SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY
- INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

VectorQ: Advanced Semantic Prompt Caching With Dynamic Thresholds and Performance-Based Clustering

Luis Gaspar Schroeder

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

VectorQ: Advanced Semantic Prompt Caching With Dynamic Thresholds and Performance-Based Clustering

VectorQ: Effizientes semantisches Prompt-Caching mit dynamischen Schwellenwerten und leistungsbasiertem Clustering

Author:	Luis Gaspar Schroeder
Advisor:	Prof. Alfons Kemper
Supervisors:	Prof. Joseph E. Gonzalez, Shu Liu
Submission Date:	26.11.2024

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Berkeley, United States, 26.11.2024 Luis Gaspar Schroeder

Acknowledgments

I am deeply grateful to my supervisors, Prof. Joseph Gonzalez and Shu Liu, from the Sky Lab at UC Berkeley, for their dedication and guidance throughout this project. I appreciate Joseph Gonzalez's multidisciplinary thinking and his supportive, optimistic approach, which allowed me to explore various methodologies. His encouragement taught me to embrace an iterative process, challenge my thinking, and find value in every attempt. I would also like to thank Prof. Alfons Kemper from TU Munich for his guidance and support during my academic journey at TU Munich. Their mentorship has been instrumental in my development as a researcher.

Abstract

State-of-the-art semantic prompt caches depend on manually defined static similarity thresholds and employ least recently used (LRU) cache eviction policies. Manual threshold adjustment is challenging because it depends on altering prompt complexity and the embedding model. Incorrect thresholds either reuse wrong responses or trigger unnecessary LLM inferences. This requires constant threshold monitoring, which is inefficient. LRU policies for semantic prompt cache management do not correct errors in embedding mappings and result in an inaccurate cache population. This research introduces an online heuristic algorithm that dynamically adjusts the threshold based on a user-defined error rate target and proposes a performance-based cache eviction policy. On top of this, we build the advanced semantic prompt cache VectorQ that supports the integration with any inference server or embedding model. We demonstrate VectorQ's performance across three datasets, achieve a 9x reduction in error rates, and maintain comparable latency performance compared to state-of-the-art semantic prompt caches.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 Background **3**
2.1 Large Language Models . . . 3
2.1.1 Inference Server . . . 4
2.1.2 Caching Challenges . . . 4
2.2 DBMS . . . 5
2.2.1 UDFs . . . 5
2.2.2 Query Optimization . . . 5
2.2.3 Vector Databases . . . 6
2.2.4 Vector Embeddings . . . 6
2.2.5 Vector Embedding Similarity . . . 6

3 Motivation **8**

4 Related Work **10**
4.1 Semantic Prompt Caching . . . 10
4.2 Inference-optimized Systems . . . 11
4.3 Serving Frameworks . . . 11
4.4 Active Online Learning Nearest Neighbor Classifier . . . 11

5 VectorQ **13**
5.1 Architecture . . . 13
5.1.1 Building Blocks . . . 14
5.2 Dynamic Threshold . . . 17
5.2.1 Epochs . . . 17
5.2.2 Abstract Algorithm . . . 18
5.2.3 Error Rate and Reuse Rate . . . 20

5.2.4 Cluster Rate 20
5.2.5 Weighted Scaling 21
5.2.6 Threshold Factors 24
5.2.7 Algorithm 28
5.2.8 An Active Online Learning Nearest Neighbor Classifier 29
5.3 Re-Clustering and Cache Eviction 30
5.3.1 Create Cluster 31
5.3.2 Update Existing Cluster 31
5.3.3 Evict Cluster 34

6 Implementation **36**
6.1 Multiple User Resource Management 36
6.2 UML Architecture 36

7 Evaluation **39**
7.1 Baseline 1: Direct Inference 40
7.1.1 Accuracy 41
7.1.2 Summary 43
7.2 Baseline 2: GPT Cache 44
7.2.1 Average Case 44
7.2.2 Worst Case 46
7.3 Limitations 47

8 Conclusion **50**

List of Figures **51**

List of Tables **53**

Bibliography **54**

1 Introduction

Two high school classmates lost touch after graduation. Years later one became a data analyst and tried to query an international social media database to find his friend. How should he translate and semantically cluster database entries with traditional SQL? While SQL excels at structured queries, Large Language Models (LLMs) such as LLaMA (Touvron, Lavril, Izacard, et al., 2023) or GPT (Brown, 2020) transformed how computers approach language understanding and reasoning at scale. However, running these applications is very resource-expensive, requiring a large number of hardware accelerators such as GPUs or FPGAs (Wei, Langer, Yu, et al., 2022). For example, an NVIDIA L4 GPU running LLaMa3-8B can only process 6 KB of text per second, taking about a day to handle 15 GB of data; and costs around $10K on OpenAI's GPT-4o.

Most modern Large Language Models (LLMs) are built on autoregressive transformer architectures, which generate tokens sequentially by conditioning on a given prompt and the previously generated token sequence (Shi, Zhang, Yao, et al., 2024). During this process, the model utilizes a key-value (KV) cache to store intermediate representations, such as query, key, and value embeddings for each token. Existing research optimizes the KV cache hit rate with prefix caching (Zheng, Yin, Xie, et al., 2023) or memory management strategies like paged attention (Kwon, Li, Zhuang, et al., 2023) to reduce prompt latency. However, common LLM applications like chatbots or LLM-based SQL queries share semantically similar prompts with the same answer (Zhu, Zhu, & Jiao, 2024). Instead of only reusing prefixes to reduce the inference latency, we can use semantic prompt caches to reuse entire prompts and not make an inference call at all. A semantic prompt cache reduces redundant LLM inference calls by caching responses to prompts based on their semantic similarity (Zhu, Zhu, & Jiao, 2024). The workflow involves embedding each incoming prompt, checking for semantically similar embeddings in the cache, and retrieving the cached response if a similar prompt is found. If the cache cannot return a response, it queries the LLM and updates the cache with the new response. The architecture typically includes a vector database-based cache for storing prompt embeddings, a K-nearest neighbor similarity search to identify similar embeddings, and a threshold mechanism to determine if a reuse candidate is qualified.

Semantic prompt caches, such as GPT Cache (Bang, 2023) and Microsoft's Cosmos DB cache extension (Markjbrown, 2024), are widely used in production systems. These systems use vector embeddings to evaluate prompt similarity and rely on a threshold to decide if a cached response can be reused. To set this threshold manually is labor-intensive, error-prone, and impractical, as it varies with prompt complexity and embedding models. Incorrect thresholds reduce performance because they are either too low, reusing incorrect responses, or too high, triggering unnecessary LLM inferences. This forces users into the impractical task of constantly monitoring and adjusting the threshold to maintain accuracy and efficiency (Sudarsan & MasayaNishimaki, 2024). GPT Cache employs a static threshold and a least recently used (LRU) policy to manage the vector embedding cache; however, this approach does not address erroneous embedding mappings and results in a poorly populated and unreliable embedding cache.

Our research presents novel advancements in dynamic threshold adjustment and performance-based cache management for semantic prompt caching. We propose an online heuristic approach that dynamically adjusts the threshold based on observed performance, allowing the system to adapt across diverse datasets and varying prompt complexities. Users can specify a desired accuracy level, and the system automatically fine-tunes the threshold to optimize latency while maintaining the specified accuracy, eliminating the need for manual threshold tuning. Additionally, we introduce adaptive re-clustering mechanisms to effectively manage cache size and enhance embedding accuracy by intelligently merging related embeddings and evicting less relevant ones. In this work, we present VectorQ, a comprehensive semantic prompt cache that incorporates dynamic thresholding and adaptive re-clustering. VectorQ is designed to support the integration with any inference server or embedding model. We evaluate its performance across three datasets and demonstrate its ability to adjust to user-defined error rate targets. Compared to state-of-the-art semantic prompt caching systems, VectorQ achieves a 9x reduction in error rates and maintains comparable latency performance.

2 Background

This chapter provides the foundational knowledge to understand the concepts and techniques explored in this thesis. It introduces the architecture and challenges of transformer-based large language models (LLMs), including their inference and caching constraints. We outline database management system (DBMS) concepts, introduce query optimization, user-defined functions (UDFs), and the role of vector databases in similarity-based retrieval. Together, these topics create a comprehensive background to understand the intersection of LLMs and semantic prompt caching in modern computational systems.

2.1 Large Language Models

Transformer-based large language models (LLMs) are designed to model the probability distribution of a sequence of tokens to capture dependencies between elements in sequential data. They employ factorization to decompose the joint probability of a token sequence into a product of conditional probabilities to predict the next token based on all preceding tokens (Bengio, Ducharme, & Vincent, 2000). The core mechanism is self-attention, where for each input token, a query vector interacts with the key and value vectors of all other tokens to compute attention scores. These scores determine the relative importance of tokens in the sequence and enable the model to weigh contextual information effectively.
The architecture of a Transformer model (Vaswani, 2017) includes components such as embedding layers, feed-forward layers, and residual connections, where the self-attention layer captures relationships across token positions. Transformer-based LLMs operate in two distinct phases. The prompt phase processes an input sequence in parallel to compute probabilities for initial tokens, and the autoregressive generation phase which sequentially predicts tokens one by one. These phases enable the models to handle tasks such as text completion, generation, and classification, but optimizing their inference performance remains a critical challenge.

2.1.1 Inference Server

An inference server facilitates the deployment and execution of pre-trained machine-learning models to process and return client requests.

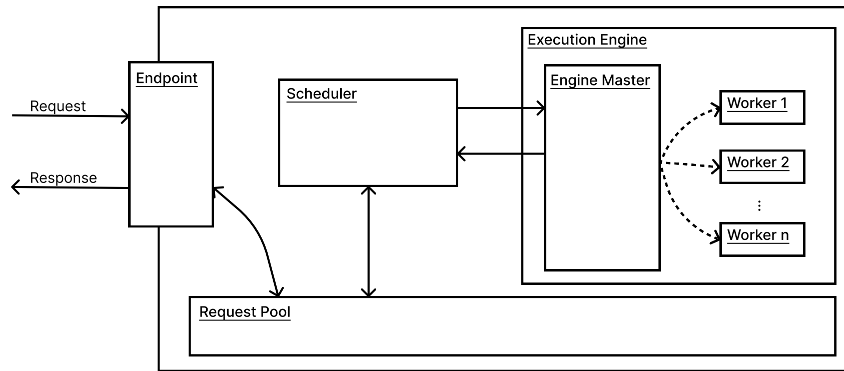


Figure 2.1: Inference server architecture.

The **Endpoint** enables clients to interact with the inference server. The **Request Pool** temporarily stores incoming requests in a queue and the **Scheduler** schedules the requests sequentially or in batches. The **Engine Master** serves as the central controller, routes requests among available resources, and supervises the execution process. The **Workers** execute inference tasks in parallel and utilize computational resources such as GPUs or FPGAs (Wei, Langer, Yu, et al., 2022).

2.1.2 Caching Challenges

The GPU's KV cache stores the key and value vectors generated during the forward pass of a transformer-based LLM. This allows the model to reuse key and value vectors during autoregressive generation, reducing redundant computations for previously processed tokens. The cache grows with the number of tokens and layers and the memory requirements scale with request size and batch count (Shi, Zhang, Yao, et al., 2024). Efficient management of the KV cache is critical to optimize GPU memory usage, as inadequate handling can constrain throughput and increase latency in LLM serving.

2.2 DBMS

Database Management Systems (DBMS) are used to organize, store, and retrieve data in a structured and efficient manner. They provide extensibility through features such as User-Defined Functions (UDFs), which allow users to define custom operations. Query optimization ensures that the database executes queries efficiently to reduce latency and resource usage. Vector databases introduce semantic search capabilities and enable unstructured data operations through vector embeddings that capture contextual and semantic relationships.

2.2.1 UDFs

User-Defined Functions (UDFs) are custom functions that extend the capabilities of Database Management Systems (DBMS) by allowing users to define specific operations beyond the scope of standard SQL. These functions are typically written in procedural languages supported by the DBMS, such as PL/pgSQL or PL/Python. They can encapsulate complex logic or integrate external processing directly within a database query. UDFs enable applications to perform data transformations, advanced calculations, and external system integrations directly within the database environment (Hsu, Chen, Wu, et al., 2010). Database users can integrate external services, such as LLM inference servers, by making API calls through UDFs. This approach enables real-time processing of queries that require external computation, such as generating embeddings or performing classification tasks. For example, a Python-based UDF can utilize the requests library to send an API request to an inference server and return the response as part of a database query (Friedman, Pawlowski, & Cieslewicz, 2009).

2.2.2 Query Optimization

Query optimizers in database management systems generate an efficient execution plan for a given query by considering factors like index usage, join order, and data distribution. The optimizer relies on deterministic cost models to estimate the execution time and resource requirements of each query operation (Jarke & Koch, 1984). However, UDFs present a challenge because their behavior is often non-deterministic and dependent on external factors such as API responses. The optimizer cannot predict or analyze the execution cost of UDFs due to their lack of transparency and reliance on external environments. As a result, queries involving UDFs are typically executed row by row (Franz, Arch, Hirn, et al., 2024), which bypasses many optimization techniques and leads to increased latency.

2.2.3 Vector Databases

Vector databases are specialized database systems designed to store, retrieve, and manage high-dimensional vector representations of data. These vectors, often generated by machine learning models, encode the semantic meaning of unstructured data, such as text, images, or audio, into a numerical format suitable for similarity-based retrieval (Malkov & Yashunin, 2018). The primary goal of a vector database is to efficiently support operations like nearest neighbor searches, which are essential for applications such as recommendation systems, semantic search, and machine learning pipelines.
Vector databases consist of several components that enable storage and retrieval of high-dimensional vectors. **Vector indexing** structures, such as HNSW (Malkov & Yashunin, 2018) or FAISS (Douze, Guzhva, Deng, et al., 2024), organize vectors based on their proximity to support fast approximate nearest neighbor searches. The **storage engine** handles the vector data and incorporates hybrid storage to combine vector embeddings with metadata. **Similarity metrics** like cosine similarity, Euclidean distance, or inner product are used to measure vector closeness.

2.2.4 Vector Embeddings

Vector embeddings are numerical representations that encode the semantic or contextual meaning of data in a fixed-dimensional vector space and enable tasks like similarity search, clustering, and classification. These embeddings are mainly generated with transformer-based or bi-directional machine learning models (Grohe, 2020). **Transformer-based** models, such as GPT (Brown, 2020), generate embeddings using attention mechanisms to weigh the relationships between tokens. They operate sequentially and predict tokens one at a time and condition on prior context. **Bi-directional** models, such as BERT (Alsentzer, Murphy, Boag, et al., 2019), compute embeddings by considering both past and future tokens simultaneously and capture a more holistic context. The key difference is their focus: transformer-based models prioritize sequential generation, while bi-directional models emphasize holistic context understanding.

2.2.5 Vector Embedding Similarity

Vector embedding similarity quantifies the relationship between two embeddings by measuring their closeness in a high-dimensional vector space.

Cosine Similarity

Cosine similarity measures the similarity between two vectors by calculating the cosine of the angle between them in a high-dimensional space. It quantifies how aligned the vectors are regardless of their magnitude (Xia, Zhang, & Li, 2015). The formula for cosine similarity is:

$$\text{Cosine Similarity}(A, B) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

where $A \cdot B$ is the dot product of vectors A and B, and $\|A\|$ and $\|B\|$ are their respective magnitudes, computed as the Euclidean norm. For our dynamic threshold, we normalize the cosine similarity $\cos(A, B)$ to the interval $[0, 1]$, where 1 corresponds to a perfect semantic match ($A = B$) and 0 indicates no semantic similarity ($A \perp B$).

3 Motivation

Semantic prompt caches address the computational cost and latency challenges associated with Large Language Models (LLMs), particularly in applications with repetitive or semantically similar queries (Zhu, Zhu, & Jiao, 2024). By storing embeddings of prompts and their corresponding responses, these caches enable systems to reuse answers for new, semantically similar prompts through vector similarity search. For example, in a customer support system, if multiple users ask variations of "How do I reset my password?" the system can return the cached response rather than querying the LLM repeatedly. This reduces latency and operational costs by minimizing redundant LLM calls and lowering token consumption, which is particularly beneficial in high-demand scenarios like Retrieval-Augmented Generation (RAG) where payloads can be similar and large (Markjbrown, 2024).

Semantic prompt caches are particularly useful in relational database systems that leverage Large Language Models (LLMs) for query processing. LLMs extend the capabilities of SQL by handling complex tasks such as natural language understanding, semantic classification, and contextual reasoning, which SQL alone cannot achieve. For example, an LLM can classify product descriptions, enabling category-specific analysis, which goes beyond SQL's native functionality.

```
SELECT
  id, name, description,
  LLM('Which product category does this description
  belong to? {description}', description) AS category
FROM product_description;
```

In Postgres, user-defined functions (UDFs) can call inference server APIs, such as OpenAI, to perform such tasks. However, database systems execute UDFs queries row by row due to their non-deterministic outputs that make them incompatible with the query optimizer (Franz, Arch, Hirn, et al., 2024). As a result, queries rely on synchronous execution and are constrained by the latency of the inference server. Semantic prompt caching mitigates this issue by reusing responses for semantically similar rows, which are more prevalent in structured databases. This reduces the need

for repeated inference calls and reduces query execution time.

State-of-the-art semantic prompt caches, such as GPT Cache (Bang, 2023) and implementations from Microsoft (Markjbrown, 2024) and MongoDB (Joshi, 2024), rely on user-defined static threshold values to determine whether the nearest embedding is sufficiently close for reuse. The choice of threshold significantly impacts cache performance: a low threshold increases cache hits but risks inaccuracies, while a high threshold improves accuracy but reduces reuse. How should a user select this threshold? One approach involves performing an initial analysis using the embedding model and a subset of possible prompts, testing various thresholds to find one that aligns with a desired accuracy rate. However, a static threshold proves insufficient because the optimal threshold varies with the complexity of incoming prompts—complex prompts require higher thresholds, while simpler prompts perform well with lower ones. As a result, a fixed threshold cannot guarantee consistent accuracy. While users could manually adjust the threshold over time, this approach is impractical due to the need for frequent manual changes and uncertainty about the appropriate adjustment magnitude. As a result, state-of-the-art semantic prompt caches either sacrifice accuracy or fail to achieve sufficient prompt reuse.

Our research introduces an online heuristic algorithm that dynamically adjusts the threshold based on the latest semantic prompt cache performance. Users specify a desired upper error rate bound, and the algorithm maintains this accuracy and optimizes prompt reuse. The algorithm is more robust to biased data and adapts to changes in the complexity of incoming prompts to ensure reliable performance across varying scenarios.

4 Related Work

The widespread adoption of large language models (LLMs) has driven the need for optimization techniques to manage their computational cost, latency, and scalability. Among these techniques, semantic prompt caching has emerged as a solution to reduce redundant computations by reusing responses for semantically similar prompts. However, semantic prompt caching is only one part of a larger ecosystem of optimization approaches. Inference-optimized systems, such as vLLM (Kwon, Li, Zhuang, et al., 2023) and Orca (Yu, Jeong, Kim, et al., 2022), focus on accelerating LLM execution and resource management. General-purpose prompt caching frameworks like SGLang (Zheng, Yin, Xie, et al., 2023) and INFaaS (Romero, Li, Yadwadkar, & Kozyrakis, 2021) explore strategies to improve reuse efficiency by incorporating user knowledge with a serving framework. Active online learning nearest neighbor classifiers provide theoretical frameworks for adaptive systems and offer insights into how thresholds can evolve dynamically. This chapter examines the distinctions among these approaches and positions semantic prompt caching within the broader landscape of LLM serving and adaptive learning methodologies.

4.1 Semantic Prompt Caching

GPTCache (Bang, 2023) is an open-source semantic prompt caching framework designed to optimize the efficiency of large language model (LLM) queries. It stores embeddings of prompts and their corresponding responses and uses vector similarity search to retrieve cached responses for semantically similar incoming prompts. This reduces redundant LLM inference calls, lowering both computational costs and response latency. Microsoft (Markjbrown, 2024) and MongoDB (Joshi, 2024) provide frameworks that build upon the architectural design of Prompt Cache. However, these frameworks rely on static user-defined similarity thresholds, which are inadequate for dynamic workloads where the complexity of incoming prompts fluctuates, leading to suboptimal reuse or inaccurate responses.

4.2 Inference-optimized Systems

vLLM (Kwon, Li, Zhuang, et al., 2023) and Orca (Yu, Jeong, Kim, et al., 2022) are systems designed to enhance the efficiency of large language model (LLM) serving by optimizing memory management and scheduling. vLLM introduces PagedAttention, an attention algorithm inspired by virtual memory and paging techniques, to address inefficiencies in managing the key-value (KV) cache memory for LLMs (Kwon, Li, Zhuang, et al., 2023). This approach achieves near-zero waste in KV cache memory and allows flexible sharing within and across requests, improving throughput by 2-4 times compared to existing systems like FasterTransformer and Orca. Orca focuses on efficient inference by scheduling and interleaving requests, enabling more parallel processing and increasing GPU utilization (Yu, Jeong, Kim, et al., 2022). Semantic prompt caches complement inference-optimized systems by prioritizing the reuse of existing answers whenever possible. When reuse is not feasible, inference-optimized systems ensure optimal resource utilization and reduced computational overhead.

4.3 Serving Frameworks

SGLang (Zheng, Yin, Xie, et al., 2023) is a serving framework for large language models designed to enhance interaction speed and control by co-designing the backend runtime and frontend language. It offers features like RadixAttention for efficient prefix caching and supports a range of generative models to provide an intuitive interface for programming LLM applications. INFaaS (Romero, Li, Yadwadkar, & Kozyrakis, 2021) is an inference-as-a-service system that automates model selection and scaling for deep learning inference. It dynamically chooses the most appropriate model and instance type based on user-specified latency and accuracy requirements. By integrating semantic prompt caches, these systems can further reduce latency and adjust the prompt cache based on user-defined parameters.

4.4 Active Online Learning Nearest Neighbor Classifier

The Never-Ending Learning from Time Series Streams (Hao, Chen, Zakaria, et al., 2013) framework proposes a solution for continuous learning from time series data streams. The framework leverages active learning and queries labels selectively to adapt to changing patterns within evolving data streams. It introduces motif detection as a scalable proxy for more complex patterns and enables learning with minimal labeled data while maintaining adaptability to new patterns over time. The Active Learning for

Time Series Classification (ACTS) (Peng, Luo, & Ni, 2017) approach focuses on time series classification optimization through active learning. It uses an informativeness metric that combines uncertainty and utility and is specialized to time series data. The authors adapt shapelet discovery to identify key discriminative features to efficiently select data points that maximize classifier performance while minimizing the need for labeled examples. Our dynamic threshold algorithm draws inspiration from the dynamic parameter adaptation in time series data and applies this approach to cosine similarity thresholds for semantic prompt caching.

5 VectorQ

VectorQ is an advanced semantic prompt cache that enables user-defined accuracy objectives by utilizing dynamic thresholds and embedding cluster optimizations. A semantic prompt cache reduces redundant large language model (LLM) inference calls by caching responses to prompts based on their semantic similarity. The workflow involves embedding each incoming prompt, checking for semantically similar embeddings in the cache, and retrieving the cached response if a similar prompt is found. If the cache cannot return a response, it queries the LLM and updates the cache with the new response. The architecture typically includes a vector database-based cache for storing prompt embeddings, a K-nearest neighbor similarity search to identify similar embeddings, and a threshold mechanism to determine if a reuse candidate is qualified.

5.1 Architecture

Traditional architectures process Large Language Model (LLM) queries by directly querying inference servers. Inference servers are essential as LLMs depend on specialized hardware, such as GPUs and FPGAs, to deliver reasonable performance. These servers centralize the computational load and eliminate the need for resource-heavy hardware on client machines. However, direct querying requires repeated inference computations, even for semantically similar prompts, which leads to inefficient use of hardware resources. This inefficiency increases latency and operational costs, particularly for high-throughput applications, as no caching mechanism is employed to reuse prior results.

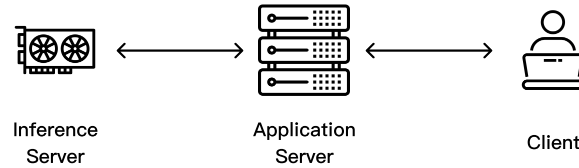


Figure 5.1: Client-server architecture without caching, where LLM requests are sent directly to the inference server.

VectorQ is a standalone server that acts as an intermediary between the application and the inference server. Upon receiving an inference request from the client, it attempts to reuse a response from a previously processed request to avoid the expensive inference server.

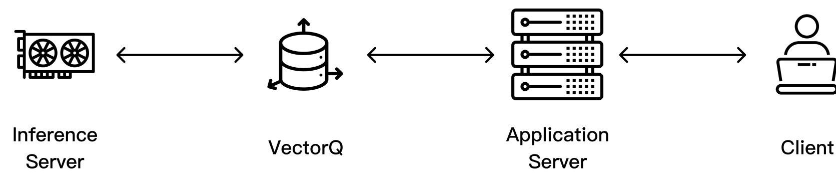


Figure 5.2: Client-server architecture with semantic VectorQ caching, where LLM requests are routed through the cache before eventually reaching the inference server.

With this architecture in place, VectorQ leverages specialized building blocks to efficiently handle, process, and cache LLM prompt requests, as detailed in the following section.

5.1.1 Building Blocks

VectorQ receives LLM prompt requests through its **API Endpoint**. The **Embedding Generator** converts each query into a vector embedding representation. The **Similar Answer Extractor** searches for a previously processed prompt with similar semantics to reuse its answer. The **Vector DB** stores embeddings of previously processed prompts and enables a K-nearest neighbors search to find the most similar prompt. The

Similarity Evaluator applies a dynamic threshold to determine whether the nearest neighbor's cosine similarity is large enough for reuse. If the similarity exceeds this threshold, the **Answer Cache** provides the answer that corresponds to the nearest neighbor. Otherwise, VectorQ performs a direct inference to generate a new answer. Subsequently, the **Similarity Evaluator** updates the dynamic threshold based on recent performance. The **Vector DB Supervisor** adds the newly generated answer to the vector database, refines the embedding clusters, and evicts low-quality embeddings. Finally, VectorQ returns the answer to the client.

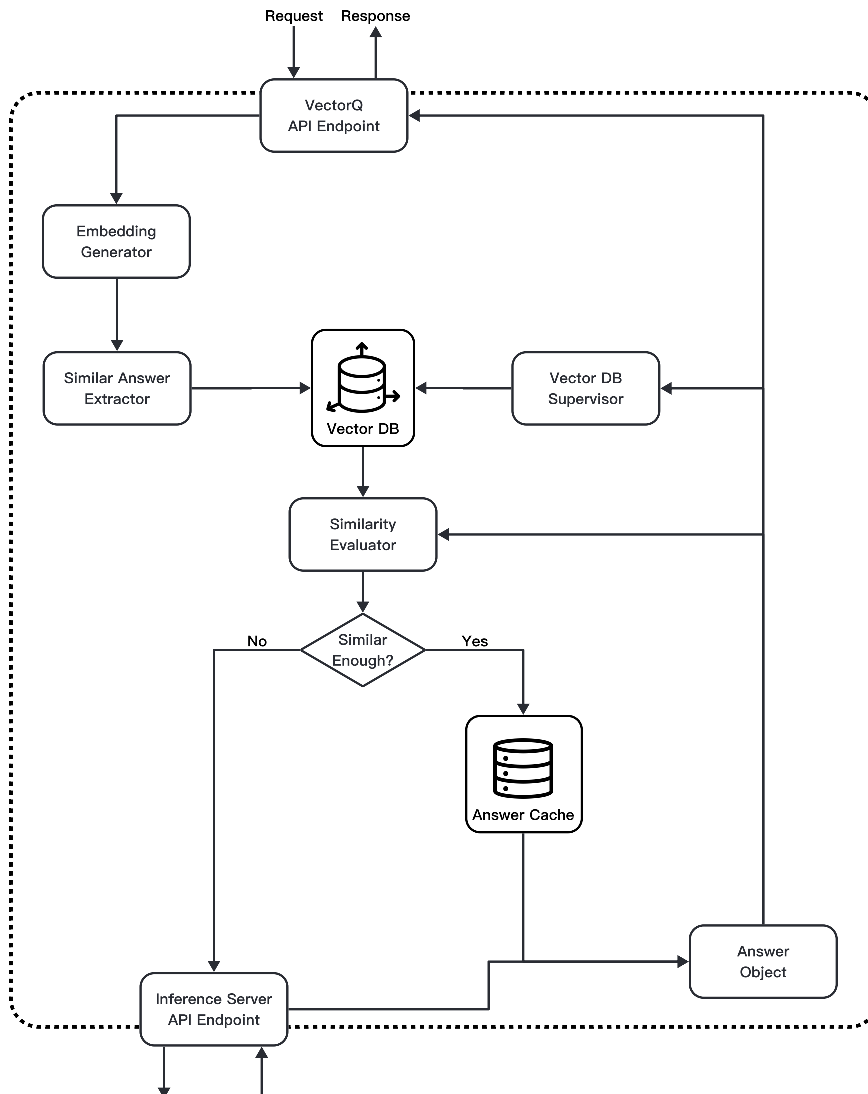


Figure 5.3: The VectorQ architecture enables efficient LLM inference by reusing LLM responses through semantic prompt caching. It consists of components for embedding generation, similarity evaluation, and dynamic thresholding, supported by a vector database, answer cache, and supervisor for cache optimization.

The following two sections explain the dynamic threshold computation and the re-

clustering and cache eviction policies.

5.2 Dynamic Threshold

We propose an online heuristic approach that integrates internal knowledge to achieve dynamic performance-based threshold adjustments. The user defines an upper bound for the error rate and the threshold adjusts to maintain the desired accuracy while optimizing the reuse rate. The key idea of this approach is to reward or punish the threshold based on its accuracy in reusing correct answers.

5.2.1 Epochs

In the dynamic threshold adjustment mechanism, epochs serve as self-contained intervals to evaluate and adjust the system's performance in response to varying prompt complexities. Each epoch approximates error and reuse rate metrics to adapt and align the threshold with a user-defined maximum error rate. At the start of each epoch, the threshold is set to its initial value, typically 1.0. The threshold dynamically adjusts during the epoch to balance accuracy and reuse efficiency.

If the error rate within an epoch exceeds the user-defined maximum, the system terminates the current epoch, resets the threshold, and initiates a new epoch. This approach allows the system to rapidly adapt to rapid changes in prompt complexity and prevents gradual drift in performance metrics. Unlike a global error rate, which averages performance over time and is slow to respond, epoch-specific error rates enable more precise, localized adjustments, making the system responsive to dynamic input conditions.

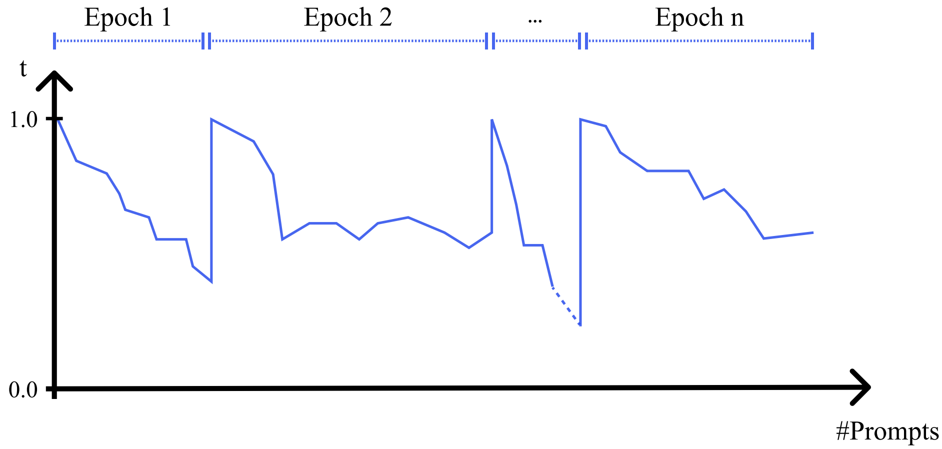


Figure 5.4: Epochs enable interval-specific error rate calculations and reset the threshold t to its maximum after the user-defined error rate is reached.

The transitions between epochs are designed to optimize reuse while minimizing performance degradation and ensure that the system remains robust when handling diverse prompts. The following subsections position this approach within the dynamic threshold framework.

5.2.2 Abstract Algorithm

This section introduces the abstract algorithm for the dynamic threshold to provide a holistic overview before we explain the components in more detail. The algorithm has two major cases. In the first case, the epoch error rate exceeds the user-defined maximum error rate. Consequently, we start a new epoch and set the threshold to its maximum value of 1.0. In the second case, we explore the ongoing epoch and adjust the threshold based on its performance. We normalize the cosine similarity $cs = \cos(A, B)$ to the interval $[0, 1]$, where 1 corresponds to a perfect semantic match ($A = B$) and 0 indicates no semantic similarity ($A \perp B$). We identified four distinct epoch scenarios that enable us to increase or decrease the cosine similarity threshold use-case specific.

In the first scenario, the system performed a direct inference because the cosine similarity between the current embedding and its nearest neighbor is below the threshold. This outcome is suboptimal because we could have avoided the expensive direct inference and returned the reused VectorQ answer instead. Consequently, we punish the threshold and decrease it. In the second scenario, the system correctly reused the

answer from VectorQ and we further decrease the threshold to increase the reuse rate.

In the third scenario, the system made a direct inference because the cosine similarity between the current embedding and its nearest neighbor is below the threshold. This outcome is optimal because the VectorQ response would have been incorrect. Consequently, we reward the threshold and increase it to improve the error rate. In the fourth scenario, the system reused an answer, but this answer is wrong because it deviates from the direct inference result. We consider this suboptimal because the threshold should have been higher to reject the reuse. Consequently, we punish the threshold and increase it.

Algorithm 1: Abstract Dynamic Threshold Algorithm

Data : Direct Inference Answer *d_a*, VectorQ Answer *v_a*, Threshold t, Cosine Similarity *cs* Between Current Embedding and Nearest Neighbor, Epoch Error Rate *e_e*, User-Defined Maximum Error Rate *m_e*

Result: Threshold increase or decrease

```
1  delta ← m_e − e_e
2  if delta < 0 then
3      t' ← 1.0
4      return t'
5  end
6  if d_a == v_a then
7      if cs < t then
8          t' ← t − factor_1          // Case 1) direct inference, punish
9      else
10         t' ← t − factor_2          // Case 2) vectorQ, reward
11     end
12 else
13     if cs < t then
14         t' ← t + factor_3          // Case 3) direct inference, reward
15     else
16         t' ← t + factor_4          // Case 4) vectorQ, punish
17     end
18 end
19 return t'
```

Now we know when to adjust the threshold, but how do we determine the factor by which it should be increased or decreased? The next sections introduce, combine, and

scale the three parameters **Error Rate**, **Reuse Rate**, and **Cluster Rate** to determine the case-specific factor.

5.2.3 Error Rate and Reuse Rate

We perform occasional checks on a five-step basis to monitor the current performance by evaluating whether a reused answer aligns with the ground truth. We use an LLM response as the ground truth. Consequently, we can calculate the error rate based on correctly and incorrectly reused answers during an epoch.

$$EpochErrorRate = e_e = \frac{\#epoch_wrong_reused_answers}{\#epoch_all_answers}$$

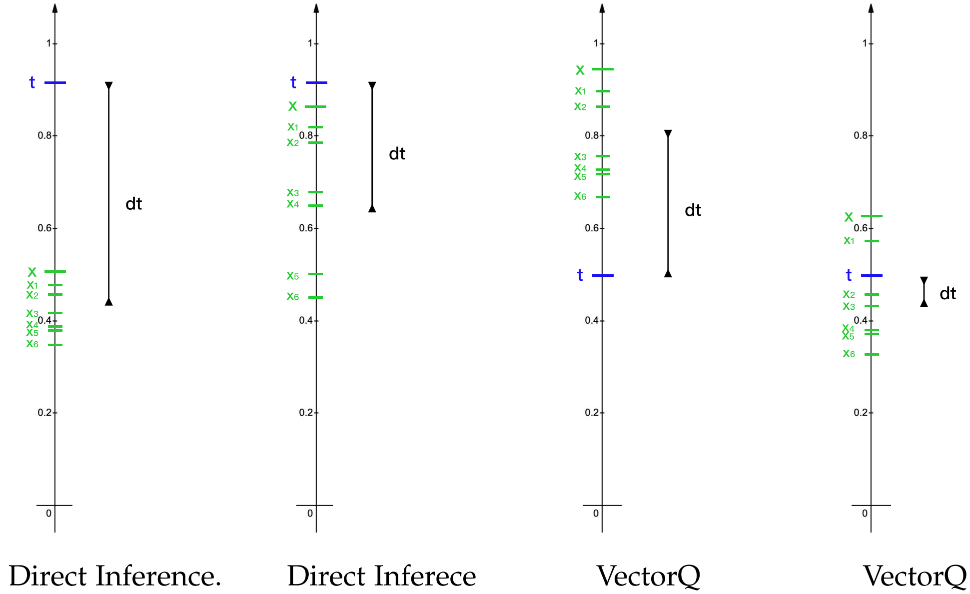
Since the number of embeddings in the vector database corresponds to the number of non-reused answers, we can infer the reuse rate as follows.

$$EpochReuseRate = e_r = \frac{\#epoch_reused_answers}{\#epoch_all_answers}$$

The three counter variables get updated at every occasional check and reset when a new epoch starts. We leverage the error rate and reuse rate to construct the threshold factors in section 5.2.6.

5.2.4 Cluster Rate

The vector database enables k-nearest neighbor retrieval of the closest vector embeddings based on cosine similarity. The k-nearest neighbor cosine similarity values are structured as $[x, x_1, x_2, \ldots, x_{k-1}]$, where x represents the cosine similarity between the embedding of the current prompt and the closest embedding in the vector database. The parameter dt encapsulates the knn cluster knowledge by determining the distance between the cosine similarity threshold t and the cluster center of K-nearest neighbors.



Direct Inference. Direct Inferece VectorQ VectorQ

We compute the cluster rate dt as the absolute distance between the threshold t and the cluster center, defined as the median of the k-nearest neighbors.

$$dt = |t - median(x_1, x_2, \ldots, x_k)|$$

We leverage the cluster rates information to construct the threshold factors in section 5.2.6.

5.2.5 Weighted Scaling

Next, we apply weighted scaling to the parameters $error_rate$, e_r, and dt based on the user-defined error rate bound to ensure that each parameter's contribution accurately reflects its real significance. Rather than using a uniform impact across all values, weighted scaling allows us to emphasize critical values while minimizing the influence of lower, less meaningful values.

Error Rate

We represent the distance between the user-defined maximum error rate m_e and the actual error rate e_e as $delta$. We use the non-inverted scaling function $e(delta, m_e)$ in threshold-decrease cases to signal that a small $delta$ should not further contribute to a

threshold decrease. A small delta implies that we almost exceed the error rate upper bound and therefore do not want to decrease it further. A large delta implies room for errors because the distance to the upper bound is large. We decrease the threshold more. The reversed logic applies for the threshold-increase case where we scale with $e^{-1}(delta, m_e)$. The following weighted sigmoid functions implement this behavior where m_e is the upper bound for the error rate.

$$e(delta, m_e) = m_e \cdot \frac{2.0}{1.0 + e^{8 \cdot (-\frac{delta}{m_e} - 0.5)}}$$

We need the inverse for the case-specific factor adjustment in section 5.2.6.

$$e^{-1}(delta, m_e) = m_e \cdot \frac{2.0}{1.0 + e^{8 \cdot (\frac{delta}{m_e} + 0.5)}}$$

The functions adjust their y-axis height based on m_e because small upper bounds require small consecutive threshold changes. We use a skewing factor of eight to capture the full spectrum of the x-axis within the range of 0 to 0.5. We have 2.0 in the nominator and shift by 0.5 because it is the maximum m_e value. The graph below visualizes both functions.

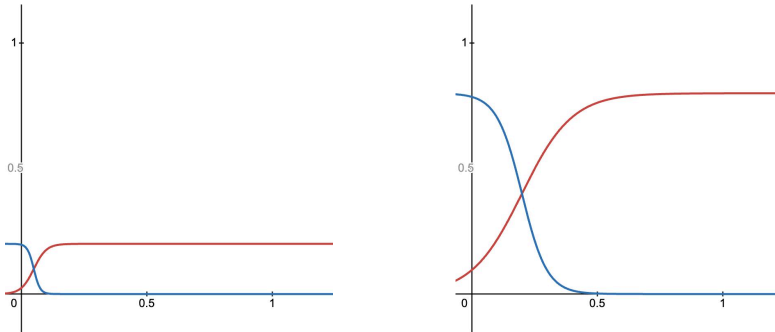


Figure 5.5: Red line = $e(delta, m_e)$, Blue line = $e^{-1}(delta, m_e)$, Left: $m_e = 0.1$. Right: $m_e = 0.4$, x-axis = delta, y-axis = scaled delta with m_e bias

We apply the scaling functions $e(delta, m_e)$ and $e^{-1}(delta, m_e)$ in section 5.2.6.

Reuse Rate

In a threshold-increase case, we want to increase less if the reuse rate is low because it would further decrease the threshold. The function $r(e_r, m_e)$ represents this behavior. In a threshold-decrease case, we want to decrease more if the reuse rate is low to increase the reuse. The function $r^{-1}(e_r, m_e)$ implements this behavior. The following weighted sigmoid functions implement this behavior where e_r is the reuse rate and m_e is the upper bound for the error rate.

$$r(e_r, m_e) = \frac{1.0}{1.0 + e^{8 \cdot (-e_r + 2 \cdot m_e)}}$$

We need the inverse for the case-specific factor adjustment in section 5.2.6.

$$r^{-1}(e_r, m_e) = \frac{1.0}{1.0 + e^{8 \cdot (e_r - 2 \cdot m_e)}}$$

The functions adjust their x-axis position based on m_e, as small error rates prioritize reuse less than large ones. Consequently, we want to scale more reuse rates to large values when m_e is large and vice versa. We use a skewing factor of eight to capture the full spectrum of the x-axis within the range of 0 to 1. We multiply m_e by two to prioritize reuse more for larger values. The graph below visualizes the weighted sigmoid function.

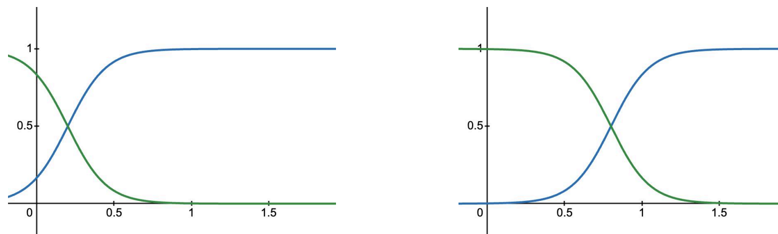


Figure 5.6: Blue line = $r(e_r, m_e)$, Green line = $r^{-1}(e_r, m_e)$. Left: $m_e = 0.1$, Right: $m_e = 0.4$, x-axis = reuse rate, y-axis = scaled reuse rate with m_e bias

We apply the scaling functions $r(e_r, m_e)$ and $r^{-1}(e_r, m_e)$ in section 5.2.6.

Cluster Rate

For dt, a small value implies a dense or closer cluster of neighbors and its contribution should remain low. As dt grows, reflecting more scattered neighbors, the contribution

should increase, to encourage threshold adjustments. We use the following weighted sigmoid functions $d(dt)$ and $d^{-1}(dt)$ to put significance on boundary values as they represent dense and close clusters.

$$d(dt) = \frac{1.0}{1 + e^{8 \cdot (-dt + 0.5)}}$$

We need the inverse for the case-specific factor adjustment in section 5.2.6.

$$d(dt) = \frac{1.0}{1 + e^{8 \cdot (-dt + 0.5)}}$$

We use a skewing factor of eight and shift by 0.5 to capture the full spectrum and center on the x-axis within the dt range of 0 to 1.The graph below visualizes the weighted sigmoid function.

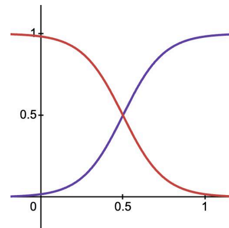


Figure 5.7: Blue line = $d(dt)$, Red line = $d^{-1}(dt)$. X-axis = dt, y-axis = scaled dt

We apply the scaling functions in section 5.2.6.

5.2.6 Threshold Factors

Now that we defined and scaled the three parameters, we are ready to construct the case-specific factors to modify the threshold.

Case 1

Case 1 in Algorithm 2 implies a direct inference that could have been replaced with a VectorQ answer reuse and we decrease the threshold.

delta	e_r	dt
$e(delta, m_e)$	$r^{-1}(e_r)$	$d(dt)$

A small **delta** should decrease the threshold less as we almost exceed the error rate bound and a decrease increases the likelihood of errors. If the delta is large, we have room for errors and want to decrease the threshold by a larger value to encourage more reuse.

For **e_r**, a high value suggests frequent answer reuse. Therefore, we can afford a threshold decrease to lower the error rate. However, if the reuse rate is low, the threshold should be decreased to encourage more reuse.

For **dt**, a small value implies that the cluster center is close to the current threshold and that the threshold was close to accepting it. Consequently, we want to decrease the threshold only slights. A large **dt** implies that the cluster center is far away and our threshold was not even close to accepting the reuse. We use the large **dt** value to decrease the threshold.

$$factor_1 = e(d, m_e) \cdot \left(\frac{r^{-1}(e_r, m_e) + d(dt)}{\sqrt{e_c}} \right)$$

We multiply the scaled delta value with the sum of the scaled error rate and cluster rate. The sum of error and cluster rate provides an initial factor size and gets scaled by the delta multiplication. The delta respects the upper bound for the error rate and scales the initial factor down in case of a low upper bound to decrease less. We divide by the root of the epoch counter e_c to make smaller threshold decreases with an increasing epoch length.

Case 2

Case 2 in Algorithm 2 implies that VectorQ correctly reused the answer and we decrease the threshold.

delta	**e_r**	**dt**
$e(delta)$	$r^{-1}(e_r)$	$d^{-1}(dt)$

A small **delta** should decrease the threshold less as we almost exceed the error rate bound and a decrease increases the likelihood of errors. If the delta is large, we have room for errors and want to decrease the threshold by a larger value to encourage more reuse.

When the **e_r** is high, there is no need to lower the threshold to maintain accuracy. If the reuse rate is low, we try a lower threshold to encourage more reuse.

A small **dt** value implies that the cluster center is close to the threshold and we almost rejected the reuse. Consequently, we invert the small **dt** value to decrease more. A large **dt** implies a large distance between the cluster center and threshold. We do not want to decrease further because this might increase the error rate.

$$factor_2 = e(d, m_e) \cdot \left(\frac{r^{-1}(e_r, m_e) + d^{-1}(dt)}{log(e_c) \cdot \sqrt{e_c}} \right)$$

We multiply the scaled delta value with the sum of the scaled error rate and cluster rate. The sum of error and cluster rate provides an initial factor size and gets scaled by the delta multiplication. The delta respects the upper bound for the error rate and scales the initial factor down in case of a low upper bound to decrease less. We divide by the product of the epoch counter's root and logarithm to make smaller threshold decreases with an increasing epoch length and reward less.

Case 3

Case 3 in Algorithm 2 implies a direct inference that correctly rejected the VectorQ answer reuse and we increase the threshold.

delta	**e_r**	**dt**
$e^{-1}(delta)$	$r(e_r)$	$d^{-1}(dt)$

We take the inverse of the **delta** because a large one should increase the threshold less as we have room for errors. If the delta is small we encourage threshold increments because we are about to exceed the error rates upper bound.

When the **e_r** is high, we can further increase the threshold for potentially better accuracy. If the reuse rate is low, we want to avoid large increases that could further reduce reuse efficiency.

A small **dt** implies that the cluster center is close to the threshold and we almost reused the answer. Consequently, we invert the small **dt** value to increase the threshold more. A large **dt** implies that the cluster center is far away and that our threshold value could be too high. Consequently, we do not want to increase the **dt** much further.

$$factor_3 = e^{-1}(d, m_e) \cdot \left(\frac{r(e_r, m_e) + d^{-1}(dt)}{log(e_c) \cdot \sqrt{e_c}} \right)$$

We multiply the scaled delta value with the sum of the scaled error rate and cluster rate. The sum of error and cluster rate provides an initial factor size and gets scaled

by the delta multiplication. The delta respects the upper bound for the error rate and scales the initial factor down in case of a low upper bound to decrease less. We divide by the product of the epoch counter's root and logarithm to make smaller threshold decreases with an increasing epoch length and reward less.

Case 4

Case 4 in Algorithm 2 implies that VectorQ wrongly reused the answer and we increase the threshold.

delta	**e_r**	**dt**
$e^{-1}(delta)$	$r(e_r)$	$d(dt)$

We take the inverse of the **delta** because a large one should increase the threshold less as we have room for errors. If the delta is small we encourage threshold increments because we are about to exceed the error rates upper bound.

When the **e_r** is high, we can further increase the threshold for potentially better accuracy. If the reuse rate is low, we want to avoid large increases that could further reduce reuse efficiency.

A large **dt** implies that the threshold is far away from the cluster center and we were not even close to rejecting the answer. Consequently, we use the large **dt** to increase the threshold. A small **dt** implies that we were close to rejecting the reuse. Consequently, we increase the threshold slightly.

$$factor_4 = e^{-1}(d, m_e) \cdot \left(\frac{r(e_r, m_e) + d(dt)}{\sqrt{e_c}} \right)$$

We multiply the scaled delta value with the sum of the scaled error rate and cluster rate. The sum of error and cluster rate provides an initial factor size and gets scaled by the delta multiplication. The delta respects the upper bound for the error rate and scales the initial factor down in case of a low upper bound to decrease less. We divide by the root of the epoch counter e_c to make smaller threshold decreases with an increasing epoch length.

Finally, we identified four factors that dynamically adjust the threshold according to the specific characteristics of the existing data points.

5.2.7 Algorithm

Each $factor_x$ combines three parameters to adjust the current threshold value. To ensure the threshold converges to the upper bound error rate and minimizes the impact of outliers, we divide $factor_x$ by the square root of epoch elements. Cases 2 and 3 are reward cases and should change the threshold less. To minimize the impact of $factor_2$ and $factor_3$, we divide them by the logarithm and root product of epoch elements. We set a lower and upper bound for the threshold to avoid divergence that goes out of bounds for a normalized cosine similarity between 0.0 and 1.0.

Algorithm 2: Complete Dynamic Threshold Algorithm

Data : Direct Inference Answer d_a, VectorQ Answer v_a, Threshold t, Cosine Similarity cs Between Current Embedding and Nearest Neighbor, Epoch Error Rate e_e, Epoch Reuse Rate e_r, Epoch Cluster Rate dt, Epoch Count e_c, User-Defined Maximum Error Rate m_e

Result: Threshold increase or decrease

```
1  d ← m_e − e_e
2  if d < 0 then
3      t′ ← 1.0
4      return t′
5  end
6  if d_a == v_a then
7      if cs < t then
8          t′ ← t − e(d, m_e) · ( (r^{-1}(e_r, m_e) + d(dt)) / sqrt(e_c) )
9      else
10         t′ ← t − e(d, m_e) · ( (r^{-1}(e_r, m_e) + d^{-1}(dt)) / (log(e_c) · sqrt(e_c)) )
11     end
12 else
13     if cs < t then
14         t′ ← t + e^{-1}(d, m_e) · ( (r(e_r, m_e) + d^{-1}(dt)) / (log(e_c) · sqrt(e_c)) )
15     else
16         t′ ← t + e^{-1}(d, m_e) · ( (r(e_r, m_e) + d(dt)) / sqrt(e_c) )
17     end
18 end
19 return min(1.0, max(0.0, t′))
```

The dynamic threshold algorithm uses performance-based adjustments and epoch-specific error rates to adapt to changes in prompt complexity, maintains reuse efficiency, and aims for user-defined accuracy. The threshold does not guarantee the optimal reuse rate but achieves a reliable error rate performance. In the next subsection, we frame this approach as an Active Online Learning Problem to contextualize its design within established paradigms.

5.2.8 An Active Online Learning Nearest Neighbor Classifier

We classify our dynamic threshold heuristic as an Active Online Learning Nearest Neighbor Classifier to contextualize its design within established paradigms. First, we define Active Learning, Online Learning, and Nearest Neighbor Classifiers.

Active Learning

According to Lindenbaum, Markovitch, and Rusakov (2004), Active Learning systems do not have access to all labeled data at once; instead, they select a subset of incoming samples to query for labels and aim to maximize accuracy with minimal labeled data. The proposed dynamic threshold heuristic, reuses or generates new answers that mirror the active selection of useful samples, similar to choosing the most confident and valuable labels. VectorQ's choice to reuse an answer based on the current threshold resembles how an active learner prioritizes confident or informative data points, guided by cosine similarity between embeddings. Furthermore, active learning systems balance new data exploration (direct inference) with the exploitation of known information (answer reuse). VectorQ's dynamic heuristic manages this balance by actively deciding when to reuse an existing answer and when to perform direct inference. Lindenbaum, Markovitch, and Rusakov (2004) refers to a strong and expensive teacher that labels the incoming data, whereas VectorQ utilizes the LLM's ground truth response as its teacher.

Online Learning

In online learning, data arrives sequentially in a streaming fashion, which requires the model to continuously adapt to the new data points (Hoi, Sahoo, Lu, & Zhao, 2021). Each decision to reuse or generate a new answer within the dynamic threshold heuristic resembles an active selection of relevant samples. The threshold is updated in real-time based on feedback on whether a reused answer was correct or incorrect. This allows the model to adjust its behavior for future decisions by incrementally decreasing or

increasing the threshold that the model's continual learning from each incoming data point.

Nearest Neighbor Classifier

The K-nearest neighbor classifier (KNN) is a multi-class classifier that assigns labels based on the majority label of the k nearest points in a set (Jain & Kapoor, 2009). The threshold heuristic leverages KNN to adjust the threshold by comparing the cosine similarity between the query embedding and its nearest neighbors against the current threshold. The cluster center (KNN median) guides the degree of adjustment by discouraging large shifts in dense regions near the threshold, and vice versa.

5.3 Re-Clustering and Cache Eviction

Since VectorQ aims to process a high volume of queries while maintaining reasonable accuracy, latency, and storage efficiency the vector database size can become a bottleneck. A naive approach would populate embeddings until a capacity limit is reached and evict based on standard cache policies like LRU, MRU, or LFU (Bang, 2023). However, this method neither accounts for the accuracy or quality of stored embeddings nor combines similar embeddings to reduce storage overhead. To address this, we propose a performance-based re-clustering and eviction strategy to optimize the embeddings in the vector database. Those strategies apply at every direct inference and the occasional check. The following four cases represent scenarios where we add, update, or remove embeddings from the vector database.

In the first case, the system performed a direct inference because the cosine similarity between the current embedding and its nearest neighbor is below the threshold. No close enough cluster exists and we create a new cluster. In the second case, the system correctly reused the answer from VectorQ. Instead of creating a new cluster, which could lead to cluster fragmentation, we merge the embeddings to combine the knowledge.

In the third case, the system made a direct inference because the cosine similarity between the current embedding and its nearest neighbor is below the threshold. No close enough cluster exists and we create a new cluster. In the fourth case, the system reused an answer, but this answer is wrong because it deviates from the direct inference result. We consider this suboptimal and remove the cluster.

Algorithm 3: Re-Cluster and Cache Eviction

Data : Direct Inference Answer d_a, VectorQ Answer v_a, Threshold t, Cosine Similarity cs Between Current Embedding e_c and Nearest Neighbor e_n, Vector Database V

Result: Updated Vector Database V

```
1  if d_a == v_a then
2      if cs < t then
3          V ← V ∪ {e_c}          // Case 1:  Create Cluster
4      else
5          c ← (e_c + e_n) / 2
6          V ← V \ {e_n}
7          V ← V ∪ {c}            // Case 2:  Merge Cluster
8      end
9  else
10     if cs < t then
11         V ← V ∪ {e_c}          // Case 3:  Create Cluster
12     else
13         V ← V \ {e_n}          // Case 4:  Evict Cluster
14     end
15 end
16 return V
```

The following three sections explain why and how we implement the create, update, and evict operations.

5.3.1 Create Cluster

In Cases 1 and 3 of Algorithm 3, direct inference generated a new answer which implies that even the nearest embedding's answer was not sufficiently similar for reuse. This indicates that the nearest cluster does not represent the intended meaning, and requires us to add the embedding to form a new cluster.

5.3.2 Update Existing Cluster

In Case 2 of Algorithm 3, VectorQ correctly reused the answer because the nearest embedding sufficiently captures the meaning required by the current embedding candidate. We merge the embeddings to capture their combined meaning. The cluster

update is decomposed into three phases.

Phase 1) We treat all embeddings in the vector space as potential reuse candidates for the incoming embedding, with each embedding mapped to its corresponding answer.

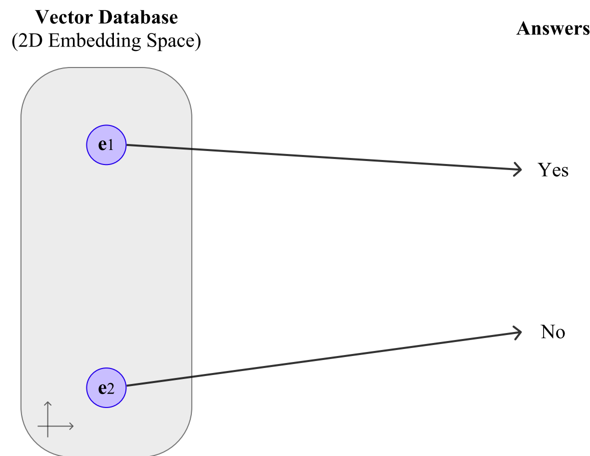


Figure 5.8: Phase 1) Simplified 2D vector space with two embeddings e_1, e_2 that form two distinct clusters.

Phase 2) When a new embedding candidate arrives, we check if an existing embedding is sufficiently close for the candidate to reuse its answer. If so, instead of keeping both embeddings in the vector space, we compute their centroid by averaging the two embeddings to capture the combined meaning. If no sufficiently close embedding exists, we add the new candidate to the vector database.

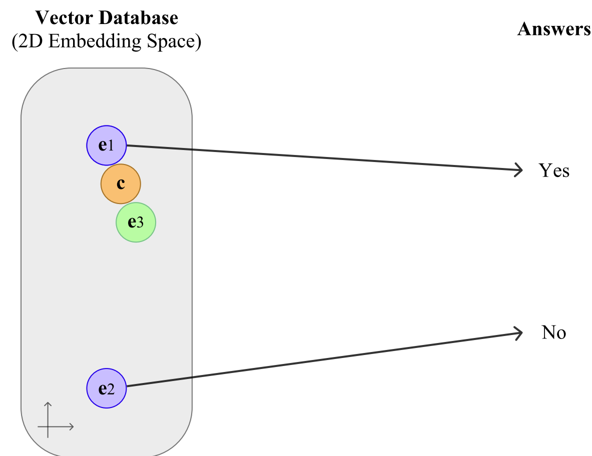


Figure 5.9: Phase 2) New embedding e_3 arrives and conforms to e_1's cluster. Compute centroid c.

Phase 3) We remove the nearest embedding and the reuse candidate from the vector space, insert the centroid into the vector database, and map it to the answer associated with the nearest embedding.

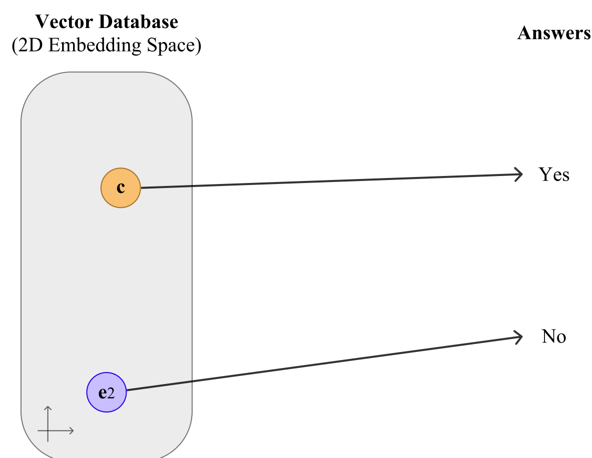


Figure 5.10: Phase 3) Remove embeddings e_1 and e_2, only keep centroid c, and map it to e_1's answer.

This ensures that we do not overpopulate the vector database with similar embeddings.

5.3.3 Evict Cluster

In Case 4 of Algorithm 3, a low threshold or an inaccurately formed cluster caused VectorQ to wrongly reuse an answer. We can create a scenario where re-clustering causes the embeddings to gradually shift toward a neighboring cluster. Eventually, the propagated embedding gets so close to this neighboring cluster that it incorrectly maps to the neighboring cluster's answer instead of its intended answer.

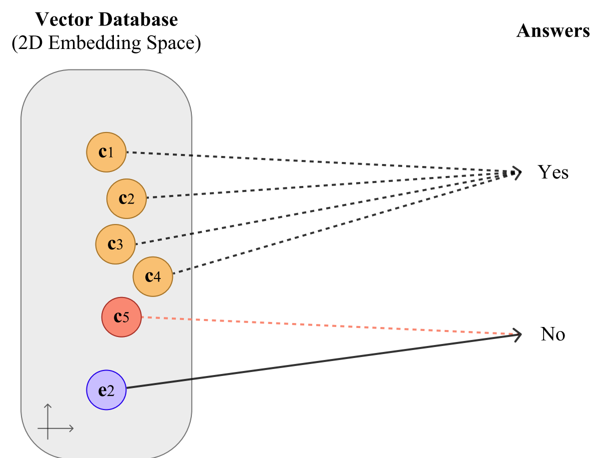


Figure 5.11: Through multiple cluster updates, the centroid moves from its initial position $c1$ to $c5$, where $c5$ now aligns with the cluster of $e2$ and incorrectly changes its mapping from 'Yes' to 'No'.

In such a scenario, we remove the propagated embedding to eliminate the incorrect mapping.

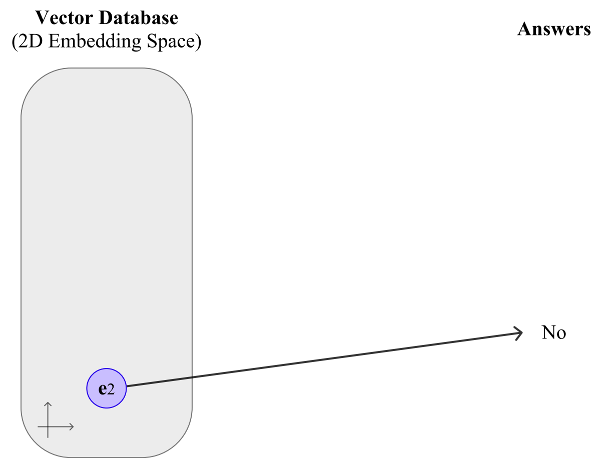


Figure 5.12: When we detect incorrect reuse, as in the case of $c5$, we remove this embedding to eliminate the inaccurate mapping.

To conclude, VectorQ enhances state-of-the-art semantic prompt caches with a dynamic cosine similarity threshold and optimized vector database clusters through re-clustering and a performance-based cache eviction policy.

6 Implementation

VectorQ is an end-to-end semantic prompt caching solution implemented as a standalone, dockerized server featuring dynamic thresholding and performance-based clustering. The system is developed in 1.6k lines of Python code, supports the HNSWLIB vector database (Malkov & Yashunin, 2018), integrates six different embedding models, and can extend to any API-accessible inference server with minimal effort—requiring only 15 lines of Python code. The current version includes support for OpenAI and Ollama inference servers. VectorQ leverages Flask (Armin Ronacher, 2024) to facilitate communication between the client, vector database, and inference server.

6.1 Multiple User Resource Management

VectorQ is designed to support multi-user environments and concurrent requests efficiently. To ensure resource management and prevent data conflicts, the system employs a session-based architecture comprising client and question sessions. Each client initiates a dedicated client session, which encapsulates an isolated vector database to prevent data leakage across users. For every new task, a question session is established to monitor task-specific performance metrics, including epoch-specific error rates, reuse rates, and the dynamically adjusted threshold. To optimize resource usage, an observer thread identifies and removes inactive client and question sessions. All users share a unified model cache to optimize the utilization of the shared GPU architecture. The model cache ensures that a model is not redundantly loaded onto the GPU when multiple clients use the same model. An observer thread tracks the number of active clients using each loaded model and unloads it from the GPU when no client utilizes it. This approach ensures efficient use of the limited GPU memory resource.

6.2 UML Architecture

The following UML diagram illustrates the key components of VectorQ introduced in the preceding section.

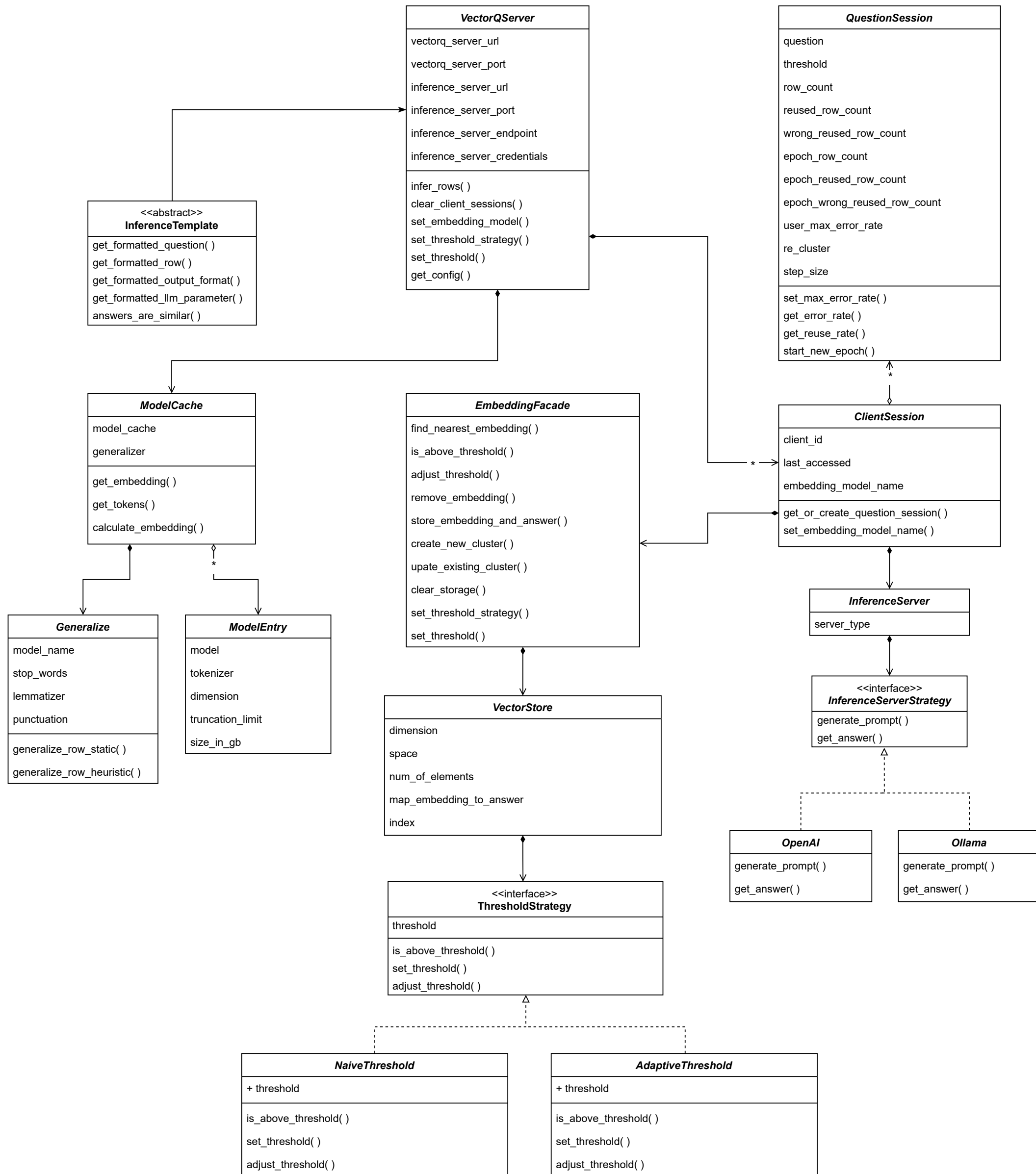


Figure 6.1: VectorQ UML Diagram.

The current implementation stores cached answers associated with vector embeddings in an in-memory map. This design provides fast retrieval but introduces limitations in data persistence. Specifically, if the VectorQ server experiences an outage, all cached data is lost, impacting reliability for long-running or mission-critical tasks. To address

this, future iterations of VectorQ will integrate a distributed storage system to enable persistent and fault-tolerant caching to mitigate the risk of data loss during server downtime. The planned distributed architecture will improve scalability by accommodating larger datasets and supporting more simultaneous user sessions. VectorQ will be made open source following its publication in spring 2025.

7 Evaluation

In this section, we evaluate the performance of VectorQ under a variety of workloads. We compare VectorQ against direct inference and the state-of-the-art semantic prompt cache GPT Cache.

Server and Model Configuration. For all of our experiments, we use an N1 GCP instance with one f1-micro CPU and one NVIDIA T4 GPU to host the VectorQ server and inference server. The T4 has 16 GB GPU memory and processes 65 TFLOPS with mixed precision (FP16/FP32). We use the lightweight LLaMA-3.1-8B model (Touvron, Lavril, Izacard, et al., 2023), hosted on our GCP instance with an Ollama (Morgan & Chiang, 2023) inference server. For all experiments, we generate vector embeddings using the lightweight model gte-large-en-v1.5 (X. Zhang, Zhang, Long, et al., 2024) and store them in the HNSWLIB vector database (Malkov & Yashunin, 2018).

Datasets. Semantic prompt caching requires the dataset to consist of clusters where the rows in each cluster map to the same answer. If all rows have distinct answers, it is impossible to reuse them. We identified three task categories that satisfy this requirement.

- **Classification** Classification tasks often involve mapping a variable amount of input data to a finite set of predefined categories. For this experiment, we use the E-Commerce Text Classification dataset (Saurabh Shahane, 2023), which assigns product descriptions to one of four categories: Books, Electronics, Household, and Clothing & Accessories. To ensure a balanced representation, we shuffle the dataset to distribute all categories evenly. The prompt for classification is structured as follows: "Which category does the text belong to? Answer with 'Books', 'Electronics', 'Household', or 'Clothing & Accessories' only." The specific product description is then appended to the end of the prompt. Additionally, we use the CommonsenseQA (Talmor, Herzig, Lourie, & Berant, 2018) dataset, which assigns questions to question categories. To ensure a balanced representation, we shuffle the dataset to distribute all categories evenly. The prompt is structured as follows: "What is the main subject of the following question? Answer with only one of the words of this set: ["people", "small dog", "cat", "car", "children", "weasel", "water", "doing homework", "human", "shark", "chatting with friends",

"student", "bald eagle", "fox", "food", "snake", "ficus", "potato", "driving car", "monkey", "animals", "apple tree", "horse", "crab", "lizard", "person", "getting drunk", "competing", "killing"]".

- **User Prompts** To simulate chatbot behavior, we use the ComQA dataset (Rogers, Gardner, & Augenstein, 2023) and modify it after the evaluation methodology employed by GPT Cache. The dataset includes a set of semantically distinct user questions; for each, we generate a corresponding semantically similar question using GPT-4o-mini. The evaluation involves two phases: first, we process the distinct questions and expect no reuse due to their uniqueness; second, we process the semantically similar questions and expect all of them to be reused. This setup tests the cache's ability to effectively identify and handle semantic similarity. Each prompt consists solely of the user-provided question.
- **Sentiment** Sentiment classification maps a variable amount of input data to a finite set of possible sentiments. While NLP-based methods, such as those proposed by (Dang, Moreno-García, & De la Prieta, 2020), outperform LLM-based sentiment analysis in terms of latency, this remains a relevant scenario, as sentiment analysis is used in semantic prompt caches within Retrieval-Augmented Generation (RAG) systems (B. Zhang, Yang, Zhou, et al., 2023). For this experiment, we use the Amazon Instant Video Review dataset (Ni, Li, & McAuley, 2019) and prompt the question: "Is this review friendly?". To ensure unbiased results, we shuffle the dataset to create an even distribution of friendly and unfriendly reviews.

Key Metrics. We evaluate each dataset based on latency, error, and reuse rates. Latency measures the time taken to process a single request. The error rate is calculated exclusively for reused answers and determined by a one-to-one equality comparison with the ground truth derived from the direct inference LLM output. We avoid using LLMs or vector embeddings for equality comparisons, as employed in GPT Cache benchmarks (Bang, 2023), to ensure more precise and reliable accuracy results. The reuse rate represents the proportion of rows with reused answers and is equivalent to the cache hit rate, as every reused answer is retrieved from the vector database cache.

7.1 Baseline 1: Direct Inference

We evaluate VectorQ's performance by comparing it with direct inference to an Ollama inference server to demonstrate the possible latency improvements of semantic prompt caching and the reliability of our dynamic threshold.

7.1.1 Accuracy

This benchmark evaluates the accuracy of the dynamic threshold as it converges to the user-defined error rate. To test error rate conservative and optimistic scenarios, we simulate error rates of 3%, 6%, 9%, 12%, and 15%.

E-Commerce Dataset

The following graph visualizes the error rate accuracy with the E-Commerce dataset.

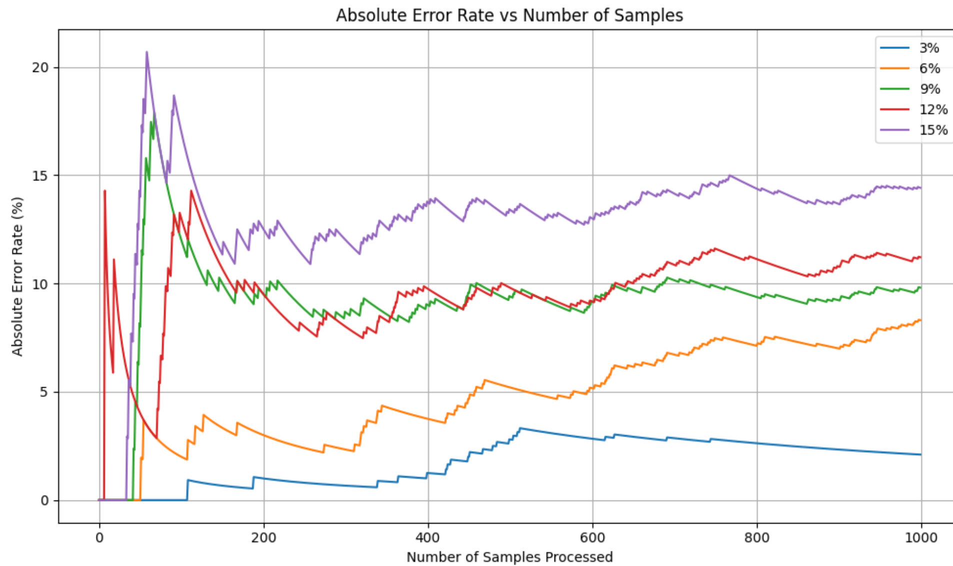


Figure 7.1: Dynamic threshold convergence performance with five different user-defined error rates for the E-Commerce dataset.

VectorQ achieves the target error rates for 3%, 9%, 12%, and 15%, but the 6% error rate shows variability due to limitations in how the dynamic threshold algorithm computes epochs. The epoch-based error rate resets the threshold when the user-defined error rate is exceeded. However, the algorithm relies on five-step intervals to perform periodic checks, determining whether reused answers are correct. In worst-case scenarios, incorrect reuses may occur in the first four steps, followed by correct reuse in the fifth step, resulting in an epoch error rate of zero percent despite 80% of the answers being incorrect. This discrepancy prevents the threshold from being adjusted appropriately, leading to further errors without recovery. We note that this issue is not unique to the dynamic threshold; a static threshold faces the same challenge and would require

continuous manual monitoring and adjustment to maintain performance and prevent such scenarios. Section 7.3 discusses this issue in greater detail and proposes solutions to address it in future work.

CommonsenseQA Dataset

The following graph visualizes the error rate accuracy with the CommonsenseQA dataset.

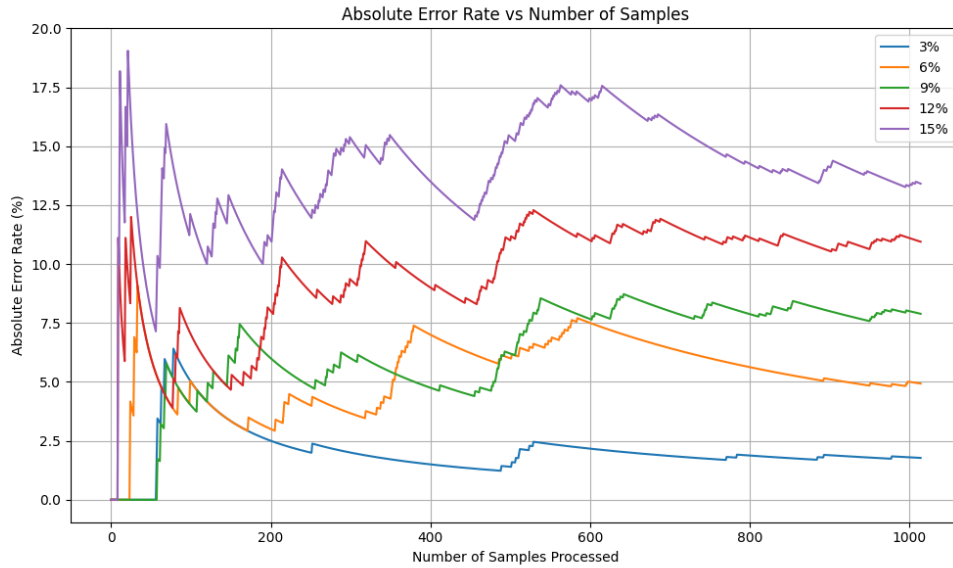


Figure 7.2: Dynamic threshold convergence performance with five different user-defined error rates for the CommonsenseQA dataset.

While the error rate targets are achieved, we observe fluctuations for the initial 150 samples. These fluctuations result from the dynamic threshold adjustment process and the pre-defined step size of five samples, which impacts the accuracy of the error rate approximation. In the early stages, when the sample size is small, the error rate approximation is less reliable, leading to higher variability and increased errors. To mitigate such fluctuations, users can configure a pre-fill phase, during which reuse is temporarily disabled, allowing the algorithm to adjust the threshold more effectively before reuse begins. The next section compares VectorQ with the state-of-the-art semantic prompt cache GPT Cache to measure the impact of a dynamic threshold.

Amazon Instant Video Dataset

The following graph visualizes the error rate accuracy with the Amazon Instant Video Review dataset.

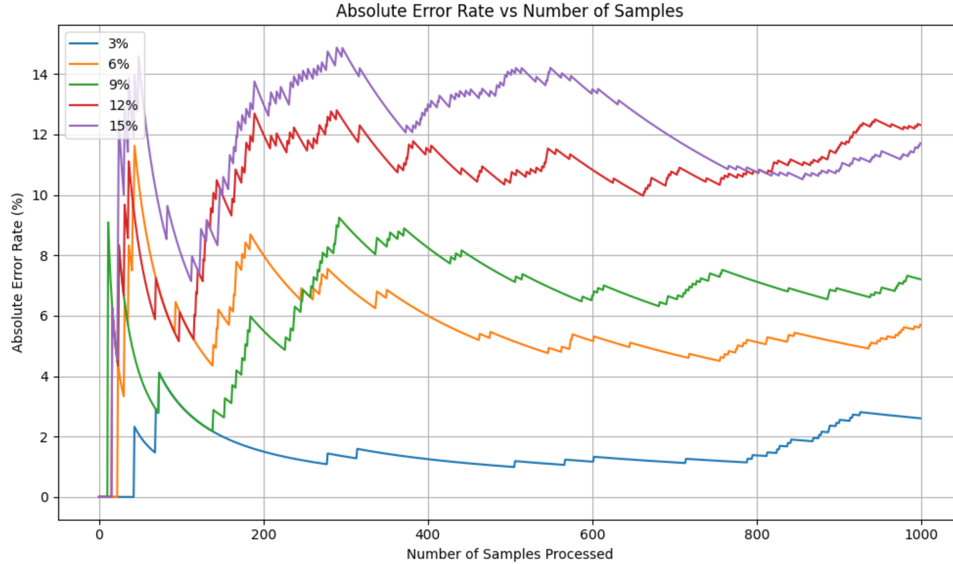


Figure 7.3: Dynamic threshold convergence performance with five different user-defined error rates for the Amazon Instant Video Review dataset.

VectorQ maintains target error rates for 3%, 6%, 9%, and 12%, but the 15% error rate shows underperformance as it could allow for higher error rates. The dynamic threshold is designed to optimize reuse rates; however, as observed in the classification dataset benchmark in section 7.1.1, the epoch error and reuse rates are only estimated in five-step intervals. If reuse occurs only in the fifth sample of a given interval, the system predicts a reuse rate of 100%, even though no reuse occurred in the previous four samples.

7.1.2 Summary

The following table outlines the performance benchmarks given the 6% and 12% user-defined error rates for all three datasets.

Dataset	User E_R	Method	Avg. Latency (sec)	Duration (min)	R_R	Real E_R
E-Commerce	6%	Direct I.	0.41	6.80	0.00	0.00
		VectorQ	0.31	5.21	48.4	6.81
	12%	Direct I.	0.41	6.80	0.00	0.00
		VectorQ	0.26	4.43	56.4	11.2
CommonsenseQA	6%	Direct I.	0.34	5.79	0.00	0.00
		VectorQ	0.32	5.51	7.50	4.93
	12%	Direct I.	0.34	5.79	0.00	0.00
		VectorQ	0.29	4.90	18.1	10.9
Amazon Review	6%	Direct I.	0.29	4.83	0.00	0.00
		VectorQ	0.25	4.20	38.5	5.71
	12%	Direct I.	0.29	4.83	0.00	0.00
		VectorQ	0.16	2.73	72.9	12.3

Table 7.1: Direct Inference and VectorQ: Average latency, duration, reuse rate (R_R), and real error rate ($RealE_R$) comparison using LLaMa 3.1-8B over three datasets, 1000 samples, and user-defined target error rates of 6% and 12%.

VectorQ achieves up to 2x latency improvement for a sample size of 1000 rows, depending on the dataset and user-defined error rate. This improvement is expected to grow with larger sample sizes, as the likelihood of semantically similar rows increases, leading to higher reuse rates. However, the relative latency of VectorQ is influenced by the output length of the LLM for a given prompt. For example, the CommonsenseQA dataset has an average LLM output length of 86 words, while the prompts for the other two datasets typically return one or two words. As a result, despite lower reuse rates for CommonsenseQA, the latency difference between direct inference and VectorQ remains comparable due to the higher inference time for longer outputs.

7.2 Baseline 2: GPT Cache

We compare VectorQ with state-of-the-art semantic prompt cache GPT Cache to evaluate the impact of a dynamic threshold over an optimal static threshold. We configure VectorQ and GPT Cache with the same embedding model and vector database as described at the beginning of chapter 7.

7.2.1 Average Case

The average case assumes a balanced dataset with semantically similar prompts, where a static threshold performs comparably to a dynamic threshold. Our benchmarks show that, despite performing a direct inference every fifth step, VectorQ remains competitive

in terms of latency. We evaluated each dataset with 200 rows for each threshold value between 0.6 and 1.0 in 0.01 increments to determine the optimal static threshold. For each error rate, we selected the lowest threshold that satisfied the target error rate. This setup ensures that both VectorQ and GPT Cache operate at comparable error rates, to enable an isolated latency comparison.

Classification Dataset

The graph illustrates the latency performance across six error rates, ranging from 2% to 15%. Each threshold value t represents the lowest possible static threshold required to achieve the corresponding error rate e_r.

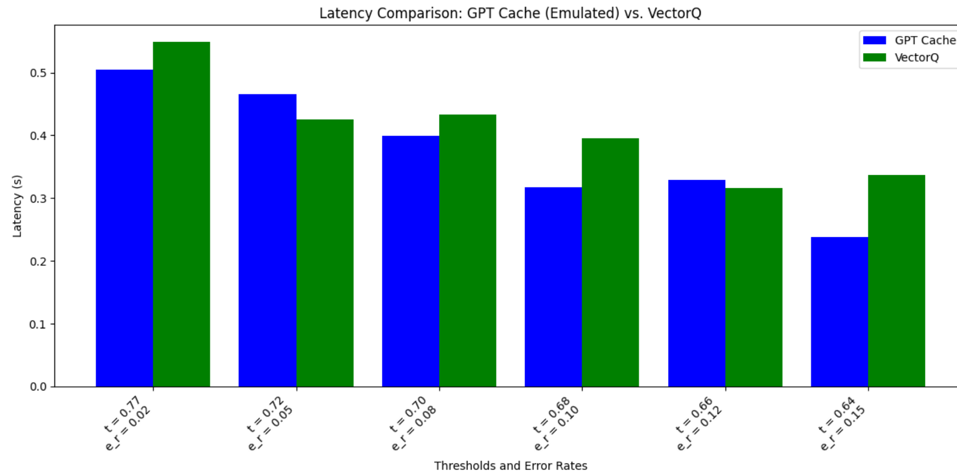


Figure 7.4: VectorQ and GPT Cache latency comparison across six threshold values and similar error rate performances with the E-Commerce dataset.

VectorQ achieves comparable latency performance while eliminating the manual effort to determine the optimal threshold for a specific set of prompts.

Sentiment Dataset

The graph illustrates the latency performance across six different error rates, ranging from 2% to 15%. Each threshold value t represents the lowest possible static threshold required to achieve the corresponding error rate e_r.

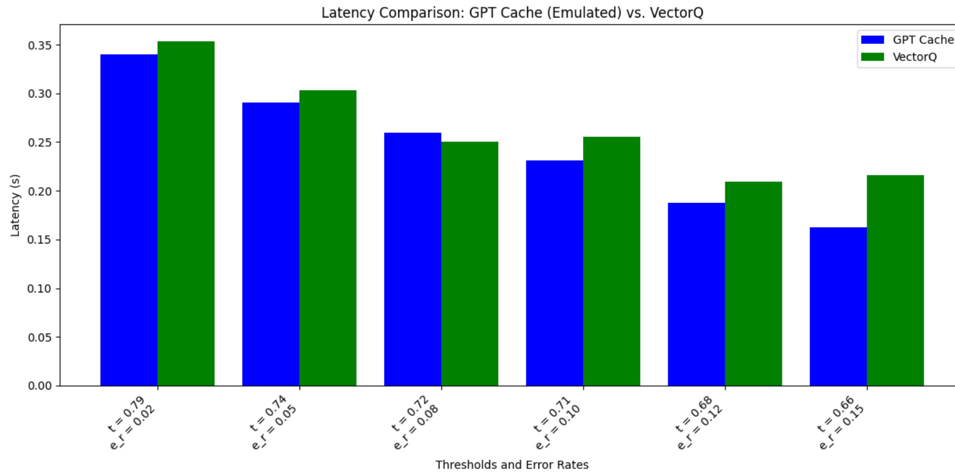


Figure 7.5: VectorQ and GPT Cache latency comparison across six threshold values and similar error rate performances with the Amazon Instant Video dataset.

VectorQ achieves comparable latency performance while eliminating the manual effort to determine the optimal threshold for a specific set of prompts.

7.2.2 Worst Case

Workloads with varying levels of complexity require different threshold values to achieve consistent performance. A significant limitation of static thresholds arises when workloads shift in difficulty, as might occur in a chatbot scenario where users ask questions of differing lengths and complexities. This benchmark simulates this behavior in four phases and uses three different datasets: the Amazon Instant Video dataset (Easy), the ComQA question dataset (Difficult), and the e-commerce classification dataset (Easy). As demonstrated in Section 7.2.2, each dataset requires a different threshold to maintain the same accuracy rate. We classify a dataset as easy if it requires a lower threshold compared to another dataset, which we classify as difficult. We use the pre-determined optimal static threshold for the first dataset to achieve a 10% error rate and process 500 samples of each of the remaining datasets.

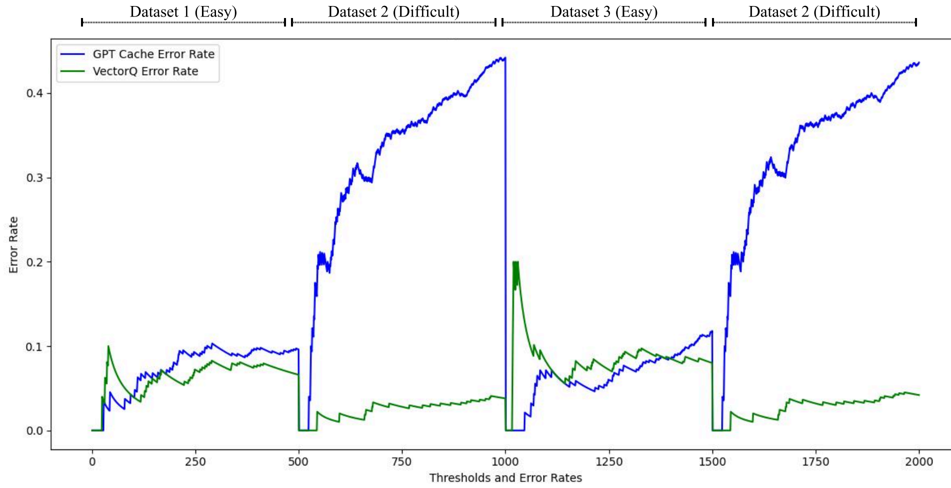


Figure 7.6: VectorQ and GPT Cache worst case error rate performance with varying workload complexities. Dataset 1: Amazon Instant Video, Dataset 2,4: ComQA, Dataset 3: E-Commerce Classification.

The results reveal that the static threshold performs well for the first dataset and third datasets which are both of similar difficulty and require a similar threshold. When the difficult dataset gets processed, the static threshold is insufficiently low and produces a 9x higher error rate compared to VectorQ. The dynamic threshold adapts to the varying complexities and maintains a relatively more stable accuracy.

7.3 Limitations

The proposed dynamic threshold offers improvements over static thresholds but remains limited by its global nature. This limitation arises from the assumption that all embedding clusters have uniform threshold requirements, which is not always the case. Some embedding clusters may be well-represented by their embeddings, requiring lower thresholds for accurate reuse. In contrast, clusters characterized by densely packed or semantically similar embeddings often require higher thresholds to avoid inaccuracies.

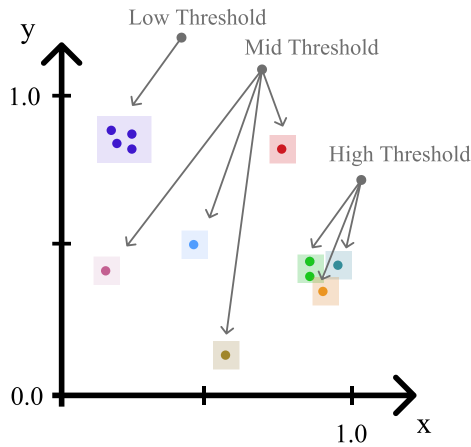


Figure 7.7: Simplified 2D vector space. Each point represents one embedding where embeddings with the same color share the same answer. Densely packed clusters like the one in the bottom right require higher thresholds to differentiate their answers.

Figure 7.7 depicts a simplified 2D vector space where each point represents an embedding, and embeddings of the same color correspond to the same answer. The clusters exhibit varying distance characteristics, necessitating different thresholds. For example, the green, blue, and orange clusters are closely spaced, requiring higher thresholds to avoid incorrect reuse, whereas the purple cluster is more distinct, allowing for a lower threshold. When embeddings from these varying clusters are processed sequentially, a single global threshold can become problematic. It may be overly restrictive for simpler clusters, limiting reuse, or insufficiently restrictive for more complex clusters, increasing errors. Although the dynamic threshold adapts over time, it fails to capture localized variations that arise within short processing intervals, leading to suboptimal performance in such scenarios.

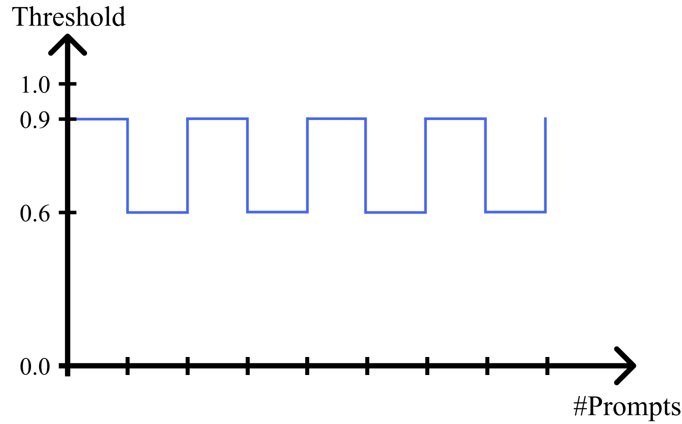


Figure 7.8: Prompts with varying complexity at a one-step interval demand dynamically adjusted thresholds.

Figure 7.8 illustrates a scenario where prompts of varying complexity are processed at one-step intervals. Complex prompts demand a high threshold of 0.9, whereas simpler prompts only require a threshold of 0.6. A single global threshold fails to adapt effectively to these rapid fluctuations. Future research will focus on implementing localized thresholds, where each cluster maintains its threshold value. This approach would leverage metadata from the nearest retrieved embedding to dynamically apply a cluster-specific threshold, enabling finer-grained and context-sensitive optimizations.

The choice of embedding model is critical, as it determines how effectively the query is represented in a compressed vector format. If the embedding model is trained on a use case that differs from the context of the prompts, it may fail to capture the relevant semantics required for accurate reuse. For instance, many embedding models are trained in a specific language and struggle to interpret prompts in other languages. To address this limitation, we propose two approaches. First, users of semantic prompt caches should leverage their domain expertise to select an embedding model that aligns with their specific application context. Second, automatic model fine-tuning, as proposed by (Zeighami, Wellmer, & Parameswaran, 2024) or (Zhu, Zhu, & Jiao, 2024), can be employed to adapt the model to the desired domain.

8 Conclusion

This thesis introduces dynamic thresholds and performance-based re-clustering as advancements to state-of-the-art semantic prompt caching systems. Unlike static thresholds, the dynamic threshold adapts based on real-time performance and eliminates the need for manual threshold selection. Performance-based re-clustering optimizes cache accuracy by combining and evicting embeddings based on their accuracy rather than their frequency of use. Our implementation, VectorQ, demonstrates a 9x improvement in accuracy compared to existing semantic prompt caches while maintaining comparable latency. However, both static and dynamic thresholds face limitations in workloads that require rapid threshold adjustments over short intervals. Future research will explore embedding location-based thresholds rather than relying on a global threshold. We will combine Bayesian Inference with Vector Space projection to create cluster-specific thresholds that better accommodate variations in prompt complexity.

List of Figures

2.1 Inference server architecture. 4

5.1 Client-server architecture without caching, where LLM requests are sent directly to the inference server. 14
5.2 Client-server architecture with semantic VectorQ caching, where LLM requests are routed through the cache before eventually reaching the inference server. 14
5.3 The VectorQ architecture enables efficient LLM inference by reusing LLM responses through semantic prompt caching. It consists of components for embedding generation, similarity evaluation, and dynamic thresholding, supported by a vector database, answer cache, and supervisor for cache optimization. 16
5.4 Epochs enable interval-specific error rate calculations and reset the threshold t to its maximum after the user-defined error rate is reached. 18
5.5 Red line = $e(delta, m_e)$, Blue line = $e^{-1}(delta, m_e)$, Left: $m_e = 0.1$. Right: $m_e = 0.4$, x-axis = delta, y-axis = scaled delta with m_e bias . . 22
5.6 Blue line = $r(e_r, m_e)$, Green line = $r^{-1}(e_r, m_e)$. Left: $m_e = 0.1$, Right: $m_e = 0.4$, x-axis = reuse rate, y-axis = scaled reuse rate with m_e bias 23
5.7 Blue line = $d(dt)$, Red line = $d^{-1}(dt)$. X-axis = dt, y-axis = scaled dt . . 24
5.8 Phase 1) Simplified 2D vector space with two embeddings e_1, e_2 that form two distinct clusters. 32
5.9 Phase 2) New embedding e_3 arrives and conforms to e_1's cluster. Compute centroid c. 33
5.10 Phase 3) Remove embeddings e_1 and e_2, only keep centroid c, and map it to e_1's answer. 33
5.11 Through multiple cluster updates, the centroid moves from its initial position $c1$ to $c5$, where $c5$ now aligns with the cluster of $e2$ and incorrectly changes its mapping from 'Yes' to 'No'. 34
5.12 When we detect incorrect reuse, as in the case of $c5$, we remove this embedding to eliminate the inaccurate mapping. 35

6.1 VectorQ UML Diagram. 37

7.1 Dynamic threshold convergence performance with five different user-defined error rates for the E-Commerce dataset. 41
7.2 Dynamic threshold convergence performance with five different user-defined error rates for the CommonsenseQA dataset. 42
7.3 Dynamic threshold convergence performance with five different user-defined error rates for the Amazon Instant Video Review dataset. 43
7.4 VectorQ and GPT Cache latency comparison across six threshold values and similar error rate performances with the E-Commerce dataset. . . . 45
7.5 VectorQ and GPT Cache latency comparison across six threshold values and similar error rate performances with the Amazon Instant Video dataset. 46
7.6 VectorQ and GPT Cache worst case error rate performance with varying workload complexities. Dataset 1: Amazon Instant Video, Dataset 2,4: ComQA, Dataset 3: E-Commerce Classification. 47
7.7 Simplified 2D vector space. Each point represents one embedding where embeddings with the same color share the same answer. Densely packed clusters like the one in the bottom right require higher thresholds to differentiate their answers. 48
7.8 Prompts with varying complexity at a one-step interval demand dynamically adjusted thresholds. 49

List of Tables

7.1 Direct Inference and VectorQ: Average latency, duration, reuse rate (R_R), and real error rate ($RealE_R$) comparison using LLaMa 3.1-8B over three datasets, 1000 samples, and user-defined target error rates of 6% and 12%. 44

Bibliography

Alsentzer, E., Murphy, J. R., Boag, W., Weng, W.-H., Jin, D., Naumann, T., & McDermott, M. (2019). Publicly available clinical bert embeddings. *arXiv preprint arXiv:1904.03323*.

Armin Ronacher. (2024). flask.

Bang, F. (2023). Gptcache: An open-source semantic cache for llm applications enabling faster answers and cost savings. *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, 212–218.

Bengio, Y., Ducharme, R., & Vincent, P. (2000). A neural probabilistic language model. *Advances in neural information processing systems*, *13*.

Brown, T. B. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Dang, N. C., Moreno-García, M. N., & De la Prieta, F. (2020). Sentiment analysis based on deep learning: A comparative study. *Electronics*, *9*(3), 483.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., & Jégou, H. (2024). The faiss library. *arXiv preprint arXiv:2401.08281*.

Franz, K., Arch, S., Hirn, D., Grust, T., Mowry, T. C., & Pavlo, A. (2024). Dear user-defined functions, inlining isn't working out so great for us. let's try batching to make our relationship work. sincerely, sql. *Conference on Innovative Data Systems Research*.

Friedman, E., Pawlowski, P., & Cieslewicz, J. (2009). Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, *2*(2), 1402–1413.

Grohe, M. (2020). Word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data. *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 1–16.

Hao, Y., Chen, Y., Zakaria, J., Hu, B., Rakthanmanon, T., & Keogh, E. (2013). Towards never-ending learning from time series streams. *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 874–882.

Hoi, S. C., Sahoo, D., Lu, J., & Zhao, P. (2021). Online learning: A comprehensive survey. *Neurocomputing*, *459*, 249–289.

Hsu, M., Chen, Q., Wu, R., Zhang, B., & Zeller, H. (2010). Generalized udf for analytics inside database engine. *Web-Age Information Management: 11th International*

Conference, WAIM 2010, Jiuzhaigou, China, July 15-17, 2010. Proceedings 11, 742–754.

Jain, P., & Kapoor, A. (2009). Active learning for large multi-class problems. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 762–769.

Jarke, M., & Koch, J. (1984). Query optimization in database systems. *ACM Computing surveys (CsUR)*, *16*(2), 111–152.

Joshi, R. A. A. (2024, August). Adding semantic caching and memory to your RAG application using MongoDB and LangChain | MongoDB.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., & Stoica, I. (2023). Efficient memory management for large language model serving with pagedattention. *Proceedings of the 29th Symposium on Operating Systems Principles*, 611–626.

Lindenbaum, M., Markovitch, S., & Rusakov, D. (2004). Selective sampling for nearest neighbor classifiers. *Machine learning*, *54*, 125–152.

Malkov, Y. A., & Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, *42*(4), 824–836.

Markjbrown. (2024, August). Semantic cache - Azure Cosmos DB.

Morgan, J., & Chiang, M. (2023). Ollama.

Ni, J., Li, J., & McAuley, J. (2019). Justifying recommendations using distantly-labeled reviews and fine-grained aspects. *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, 188–197.

Peng, F., Luo, Q., & Ni, L. M. (2017). Acts: An active learning method for time series classification. *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 175–178.

Rogers, A., Gardner, M., & Augenstein, I. (2023). Qa dataset explosion: A taxonomy of nlp resources for question answering and reading comprehension. *ACM Computing Surveys*, *55*(10), 1–45.

Romero, F., Li, Q., Yadwadkar, N. J., & Kozyrakis, C. (2021). {Infaas}: Automated model-less inference serving. *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 397–411.

Saurabh Shahane. (2023, October). Ecommerce text classification.

Shi, L., Zhang, H., Yao, Y., Li, Z., & Zhao, H. (2024). Keep the cost down: A review on methods to optimize llm's kv-cache consumption. *arXiv preprint arXiv:2407.18003*.

Sudarsan, T., & MasayaNishimaki. (2024, April). Optimize azure openai applications with semantic caching.

Talmor, A., Herzig, J., Lourie, N., & Berant, J. (2018). Commonsenseqa: A question answering challenge targeting commonsense knowledge. *arXiv preprint arXiv:1811.00937*.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Vaswani, A. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*.

Wei, Y., Langer, M., Yu, F., Lee, M., Liu, J., Shi, J., & Wang, Z. (2022). A gpu-specialized inference parameter server for large-scale deep recommendation models. *Proceedings of the 16th ACM Conference on Recommender Systems*, 408–419.

Xia, P., Zhang, L., & Li, F. (2015). Learning similarity with cosine similarity ensemble. *Information sciences*, *307*, 39–52.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., & Chun, B.-G. (2022). Orca: A distributed serving system for {transformer-based} generative models. *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 521–538.

Zeighami, S., Wellmer, Z., & Parameswaran, A. (2024). Nudge: Lightweight non-parametric fine-tuning of embeddings for retrieval. *arXiv preprint arXiv:2409.02343*.

Zhang, B., Yang, H., Zhou, T., Ali Babar, M., & Liu, X.-Y. (2023). Enhancing financial sentiment analysis via retrieval augmented large language models. *Proceedings of the fourth ACM international conference on AI in finance*, 349–356.

Zhang, X., Zhang, Y., Long, D., Xie, W., Dai, Z., Tang, J., Lin, H., Yang, B., Xie, P., Huang, F., et al. (2024). Mgte: Generalized long-context text representation and reranking models for multilingual text retrieval. *arXiv preprint arXiv:2407.19669*.

Zheng, L., Yin, L., Xie, Z., Huang, J., Sun, C., Hao Yu, C., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. (2023). Efficiently programming large language models using sglang. *arXiv e-prints*, arXiv–2312.

Zhu, H., Zhu, B., & Jiao, J. (2024). Efficient prompt caching via embedding similarity. *arXiv preprint arXiv:2402.01173*.