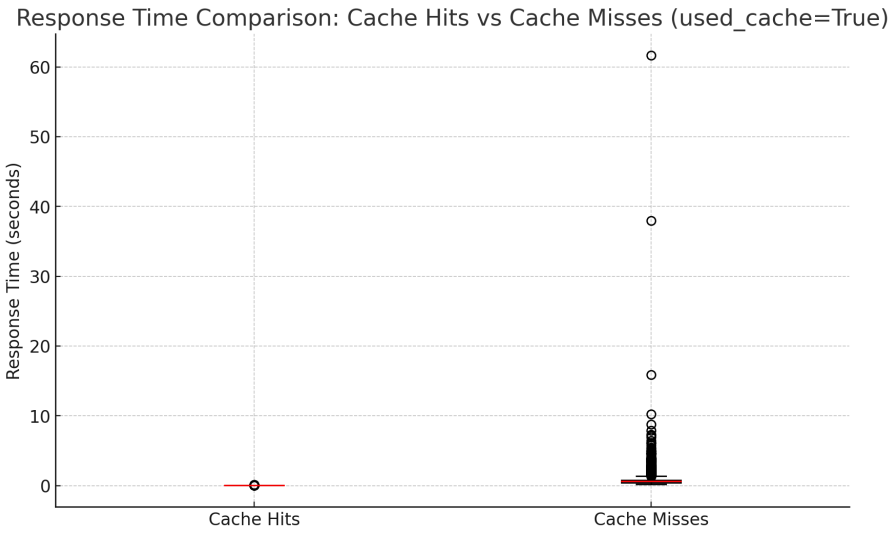
9. `exp_num`: An additional column added during loading to indicate the experiment number, which corresponds to the `random_state`.

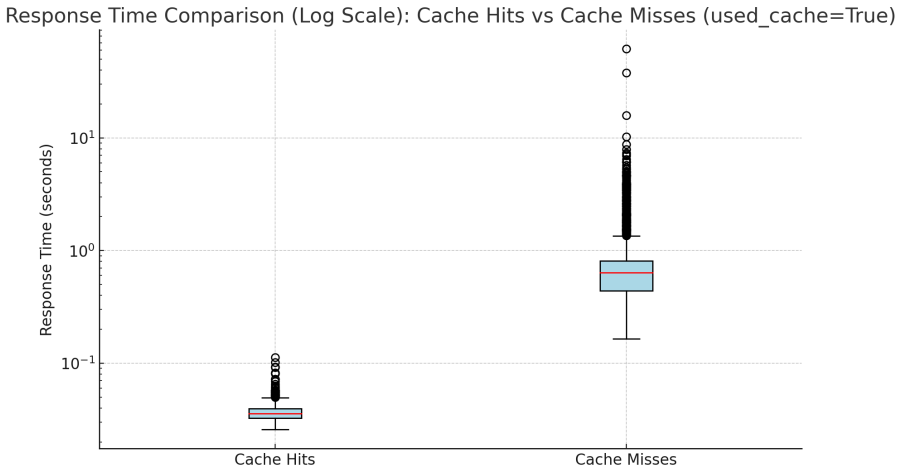
10. `used_cache`: A boolean indicating whether caching was used in this particular experiment run (`True` for `results_with_cache` and `False` for `results_no_cache`).

C) Cache Hit Impact Analysis

Within the used_cache=True group, we can further analyze the impact of cache hits on response times.

1. **Compare response times for cache hits and misses within the used_cache=True group**.
2. **Visualize the difference** using boxplots.





D) Observations from Boxplots

1. **Cache Hits**:
    - The median response time for cache hits is lower than for cache misses.
    - The interquartile range (IQR) is smaller, indicating less variability in response times.
2. **Cache Misses**:

- The response times are more spread out, with some extreme outliers.
- The median response time is higher compared to cache hits.

E) Detailed Cost-Benefit Analysis Results

1. **Total Cost with Cache**: $30.60
   - **Cost of API Calls for Cache Hits**: 503 hits×$0.01=$5.03503 hits X $0.01 = $5.03503 hits×$0.01=$5.03
   - **Cost of API Calls for Cache Misses**: 2497 misses×$0.01=$24.972497 misses X $0.01 = $24.972497 misses×$0.01=$24.97
   - **Cost of Cache Maintenance**: 6000 queries×$0.0001=$0.606000 queries X $0.0001 = $0.606000 queries×$0.0001=$0.60
2. **Total Cost without Cache**: $60.00
   - **Cost of API Calls for All Queries**: 6000 queries×$0.01=$60.006000 queries X $0.01 = $60.006000 queries×$0.01=$60.00
3. **Net Savings**: $29.40

F) Link to the Code Notebook