

WHITE PAPER · DATABASE CACHING

Semantic Caching for LLM Applications and Vector Databases

Response, retrieval, prompt and KV-cache reuse

White Paper 3 · Database Caching series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

Semantic Caching for LLM Applications and Vector Databases

White Paper 3 · Database Caching series · Bassirah Technology — Innovation & R&D · 26 October 2026

Document control

Field	Value
Series	Database Caching, paper 3 of 6
Author	Bassirah Technology — Innovation & R&D
Version	1.0 · 26 October 2026
Scope	A technology survey. It describes published research and publicly documented systems; it does not report Bassirah benchmarks or customer results.

Contents

1. Abstract	4
2. Introduction	4
2.1 Why LLM applications need a different kind of cache.....	4
2.2 How this paper builds on the series.....	4
2.3 Scope.....	5
2.4 Terms used in this paper	5
3. Where caching happens in an LLM application	5
3.1 Exact-match response caching.....	6
3.2 Semantic response caching	6
3.3 Retrieval caching in RAG	6
3.4 Provider-side prompt caching.....	6
3.5 KV-cache reuse in inference servers	7
3.6 Tool, result and plan caching for agents	7
4. How a semantic cache works.....	7
4.1 Embedding the request	7
4.2 Similarity search	8
4.3 The threshold decision	8
4.4 Verification and adaptive thresholds.....	8
4.5 Vector-capable stores as cache backends	8
5. Correctness: what a cache key must include	9
5.1 Context in the key	9
5.2 Multi-turn conversations.....	9
5.3 Personalized and time-dependent answers.....	9
5.4 The cost of a false hit	9
6. Security and privacy.....	10
6.1 Leakage across users and tenants	10

6.2 Timing side channels	10
6.3 Poisoning and key collisions	10
6.4 Retention and governance	10
7. Freshness and invalidation	10
8. KV-cache reuse in depth	11
8.1 Prefix caching	11
8.2 Reuse beyond the prefix	11
8.3 Tiered and distributed KV caches	11
9. Measuring a semantic cache	12
10. Comparative analysis	12
11. Open challenges and future opportunities	13
11.1 Correctness guarantees for approximate hits	13
11.2 Thresholds that transfer across domains	13
11.3 Privacy, isolation and side channels	13
11.4 Multi-turn and agentic context	13
11.5 Invalidation when knowledge or models change	14
11.6 Evaluating quality as well as savings	14
11.7 Composing cache layers	14
11.8 Robustness against adversarial use	14
12. What this means for platform decisions	14
References	15

1. Abstract

Applications built on large language models (LLMs) repeat work. Users ask the same questions in different words, retrieval pipelines fetch the same passages, agents re-run the same tool calls, and every request re-sends long system prompts and documents that the model has already processed. Exact-match caching captures little of this repetition, because two requests rarely match character for character. This paper examines the caching techniques built for LLM workloads. It separates two ideas that are both called caching: reusing the model's computation for an identical prompt prefix, which cannot change the answer, and returning a stored answer for a similar request, which can. It describes the layers at which caching happens, from application-level response caches to retrieval caches, provider-side prompt caching and reuse of the attention key-value (KV) cache inside inference servers. It explains how semantic caches match requests by embedding similarity in a vector index, why the similarity threshold is a correctness decision, and which context must be part of a cache key. It then covers the security and privacy risks of shared caches, freshness and invalidation, KV-cache reuse in depth, and how to measure a semantic cache on quality as well as savings. It closes with a qualitative comparison, open challenges framed as research directions and opportunities, and notes for platform decisions.

2. Introduction

2.1 Why LLM applications need a different kind of cache

A database query is cheap relative to an LLM call. Generating a response takes two phases with different resource profiles: a compute-intensive pass over the whole prompt (prefill), which governs the time to the first token, and a memory-intensive step per output token (decode) [1], [2]. The KV cache that the model builds for each request is large and changes size as the request runs, which limits how many requests an accelerator can serve at once [3]. Long prompts, which retrieval-augmented generation (RAG) and agents make common, increase the prefill cost of every request.

Caching can avoid some of this work, but the classic cache in paper 1 matches keys exactly, and natural-language requests seldom repeat exactly. Two different ideas address the gap, and the difference between them runs through this paper. **Prompt and KV caching** reuse the model's own computation for the part of a prompt that repeats exactly, such as a system prompt, tool definitions or a shared document; the model still generates every output token, so the answer is the one a fresh request would produce [3], [4]. **Semantic caching** returns a stored answer for a request whose meaning is close to an earlier one, measured by the similarity of their embeddings, and skips the model call entirely [5]. Only the second can return a wrong answer.

2.2 How this paper builds on the series

Paper 1 described caching strategies, eviction, invalidation and pitfalls for database caches. They apply here, with one change: a semantic cache can return an answer to a question that was not asked, so a false hit is a correctness failure rather than a performance problem. Paper 4 of the Data Storage Engines series described the vector indexes that semantic caches are built on; this paper refers to it rather than repeating index design. Paper 4 of this series

examines memory and storage tiers, which also hold KV caches (Section 8.3), and paper 5 examines dependency-driven invalidation (Section 7).

2.3 Scope

This paper covers caching for LLM applications: response caches (exact and semantic), retrieval caches, provider-side prompt caching, KV-cache reuse in inference servers, and caching in agents. It concentrates on peer-reviewed work and preprints from 2023 to 2026 and on official documentation, and cites earlier work for background. Statements about products reflect documentation accessed on 26 October 2026; LLM serving changes quickly, and features labeled preview or beta may change. It does not report benchmark figures, hit ratios, cost savings, prices, token limits or latency figures. Where a paper reports results, we describe them qualitatively and attribute them to the authors. Many works in this field are preprints that have not completed peer review; the references identify them.

2.4 Terms used in this paper

A **prompt** is the full input sent to a model, including system instructions, conversation history, retrieved documents and tool definitions. An **embedding** is a vector that represents the meaning of a text, produced by an embedding model [6]. **Similarity** between two embeddings is usually measured by cosine similarity, or its complement, a distance; a **threshold** decides how close two requests must be to count as a match. A **semantic hit** returns a stored response for a request that is similar, not identical, to the one that produced it; a **false hit** is a semantic hit whose stored response is wrong for the new request. The **KV cache** holds the attention keys and values the model computes for each token of a prompt. A **prefix** is the leading part of a prompt that several requests share.

3. Where caching happens in an LLM application

An LLM application has several places where work repeats, and each supports a different kind of cache. Table 1 summarizes them; the subsections give detail.

Layer	What is reused	Match type	Who operates it	Main risk
Exact response cache	The final response	Exact key (normalized prompt and parameters)	Application	Few hits; stale responses
Semantic response cache	The final response	Embedding similarity within a threshold	Application, often on a vector-capable store	False hits; leakage and poisoning across users
Retrieval cache	Retrieved passages or search results	Exact or approximate query match	Application or retrieval service	Stale passages after source updates
Provider-side prompt caching	Computation for a repeated prompt prefix	Exact prefix	Model provider	Timing exposure if scoped too widely; savings depend on prompt structure
KV-cache reuse in the inference server	Attention keys and values	Exact prefix, or reused chunks in research systems	Team running the model	Memory pressure; key completeness; complexity
Tool, result and plan cache for agents	Tool results, intermediate results or plans	Exact or approximate task match	Agent framework	Acting on outdated results

3.1 Exact-match response caching

The simplest LLM cache stores the response for a normalized prompt together with the model and generation parameters, as a database cache stores a query result. It is safe when the key captures everything that affects the answer, and useful for repeated programmatic calls, evaluation runs and popular fixed prompts. With sampling enabled, the same prompt can produce different responses, so caching one of them is a product decision: it makes the system deterministic for repeated requests.

3.2 Semantic response caching

A semantic cache embeds each incoming request, searches a vector index of earlier requests, and returns the stored response of the nearest neighbor when it is close enough; otherwise it calls the model and stores the new pair. GPTCache, an open-source library, made the pattern widely known: requests go to the cache first, and a hit skips the model call [5]. Early descriptions of the pattern store query embeddings in an in-memory store such as Redis and target repetitive questions such as customer service [7]. Research since then has studied how to make it accurate and efficient. SCALM analyzed real chat logs and found semantic patterns in queries that a cache can exploit [8]. MeanCache keeps the cache on the user's device and trains its similarity model with federated learning, so queries are not shared [9]. vCache replaces a single global threshold with decisions learned for each cached entry against an error bound the user sets [10]. Work on eviction argues that a semantic cache must weigh the cost of a mismatch between a query and a cached response, not only reuse [11]. Approximate reuse also extends beyond text: NIRVANA reuses intermediate states from earlier generations for similar prompts in text-to-image diffusion models [12]. Section 4 describes the mechanism and Section 5 its correctness questions.

3.3 Retrieval caching in RAG

RAG pipelines embed a query, search a vector index for relevant passages, and add them to the prompt. Caching can apply at several points. Proximity places an approximate cache in front of the vector database and reuses documents retrieved for earlier, similar queries, trading some retrieval quality for reuse [13]. RAGCache stores the intermediate model states of retrieved documents in a knowledge tree across accelerator and host memory, so that the prefill for frequently retrieved documents is not repeated [14]. Cache-Craft manages caches of retrieved chunks and decides which can be reused in a new context [15]. Retrieval caches inherit the freshness problem of paper 1 in a sharper form: when a source document changes, every cached result and response derived from it may be stale (Section 7).

3.4 Provider-side prompt caching

Model providers cache the processed form of repeated prompt prefixes, so that requests sharing a long system prompt, tool list or document do not pay the full prefill cost each time. In every major provider's documentation, matching is exact and by prefix: any change before the cached point, such as an edited tool definition or a timestamp in the system prompt, is a miss [16], [17], [18], [19]. Prompts should therefore place stable content first and variable content last.

Providers differ in how caching is requested. OpenAI applies it automatically, with an optional explicit mode on newer models [17]. Anthropic's Claude API caches when the request opts in, either with a top-level setting that moves the cache point forward as a conversation grows or with explicit breakpoints [16]. The Gemini API caches implicitly by default without guaranteed

savings and also offers explicit cache objects, labeled beta, with a time to live the developer can set [18], [20]. Amazon Bedrock offers implicit and explicit caching depending on the model [19]. Providers also set a minimum cacheable prefix length and a retention period that differ by model; prompts below the minimum are processed without caching [16], [17], [19]. Because only the computation for the prompt is reused, provider-side caching carries no false-hit risk.

3.5 KV-cache reuse in inference servers

Teams that run models themselves can reuse KV caches across requests inside the inference server. vLLM's PagedAttention manages KV-cache memory in fixed-size blocks, in the way an operating system manages pages, which makes it practical to share blocks between requests [3], and its automatic prefix caching reuses the blocks of a shared prefix [21]. SGLang organizes cached prefixes in a radix tree, so that the many calls of an LLM program, such as an agent or a multi-turn conversation, share computation [4]. TensorRT-LLM reuses KV blocks for shared prefixes by default [22], and NVIDIA Dynamo routes requests to the workers that already hold their prefixes, balancing cache overlap against load [23]. Section 8 covers this layer in depth.

3.6 Tool, result and plan caching for agents

Agents call tools, query databases and generate multi-step plans. Results of deterministic tool calls can be cached like any function result, subject to the freshness of the data they read, and Redis documents caching intermediate results of multi-step agent workflows as a use of its semantic cache service [24]. Agentic plan caching stores plan templates from completed runs, matches new tasks to them, and adapts the plan with a small model, because whole answers depend on live data that a response cache cannot capture [25]. In multi-agent workflows, KVFlow evicts and prefetches agents' fixed-prompt KV caches according to when each agent will run next [26]. Agent caches raise the stakes of staleness and false hits, because an agent may act on a cached result rather than only display it.

4. How a semantic cache works

4.1 Embedding the request

The cache converts the request, or a normalized part of it, into an embedding. What to embed matters: embedding only the latest user message makes unrelated conversations look alike (Section 5.2), while context that determines the answer, such as model, tenant and prompt version, is better applied as a filter than mixed into the embedded text (Section 5.1). The embedding model determines which requests look similar. General-purpose embeddings measure whether two prompts are about the same thing, which is not the same as whether one response answers both; Zhu et al. found off-the-shelf embeddings weak on hard pairs and improved them by fine-tuning for the caching objective [27]. Redis researchers report that small, domain-specific embedding models fine-tuned on targeted real and synthetic pairs can outperform larger general models for caching [28]. Leaderboards that compare embedding models across many tasks [29] are a starting point, not a substitute for evaluation on the cache's own traffic.

4.2 Similarity search

The embedding is compared with the embeddings of cached requests in a vector index. Approximate nearest neighbor (ANN) indexes such as HNSW graphs [30] are the common choice, as provided by libraries such as Faiss [31], by the pgvector extension for PostgreSQL [32], and by Redis vector search, which offers exact, HNSW and compressed graph indexes, cosine and other distance metrics, range queries by radius, and filters on tag, numeric and text fields [33]. ANN search can miss the true nearest neighbor; in a cache that causes a miss, which is safe. Filtering deserves care: with approximate indexes in pgvector, filters are applied after the index scan, so a filtered search can return fewer candidates than expected [32]. Paper 4 of the Data Storage Engines series describes these indexes and their filtering support in depth.

4.3 The threshold decision

If the nearest cached request is close enough, the cache returns its response. The threshold sets the trade-off between hits and false hits: a loose threshold returns more cached answers and more wrong ones, a strict one fewer of both. A single global threshold gives no formal correctness guarantee and either misses safe reuse or serves wrong answers, because the point at which two requests stop meaning the same thing differs by domain, phrasing and entry [10], [34]. Thresholds also come in two conventions. RedisVL's semantic cache sets a cosine distance, where lower values are stricter [35], while LangCache sets a similarity threshold, where higher values are stricter [36]. Teams moving between tools should check which convention applies, and treat the threshold as a parameter to tune and test, not a default to accept.

4.4 Verification and adaptive thresholds

Research has proposed several ways to make the hit decision more reliable. vCache learns how close a new request must be to each cached entry for its response to be correct, and aims to keep the error rate below a bound the user sets [10]. Krites leaves the serving path unchanged and asks an LLM judge, asynchronously, whether near-miss matches are safe before promoting them [34]. Category-aware caching sets thresholds and expiration per kind of query, because code, conversation and factual queries differ in how densely their embeddings cluster and how quickly they go stale [37]. LaCache, a recent preprint, checks the first tokens of the response as well as the query before accepting a hit [38]. Choosing a model for this task also needs the right metric: ranking metrics do not show how a model performs at the fixed threshold a cache uses, and the models that rank best can perform worst in operation [39]. These approaches move the threshold from a guess to a measured property, but each adds components that must themselves be evaluated.

4.5 Vector-capable stores as cache backends

A semantic cache needs storage for embeddings, responses and metadata, an ANN index, metadata filters, expiration and eviction. In-memory stores with vector search fit this profile. RedisVL provides a semantic cache on Redis vector search with a distance threshold, time to live and filterable fields, and the threshold can be overridden per call [35]. Redis LangCache offers semantic caching as a service, fully managed on Redis Cloud or self-managed, that generates embeddings itself, supports exact and semantic search, and scopes entries with custom attributes; it is in preview [24], [40]. Redis now presents LangCache as part of Redis Iris, which it describes as a context engine for AI agents [41]. PostgreSQL with pgvector can hold a

semantic cache beside the application's other data [32]. The considerations of paper 1 apply to all of them: the cache needs bounded size, an eviction policy, expiration, and invalidation when its sources change.

5. Correctness: what a cache key must include

5.1 Context in the key

In a database cache, the key is the query. In an LLM cache, the request text is only part of what determines the answer. A response depends on the model and its version, any adapter applied to it, the system prompt and prompt template, the tools available, generation parameters, the retrieved documents and the version of the corpus they came from, the conversation so far, the user's language, and often the user's identity, role and permissions. Each of these must be part of the key or applied as a filter on the search, or the cache will return an answer produced under different conditions.

Inference servers apply the same principle to KV blocks. vLLM's block hashes combine the block's tokens with the parent block, adapter identifiers, multimodal input hashes and an optional per-request salt that restricts reuse to requests sharing it [21]. TensorRT-LLM's documentation warns that reuse can be incorrect when the key does not capture everything that affects the output [22]. At the application layer, filterable fields in RedisVL and attributes in LangCache are the documented way to scope semantic matches [35], [40], and prompt versions and labels from a prompt management system, such as Langfuse's, are natural inputs to the key [42]. Above all, a response generated from documents one user may see must never be returned to a user who may not see them; filtering by tenant and permission before similarity search prevents a forbidden entry from ever being a candidate.

5.2 Multi-turn conversations

In a conversation, the same words mean different things depending on what came before: "and in Python?" is meaningless without the previous turn. Caching on the latest message alone produces false hits across conversations; caching on the whole history produces few hits. MeanCache encodes chains of context so that a follow-up is not matched to a standalone query [9]; ContextCache first retrieves candidates by vector search and then matches on the current and earlier dialogue [43]; and SmartCache reuses a response only when both the query and the conversational context match, and detects topic shifts from attention signals [44]. Multi-turn caching remains an open problem (Section 11.4).

5.3 Personalized and time-dependent answers

Answers that depend on the user, such as an account balance or an order status, on the current time, or on live data should not be cached as responses, or should be cached only per user with short expiration. Classifying requests before the cache and bypassing it for those that depend on personal or live data is a common safeguard. Responses intended to vary, such as brainstorming, gain little from caching.

5.4 The cost of a false hit

A false hit returns a fluent, confident answer to a different question. Unlike a stale database read, it can be hard for users to detect, and its cost depends on the application: minor in a casual assistant, serious in advice, compliance, finance or healthcare, and potentially harmful

when an agent acts on it. The decision to use semantic caching, and its threshold, should follow from the cost of an error in each use case, measured on representative queries (Section 9), rather than from the savings alone.

6. Security and privacy

6.1 Leakage across users and tenants

A shared semantic cache can return one user's response, which may contain that user's data, to another user who asks a similar question. Scoping caches by tenant and user, excluding personal data from shared caches, and applying permission filters before matching (Section 5.1) address the direct risk. Keeping the cache on the user's device, as MeanCache does, avoids sharing queries at all [9].

6.2 Timing side channels

Caching makes cached requests faster than uncached ones, and that difference can be observed. Gu et al. showed that if a prompt cache is shared across users, response timing can reveal other users' prompts, and their statistical audits found that some API providers shared caches globally across users at the time of the audit; providers may have changed their policies since [45]. Other researchers showed that shared KV caches and semantic caches in serving systems can be probed to recover system prompts and other users' inputs [46], [47]. Current provider documentation states that caches are not shared across organizations, and in some cases are isolated per workspace [17], [16]; vLLM's per-request salt lets self-hosted deployments isolate tenants in the same way [21]. Teams that build their own caches should scope them as narrowly and treat timing as a channel to control.

6.3 Poisoning and key collisions

If any user's request can create an entry that other users later receive, an attacker can try to plant a response. Wu et al. demonstrated such semantic cache poisoning, in which crafted entries cause other users to receive attacker-chosen responses, and found existing defenses insufficient [48]. Zhang et al. showed that the locality a semantic key needs for hits conflicts with collision resistance, so a crafted query can collide with a target query's key and hijack its response, and that the attack transfers across embedding models [49]. Mitigations include not writing unverified responses into shared tiers, separating caches by trust level, verifying hits, including on the response side [38], and monitoring entries that serve unusually many requests.

6.4 Retention and governance

Cached prompts and responses are copies of user data, subject to the same retention, deletion and residency rules as the original. A cache that keeps entries for long periods, or that cannot delete one user's entries on request, can conflict with those rules. Expiration, per-user scoping and deletion by attribute, which LangCache's API supports [40], make governance practical.

7. Freshness and invalidation

The invalidation problem of paper 1 appears at every layer of an LLM application. Fixed expiration is a blunt tool here: one time to live either wastes memory on stable answers or serves stale ones for fast-changing topics, which is why category-aware designs set it per kind

of query [37], and FreshCache, a recent preprint, estimates the probability that an entry's evidence has changed before approving a hit in RAG over web sources [50]. A response depends on the passages retrieved for it, so recording which source documents and versions each cached response used allows targeted invalidation when they change; with an attribute-scoped store, a team can then delete exactly the affected entries [40]. This is a design pattern rather than a documented product recipe, and paper 5 examines dependency tracking in general.

A change to the system prompt, template, tools or model can change every answer, so these belong in the key (Section 5.1), which retires the old entries implicitly. Prefix and KV caches are invalidated by construction, because any change to the prefix changes the match [21], [16]; provider caches simply expire, and some cannot be cleared on request [17], while explicit cache objects have a time to live the developer can update [20]. Feedback from users and evaluators provides a further signal: an entry rated poorly can be removed (Section 9).

8. KV-cache reuse in depth

8.1 Prefix caching

The KV cache of a prompt prefix depends only on the tokens of that prefix and the model, so any request that begins with the same tokens can reuse it. vLLM caches full KV blocks, identifies them by a hash chain over their tokens, and evicts them with an LRU policy [21]; TensorRT-LLM also shares full blocks, evicts with LRU and can offload to host memory [22]; and SGLang's RadixAttention keeps cached prefixes in a radix tree [4]. Prefix caching works best when applications put stable content first, the same rule as for provider-side caching (Section 3.4), and when the cluster routes requests with shared prefixes to the workers that hold them [23]. Prompt Cache proposed defining reusable prompt modules, such as system messages, templates and documents, in a schema so that their attention states can be precomputed and reused at the right positions [51].

8.2 Reuse beyond the prefix

In RAG, the same document may appear at different positions and after different text in different prompts, and its KV cache depends on the tokens before it, so prefix caching cannot reuse it. CacheBlend reuses KV caches of retrieved chunks computed separately and recomputes a small subset of tokens to recover the cross-attention between chunks [52]. Cache-Craft observes that naive reuse of a chunk's cache in a new position or context degrades output quality, and decides which chunk caches can be reused and how much to recompute [15]. These techniques trade some exactness for reuse, so their effect on quality must be evaluated, as with semantic caching.

8.3 Tiered and distributed KV caches

KV caches are large, and keeping them in accelerator memory competes with serving. Systems therefore move them to cheaper tiers. CachedAttention keeps the KV caches of conversation history in a hierarchy of memory and storage, with preloading and asynchronous saving, so that later turns can reuse them [53]. Mooncake, the serving platform for the Kimi chatbot, separates prefill and decode clusters and builds a disaggregated KV cache from spare CPU memory, DRAM and SSD in the cluster, trading storage for computation [54]. CacheGen compresses KV caches and adapts compression to network bandwidth, because

moving large tensors is part of the cost of reuse [55]. LMCache provides a KV-cache layer that offloads and shares caches across inference engine instances and tiers from accelerator memory to local disk and remote stores, including Redis and Valkey [56], [57]. Paper 4 of this series examines tiered caching across memory and flash, which applies directly to these designs.

9. Measuring a semantic cache

Hit ratio alone is a misleading measure for a semantic cache, because a loose threshold can produce a high hit ratio and many wrong answers. A useful evaluation measures, on representative traffic, the share of hits, the share of hits that are correct, the cost and latency saved, and the effect on user-facing quality, and compares models at the threshold they will run with rather than by ranking metrics [39]. Offline, teams can label pairs of requests as equivalent or not, replay them through the cache at several thresholds, and choose the threshold whose error rate the use case tolerates; the vCache authors released benchmark datasets for this purpose (SemBenchmark) [10]. Online, each request can be traced with whether it hit, the similarity score, the entry used, and quality signals such as user feedback or automated evaluation.

Langfuse, an open-source LLM engineering platform built on OpenTelemetry, records traces of LLM applications with nested observations, sessions for conversations, and user, tag and metadata attributes, which is where a cache layer, hit flag and similarity score can be recorded [58]. It tracks usage and cost per model call with arbitrary usage types, so cache-read tokens from provider prompt caching can be recorded separately from other input [59]. Its evaluation features attach scores from users, annotators and LLM-as-a-judge evaluators to production traces and run experiments on datasets, which supports auditing false hits offline and sampling cache-served answers online [60]. The OpenTelemetry semantic conventions for generative AI define common attributes for model calls, including cache-read and cache-write token counts; they have Development status [61].

10. Comparative analysis

Table 2 compares the caching approaches in this paper. The ratings are qualitative and relative to each other; actual behavior depends on the product, model, configuration and workload.

Approach	Reuse it captures	Effect on answer correctness	Implementation effort	Operational concerns	Best-fit workloads
Exact response cache	Low (identical requests only)	None if the key is complete	Low	Expiration; key completeness	Repeated programmatic calls; fixed prompts; evaluation runs
Semantic response cache with a fixed threshold	Medium to High	Risk of false hits	Low to Medium	Threshold tuning; leakage; poisoning; invalidation	Assistants with repetitive questions where occasional errors are tolerable

Approach	Reuse it captures	Effect on answer correctness	Implementation effort	Operational concerns	Best-fit workloads
Semantic cache with verification or adaptive thresholds	Medium to High	Reduced, measured error rate	Medium to High	Extra models to run and evaluate	Higher-stakes assistants that still repeat questions
Retrieval cache	Medium	Stale or less relevant passages if not managed	Medium	Source-change tracking	RAG over slowly changing corpora
Provider-side prompt caching	Medium to High for long shared prefixes	None	Low (prompt structure)	Provider-specific behavior and retention	Long system prompts, tool lists and documents
Self-hosted prefix KV caching	Medium to High for shared prefixes	None if the block key is complete	Medium (serving stack)	Memory, routing, eviction, tenant isolation	Teams running their own models
Non-prefix KV reuse for RAG chunks	High for repeated chunks	Small, to be evaluated	High (research systems)	Maturity; quality evaluation	RAG at scale, after evaluation

11. Open challenges and future opportunities

11.1 Correctness guarantees for approximate hits

Challenge. Semantic caching returns answers to questions that were not asked, and embedding similarity is an imperfect proxy for "the same answer applies" [27]. **Research direction.** Per-entry decision rules with error bounds [10], asynchronous verification of near misses [34], response-side checks [38], and embeddings trained for the caching objective [27], [28]. **Opportunity.** Caches that state and meet an error bound would let teams use semantic caching in higher-stakes applications.

11.2 Thresholds that transfer across domains

Challenge. A threshold tuned on one domain, language or embedding model does not transfer to another, and workloads drift. **Research direction.** Adaptive and category-aware policies [10], [37], operational metrics measured at the deployed threshold [39], and shared benchmarks [10]. **Opportunity.** Reusable evaluation sets and adaptive thresholds would turn tuning from a one-off exercise into a monitored control loop.

11.3 Privacy, isolation and side channels

Challenge. Shared caches can leak data across users directly or through timing [45], [46], [47]. **Research direction.** On-device and federated caches [9], per-tenant salts and scopes [21], and timing defenses for cache-aware serving. **Opportunity.** Caches with isolation designed in, rather than added later, would make caching acceptable for regulated data.

11.4 Multi-turn and agentic context

Challenge. Conversations and agent workflows carry context that a single request does not show, and agents act on what they retrieve. **Research direction.** Context-aware matching [9],

[43], [44], plan caching [25], workflow-aware KV management [26] and KV reuse across turns [53]. **Opportunity.** Caches that understand conversation and task state could serve agents safely, with explicit rules for when a cached result may be acted on.

11.5 Invalidation when knowledge or models change

Challenge. Cached answers outlive the documents, prompts and models that produced them. **Research direction.** Freshness-aware reuse decisions [50], provenance links from responses to sources, and dependency-driven invalidation, which paper 5 examines for database caches. **Opportunity.** RAG systems that invalidate exactly the responses affected by a document change would combine caching with freshness.

11.6 Evaluating quality as well as savings

Challenge. Savings are easy to measure and quality is not, so caches can be tuned for the wrong objective. **Research direction.** Evaluation that combines labeled equivalence sets, automated judges and user feedback, attached to production traces [60], and common telemetry for cache behavior [61]. **Opportunity.** Treating cache quality as a monitored metric beside cost and latency would make semantic caching an engineering decision rather than a gamble.

11.7 Composing cache layers

Challenge. Response, retrieval, prompt and KV caches interact: a response-cache hit makes the others irrelevant for that request, and a prompt structure that favors one layer may not favor another. **Research direction.** Serving systems that co-design scheduling and cache placement, such as KV-cache-centric architectures [54] and cache-aware routing [23]. **Opportunity.** An end-to-end view of LLM caching, from the application to the accelerator, would replace layer-by-layer tuning.

11.8 Robustness against adversarial use

Challenge. Attackers can poison shared caches or craft queries that collide with popular entries [48], [49]. **Research direction.** Admission control, trust levels, response-side verification [38] and anomaly detection for cache entries. **Opportunity.** Caches that resist manipulation would make it safe to share them across users.

12. What this means for platform decisions

Start with the caches that cannot change an answer: structure prompts so that stable content comes first, use provider-side prompt caching or, for self-hosted models, prefix KV caching with tenant isolation, and cache exact repeats. Add semantic caching only where questions repeat in different words and the cost of an occasional wrong answer is acceptable. Scope it by model, prompt version, tenant and permissions; choose its threshold from evaluation on representative queries rather than from a default; and keep unverified responses out of shared tiers. Link cached responses to the sources and versions that produced them so they can be invalidated, and bypass the cache for personal and live data. Measure the correctness of hits as well as savings, in production.

For semantic caching on a vector-capable in-memory store, Bassirah works with Redis as a Strategic Technology Partner. RedisVL's semantic cache and Redis vector search cover the patterns in Sections 4 and 5, and LangCache offers them as a service in preview. For tracing

and evaluating LLM applications, including the quality of cached responses, Bassirah works with Langfuse as a Strategic Technology Partner. Other products and projects named in this paper are ecosystem references. In each case, validate the design on representative workloads, failure scenarios and versions before relying on it.

References

1. Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S. and Bianchini, R. (2024). Splitwise: Efficient Generative LLM Inference Using Phase Splitting. 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), 118–132. (<https://doi.org/10.1109/ISCA59077.2024.00019>)
2. Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X. and Zhang, H. (2024). DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), 193–210. (<https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin>)
3. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H. and Stoica, I. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23), 611–626. (<https://doi.org/10.1145/3600006.3613165>)
4. Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. et al. (2024). SGLang: Efficient Execution of Structured Language Model Programs. Advances in Neural Information Processing Systems 37 (NeurIPS 2024), 62557–62583. (<https://doi.org/10.52202/079017-2000>)
5. 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. (<https://doi.org/10.18653/v1/2023.nlposs-1.24>)
6. Reimers, N. and Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), 3980–3990. (<https://doi.org/10.18653/v1/D19-1410>)
7. Regmi, S. and Pun, C. P. (2024). GPT Semantic Cache: Reducing LLM Costs and Latency via Semantic Embedding Caching. arXiv preprint 2411.05276. (<https://arxiv.org/abs/2411.05276>)
8. Li, J., Xu, C., Wang, F., von Riedemann, I. M., Zhang, C. and Liu, J. (2024). SCALM: Towards Semantic Caching for Automated Chat Services with Large Language Models. 2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS), 1–10. (<https://doi.org/10.1109/IWQoS61813.2024.10682957>)
9. Gill, W., Elidrissi, M., Kalapatapu, P., Ahmed, A., Anwar, A. and Gulzar, M. A. (2025). MeanCache: User-Centric Semantic Caching for LLM Web Services. 2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 1298–1310. (<https://doi.org/10.1109/IPDPS64566.2025.00117>)
10. Schroeder, L. G., Desai, A., Cuadron, A., Chu, K., Liu, S., Zhao, M., Krusche, S., Kemper, A., Zaharia, M. and Gonzalez, J. E. (2026). vCache: Verified Semantic Prompt Caching. International Conference on Learning Representations (ICLR 2026), poster. (<https://iclr.cc/virtual/2026/poster/10006471>; arXiv preprint 2502.03771)

11. Liu, X., Atalar, B., Dai, X., Zuo, J., Wang, S., Lui, J. C. S., Chen, W. and Joe-Wong, C. (2026). Semantic Caching for Low-Cost LLM Serving: From Offline Learning to Online Adaptation. IEEE INFOCOM 2026 – IEEE Conference on Computer Communications, 1–10. (<https://doi.org/10.1109/INFOCOM59046.2026.11571467>)
12. Agarwal, S., Mitra, S., Chakraborty, S., Karanam, S., Mukherjee, K. and Saini, S. K. (2024). Approximate Caching for Efficiently Serving Text-to-Image Diffusion Models. 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), 1173–1189. (<https://www.usenix.org/conference/nsdi24/presentation/agarwal-shubham>)
13. Bergman, S., Kermarrec, A.-M., Petrescu, D., Pires, R., Randl, M., de Vos, M. and Zhang, J. (2025). Leveraging Approximate Caching for Faster Retrieval-Augmented Generation. Proceedings of the 26th International Middleware Conference (Middleware '25), 340–353. (<https://doi.org/10.1145/3721462.3770776>)
14. Jin, C., Zhang, Z., Jiang, X., Liu, F., Liu, S., Liu, X. and Jin, X. (2025). RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. ACM Transactions on Computer Systems, 44(1), 1–27. (<https://doi.org/10.1145/3768628>)
15. Agarwal, S., Sundaresan, S., Mitra, S., Mahapatra, D., Gupta, A., Sharma, R., Kapu, N. J., Yu, T. and Saini, S. (2025). Cache-Craft: Managing Chunk-Caches for Efficient Retrieval-Augmented Generation. Proceedings of the ACM on Management of Data, 3(3), 1–28. (<https://doi.org/10.1145/3725273>)
16. Anthropic (2026). Prompt caching. Claude Developer Platform documentation. (<https://platform.claude.com/docs/en/docs/build-with-claude/prompt-caching>) Official documentation, accessed 26 October 2026.
17. OpenAI (2026). Prompt caching. OpenAI API documentation. (<https://developers.openai.com/api/docs/guides/prompt-caching>) Official documentation, accessed 26 October 2026.
18. Google (2026). Context caching. Gemini API documentation, Google AI for Developers. (<https://ai.google.dev/gemini-api/docs/caching>) Official documentation, accessed 26 October 2026.
19. Amazon Web Services (2026). Prompt caching for faster model inference. Amazon Bedrock User Guide. (<https://docs.aws.amazon.com/bedrock/latest/userguide/prompt-caching.html>) Official documentation, accessed 26 October 2026.
20. Google (2026). Context caching. Gemini Generate Content API (Legacy) documentation, Google AI for Developers. (<https://ai.google.dev/gemini-api/docs/generate-content/caching>) Official documentation, accessed 26 October 2026.
21. vLLM project (2026). Automatic Prefix Caching. vLLM documentation (design docs). (https://docs.vllm.ai/en/latest/design/prefix_caching/) Official documentation, accessed 26 October 2026.
22. NVIDIA (2026). KV cache reuse. TensorRT-LLM documentation. (<https://nvidia.github.io/TensorRT-LLM/latest/legacy/advanced/kv-cache-reuse.html>) Official documentation, accessed 26 October 2026.
23. NVIDIA (2026). Routing Concepts: KV Cache-Aware Routing. NVIDIA Dynamo documentation. (<https://docs.nvidia.com/dynamo/components/router/routing-concepts>) Official documentation, accessed 26 October 2026.
24. Redis (2026). Redis LangCache. Redis documentation (Develop → AI → Redis Iris context engine). (<https://redis.io/docs/latest/develop/ai/context-engine/langcache/>) Official documentation, accessed 26 October 2026.

25. Zhang, Q., Wornow, M. and Olukotun, K. (2025). Agentic Plan Caching: Test-Time Memory for Fast and Cost-Efficient LLM Agents. *Advances in Neural Information Processing Systems* 38 (NeurIPS 2025), 114372–114398. (<https://doi.org/10.52202/085713-3451>)
26. Pan, Z., Patel, A., Shen, Y., Hu, Z., Guan, Y., Li, W.-L., Qin, L., Wang, Y. and Ding, Y. (2025). KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows. *Advances in Neural Information Processing Systems* 38 (NeurIPS 2025), 139912–139931. (<https://doi.org/10.52202/085713-4208>)
27. Zhu, H., Zhu, B. and Jiao, J. (2024). Efficient Prompt Caching via Embedding Similarity. *arXiv preprint 2402.01173*. (<https://arxiv.org/abs/2402.01173>)
28. Gill, W., Cechmanek, J., Hutcherson, T., Rajamohan, S., Agarwal, J., Gulzar, M. A., Singh, M. and Dion, B. (2025). Advancing Semantic Caching for LLMs with Domain-Specific Embeddings and Synthetic Data. *arXiv preprint 2504.02268*. (<https://arxiv.org/abs/2504.02268>)
29. Muennighoff, N., Tazi, N., Magne, L. and Reimers, N. (2023). MTEB: Massive Text Embedding Benchmark. *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, 2014–2037. (<https://doi.org/10.18653/v1/2023.eacl-main.148>)
30. Malkov, Y. A. and Yashunin, D. A. (2020). 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. (<https://doi.org/10.1109/TPAMI.2018.2889473>)
31. Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L. and Jégou, H. (2026). The Faiss Library. *IEEE Transactions on Big Data*, 12(2), 346–361. (<https://doi.org/10.1109/TBDDATA.2025.3618474>)
32. pgvector contributors (2026). pgvector: Open-source vector similarity search for Postgres (README). GitHub. (<https://github.com/pgvector/pgvector>) Official documentation, accessed 26 October 2026.
33. Redis (2026). Vector search concepts. Redis documentation. (<https://redis.io/docs/latest/develop/ai/search-and-query/vectors/>) Official documentation, accessed 26 October 2026.
34. Singh, A. K., Wang, H., Attaluri, L. N. S., Chiam, T. and Zhu, W. (2026). Asynchronous Verified Semantic Caching for Tiered LLM Architectures. *arXiv preprint 2602.13165*. (<https://arxiv.org/abs/2602.13165>)
35. Redis (2026). LLM Cache (RedisVL API reference, version 0.27.2). Redis documentation. (<https://redis.io/docs/latest/develop/ai/redisvl/0.27.2/api/cache/>) Official documentation, accessed 26 October 2026.
36. Redis (2026). Getting Started with Redis Iris: Context & Memory for AI Agents. Redis tutorials (published 16 June 2026, updated 20 June 2026). (<https://redis.io/tutorials/getting-started-with-redis-iris/>) Official documentation, accessed 26 October 2026.
37. Wang, C., Liu, X., Zhu, Y., Youssef, A., Nagpurkar, P. and Chen, H. (2025). Category-Aware Semantic Caching for Heterogeneous LLM Workloads. *arXiv preprint 2510.26835*. (<https://arxiv.org/abs/2510.26835>)
38. Liang, J., Wang, Y., Jiang, T. and Wang, T. (2026). LaCache: Robust Semantic Caching for LLM Serving. *arXiv preprint 2608.01718*. (<https://arxiv.org/abs/2608.01718>)

39. Baral, A., Ralev, R., Zhechev, I. S., Rajamohan, S. and Agarwal, J. (2026). Closing the Operational Gap in Semantic Caching. arXiv preprint 2606.19719. (<https://arxiv.org/abs/2606.19719>)
40. Redis (2026). Use the LangCache API and SDK. Redis documentation. (<https://redis.io/docs/latest/develop/ai/context-engine/langcache/api-examples>) Official documentation, accessed 26 October 2026.
41. Redis (2026). Redis Iris context engine. Redis documentation. (<https://redis.io/docs/latest/develop/ai/context-engine/>) Official documentation, accessed 26 October 2026.
42. Langfuse (2026). Caching (Prompt Management). Langfuse documentation. (<https://langfuse.com/docs/prompt-management/features/caching>) Official documentation, accessed 26 October 2026.
43. Yan, J., Ni, W., Chen, L., Lin, X., Cheng, P., Qin, Z. and Ren, K. (2025). ContextCache: Context-Aware Semantic Cache for Multi-Turn Queries in Large Language Models. Proceedings of the VLDB Endowment, 18(12), 5391–5394. (<https://doi.org/10.14778/3750601.3750679>)
44. Yu, C., Wang, T., Shao, Z. and Jiang, S. (2025). SmartCache: Context-aware Semantic Cache for Efficient Multi-turn LLM Inference. Advances in Neural Information Processing Systems 38 (NeurIPS 2025), 191160–191181. (<https://doi.org/10.52202/085713-5740>)
45. Gu, C., Li, X. L., Kudithipudi, R., Liang, P. and Hashimoto, T. (2025). Auditing Prompt Caching in Language Model APIs. Proceedings of the 42nd International Conference on Machine Learning, PMLR 267, 20477–20496. (<https://proceedings.mlr.press/v267/gu25b.html>)
46. Song, L., Pang, Z., Wang, W., Wang, Z., Wang, X., Chen, H., Song, W., Jin, Y., Meng, D. and Hou, R. (2025). The Early Bird Catches the Leak: Unveiling Timing Side Channels in LLM Serving Systems. IEEE Transactions on Information Forensics and Security, 20, 11431–11446. (<https://doi.org/10.1109/TIFS.2025.3622954>)
47. Zheng, X., Han, H., Shi, S., Fang, Q., Du, Z., Hu, X. and Guo, Q. (2026). InputSnatch: Stealing Input in LLM Services via Cache-Sharing Timing Side-Channel Attacks. Advanced Parallel Processing Technologies (Lecture Notes in Computer Science), Springer, 483–497. (https://doi.org/10.1007/978-981-92-4805-6_32)
48. Wu, G., Wang, T., Zhang, Y., Zhang, Z., Niu, J., Wu, Y. and Zhang, Y. (2026). When Cache Poisoning Meets LLM Systems: Semantic Cache Poisoning and Its Countermeasures. Network and Distributed System Security (NDSS) Symposium 2026. (<https://doi.org/10.14722/ndss.2026.240200>)
49. Zhang, Z., Liu, Z., Xie, Y., Huang, Q. and She, D. (2026). From Similarity to Vulnerability: Key Collision Attack on LLM Semantic Caching. Proceedings of the 43rd International Conference on Machine Learning, PMLR 306, 157295–157309. (<https://proceedings.mlr.press/v306/zhang26dt.html>)
50. Mansoor, M., Ahmad, T. and Yoon, Y.-C. (2026). Risk-Constrained Freshness-Aware Semantic Caching for Open-Web Retrieval-Augmented LLMs. arXiv preprint 2607.04281. (<https://arxiv.org/abs/2607.04281>)
51. Gim, I., Chen, G., Lee, S., Sarda, N., Khandelwal, A. and Zhong, L. (2024). Prompt Cache: Modular Attention Reuse for Low-Latency Inference. Proceedings of Machine Learning and Systems 6 (MLSys 2024). (https://proceedings.mlsys.org/paper_files/paper/2024/hash/a66caa1703fe34705a4368c3014c1966-Abstract-Conference.html)

52. Yao, J., Li, H., Liu, Y., Ray, S., Cheng, Y., Zhang, Q., Du, K., Lu, S. and Jiang, J. (2025). CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25), 94–109. (<https://doi.org/10.1145/3689031.3696098>)
53. Gao, B., He, Z., Sharma, P., Kang, Q., Jevdjic, D., Deng, J., Yang, X., Yu, Z. and Zuo, P. (2024). Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. 2024 USENIX Annual Technical Conference (USENIX ATC 24), 111–126. (<https://www.usenix.org/conference/atc24/presentation/gao-bin-cost>)
54. Qin, R., Li, Z., He, W., Cui, J., Ren, F., Zhang, M., Wu, Y., Zheng, W. and Xu, X. (2025). Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. 23rd USENIX Conference on File and Storage Technologies (FAST 25), 155–170. (<https://www.usenix.org/conference/fast25/presentation/qin>)
55. Liu, Y., Li, H., Cheng, Y., Ray, S., Huang, Y., Zhang, Q., Du, K., Yao, J., Lu, S., Ananthanarayanan, G. et al. (2024). CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. Proceedings of the ACM SIGCOMM 2024 Conference, 38–56. (<https://doi.org/10.1145/3651890.3672274>)
56. Liu, Y., Cheng, Y., Yao, J., An, Y., Chen, X., Feng, S., Huang, Y., Shen, S., Zhang, R., Du, K. et al. (2025). LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. arXiv preprint 2510.09665. (<https://arxiv.org/abs/2510.09665>)
57. LMCache project (2026). Welcome to LMCache! LMCache documentation. (<https://docs.lmcache.ai/>) Official documentation, accessed 26 October 2026.
58. Langfuse (2026). Observability Data Model (Concepts). Langfuse documentation. (<https://langfuse.com/docs/observability/data-model>) Official documentation, accessed 26 October 2026.
59. Langfuse (2026). Token & Cost Tracking. Langfuse documentation. (<https://langfuse.com/docs/observability/features/token-and-cost-tracking>) Official documentation, accessed 26 October 2026.
60. Langfuse (2026). Evaluation Overview. Langfuse documentation. (<https://langfuse.com/docs/evaluation/overview>) Official documentation, accessed 26 October 2026.
61. OpenTelemetry (2026). Semantic conventions for generative client AI spans; Gen AI attribute registry. OpenTelemetry GenAI semantic conventions repository (`open-telemetry/semantic-conventions-genai`). (<https://github.com/open-telemetry/semantic-conventions-genai>) Official documentation, accessed 26 October 2026.

Product and project names are trademarks of their respective owners. Their mention describes publicly documented technology and does not imply a business relationship, except where Bassirah names a Strategic Technology Partner.