ChatGPT

Prompt Caching in Modern LLM Systems

Prompt Caching Strategies

Exact-Match Caching: The simplest form of caching uses the full prompt string (plus relevant parameters) as the key. If the exact same prompt was seen before (including system instructions and input), the stored completion is returned instead of recomputing it [1] [2]. This approach treats the cache like a hash map: any change in the prompt (even a single character or timestamp) results in a cache miss. Exact caching is easy to implement (e.g. using an in-memory `dict` or a SQL table keyed by prompt) and has virtually no runtime overhead on a hit. However, it has low tolerance for variation – *“Hello” vs “Hi”* would be treated as different prompts and not hit the same cache [3] [4]. Frameworks like LangChain implement exact caching by default; for instance, LangChain’s `InMemoryCache` uses the prompt text plus LLM config as a key, and returns a saved `Generation` if available [5] [6]. The trade-off is that exact caching maximizes precision (only returns a response when the prompt truly matches), but misses opportunities to reuse work for semantically similar queries.

Semantic Caching: A more flexible strategy is to cache by *meaning* of the prompt rather than exact text. This involves computing an embedding (vector representation) of each new prompt and comparing it to embeddings of past prompts stored in the cache [7] [8]. If a new query is semantically similar to a previous one (above some similarity threshold), the system reuses the previous answer [9] [10]. For example, *“What foods are known in Madrid?”* vs *“What do people usually eat in the capital of Spain?”* convey the same intent; a semantic cache would recognize this and return the cached answer for the latter without calling the LLM again [9] [10]. Implementation-wise, semantic caching relies on a **vector store** or index: common choices include libraries like FAISS or Redis with vector search enabled (e.g. Redis Stack), or cloud vector databases like Milvus, Pinecone, etc [11] [12]. Each prompt’s embedding is stored, along with the LLM response, enabling fast *nearest-neighbor* lookup for new queries. The advantages are significant: higher cache hit rates (catching rephrased questions) lead to greater cost savings – real-world reports suggest 50–90% reduction in API calls for applications with repetitive queries [13] [14] – and more consistent answers (the same underlying question gets the same answer every time) [15]. The downsides include the overhead of generating and searching embeddings for each request, and the risk of false positives if the similarity threshold isn’t tuned well (returning a cached answer for a query that was actually meaningfully different) [16]. There’s also a maintenance aspect: if the factual context changes, semantic caches might serve an outdated answer unless invalidated (more on that later) [16]. Still, this strategy is powerful for caching **open-ended or natural language inputs** where exact string matching is too brittle.

Hierarchical & Multi-Level Caching: In practice, modern systems often combine caching techniques or apply caching at multiple prompt granularities. *Hierarchical caching* refers to breaking the prompt into segments or layers that can be cached independently [17] [18]. For example, you might cache a large system prompt or a chunk of documentation separately from the user’s query. If only the user query changes, the system can reuse the cached embeddings/state for the static context portion [19] [20]. This is the idea behind Anthropic’s *cache breakpoints*: you can mark up to four sections of a prompt to cache, such as a tools definition, system message, and conversation history, each of which can be reused until that particular section changes [21] [22]. Hierarchical caching yields a finer control over invalidation (e.g. update one section's cache while others remain valid) and can reduce memory usage by not storing an entire huge prompt as one key when much of it overlaps with other prompts.

Multi-level caching also means having layers like **L1 vs L2 caches**, similar to CPU caches. An LLM application might first check a fast in-memory cache (L1) for a prompt; if not found, check a more persistent Redis or database cache (L2); if still not found, call the model. This ensures maximum hit rate with minimal latency on hits. Tools like **GPTCache** support chaining caches: you can configure an exact-match cache to try first and a semantic cache as a fallback, combining the precision of one with the recall of the other [23] [24]. In terms of implementation, in-memory caches (using Python dictionaries or similar) are fastest but volatile (cleared on restart) and typically single-node. Distributed caches (Redis, Memcached) or file-based caches (SQLite) persist across sessions and can be shared by multiple app servers [25] [26]. Vector stores (for semantic caching) add operational complexity but can also be distributed and scalable. The choice of strategy often depends on the use case requirements: for instance, a **local hobby app** might just use an in-memory exact cache for simplicity, whereas a **production FAQ bot** with high traffic and rephrased questions would benefit from a semantic cache backed by a persistent database for scale.

Efficiency Trade-offs: All caching strategies aim to save time and cost, but they introduce overhead in checking the cache and storing data. Exact match caches have very low overhead (a hash table lookup) and thus are almost always worth it if there's any chance of repetition. Semantic caches incur extra work – generating an embedding and doing a similarity search – so they pay off mainly when queries are likely to repeat in meaning. As a rule of thumb, prompts that are long (costly to process) and commonly reused should be cached; unique or one-off prompts might bypass caching. OpenAI's own platform only engages caching for prompts above 1024 tokens [27] [28], since caching short prompts gives minimal benefit. When implemented well, prompt caching yields dramatic wins: OpenAI reports up to **90% input cost reduction** and **80% latency reduction** on cache hits for long prompts [29] [30]. In a Medium-scale application, developers have noted they can *"slash AI costs by 80%"* by introducing smart caching rather than calling the model redundantly [31] [32]. The key trade-off is ensuring the cache hit rate is high enough to justify any added complexity – which is usually the case when user queries have significant overlap or when the app uses large static context prefixes.

Use Cases

Reusable Inference Pipeline Steps: Many LLM applications perform multi-step processing, where the same prompt or sub-prompt may be used repeatedly. Prompt caching prevents wasteful recomputation in these scenarios. For example, imagine an agent that plans and then executes tasks by querying an LLM multiple times – if one step involves asking the same question or using the same tool instruction repeatedly, those intermediate prompts can be cached. In customer support chatbots, thousands of users might ask variants of a common question (*"How do I reset my password?"*); with caching, the answer for the first user is stored and all subsequent users get an instant response retrieved from cache [33] [34]. This not only cuts costs by reducing API calls, but also improves response time for the user. In batch processing pipelines, if a large document needs to be analyzed multiple times with slight variations of a query, caching the analysis results can avoid re-invoking the model for each variation. Another scenario is **retrieval-augmented generation (RAG)**: if your pipeline constructs a prompt with retrieved context (e.g. a wiki article) and queries the LLM, you can cache the LLM's answer for that document+question pair. Then, if a similar question on the same document comes up, the system can return the cached answer immediately. This is essentially memoization of the LLM's work in an inference pipeline. Overall, any time an LLM's output will be reused – whether across different users, or multiple times in a chain of reasoning – that's an opportunity for prompt caching to *"avoid redundant processing of identical prompt segments"* [35] .

Chatbot Memory and Long Contexts: Multi-turn conversations with LLMs pose a challenge: to maintain context, the conversation history (or a compressed form of it) must be included in each new prompt, which grows over time. Prompt caching offers ways to handle long contexts more efficiently. One approach is *prefix caching* during inference: the idea is to avoid re-sending and re-processing the entire chat history in each turn. For instance, OpenAI's system automatically caches the model's internal state after processing a long prompt prefix (like the system message and first N user-assistant exchanges) [36] [37] . On the next user question, the API can accept just the new content and retrieve the cached state for the earlier tokens, essentially "reconstructing" the conversation state without re-reading all previous messages. From the developer's perspective, it looks like the model magically remembers the prefix; under the hood it's loading a cached key/value matrix for those earlier tokens [38] [39] . This technique allows chatbots to maintain extensive context at lower cost – OpenAI's caching can reduce token usage by up to 90% for the cached portion [29] . Another use of caching in chat systems is to store **summaries or embeddings of past interactions**. For example, after 100 messages, a chatbot might summarize the conversation so far. Caching that summary (or its embedding) means that whenever the summary is reused in prompts, it doesn't need to be recomputed by asking the model again. Similarly, conversation *memory* stored in a vector database (typical in many chat frameworks) is effectively a semantic cache of important pieces of context: when a new user query comes in, the bot retrieves semantically relevant past facts from the cache instead of expecting an exact match in the raw history. This is akin to semantic prompt caching where the query is the conversation turn and the cache lookup finds a related prior answer or detail to include. In summary, caching in chat systems helps with **state reconstruction** – be it via prefix state reuse or retrieval of stored summaries – enabling longer, coherent dialogues without repeatedly paying for the same context tokens.

Integration with Tools and External Knowledge: Prompt caching also plays a role when LLMs are used in tool-using scenarios or with external data. Many applications use frameworks (like LangChain or LlamaIndex) where the LLM may call tools (APIs, calculators, etc.) or query documents in between generating prompts. In such cases, the orchestration often involves prompts that don't change across runs. For example, a tool specification (such as a JSON schema or function description for the model to use) can be cached as part of the prompt so that multiple calls to the tool don't repeatedly reintroduce the same spec. Anthropic's API explicitly allows caching *tool definitions* and instructions in a conversation via the `cache_control` parameter [21] [22] . A real-world scenario: say your LLM is using a *weather API tool*. The definition of that tool (its name, inputs, etc.) can be marked cacheable and reused across many queries [40] [41] , so the model doesn't have to ingest that definition from scratch each time a user asks for weather. Another use case is with **large documents or knowledge bases**. If your system frequently sends chunks of a document (say the company handbook) to the LLM to answer questions, you can cache the prompt+response for those chunks. Better yet, if using a *hierarchical cache*, you cache the document context separately – e.g. using OpenAI's caching by giving it as a system message with `cache_control: "ephemeral"` – so that any question about that same document reuses the cached context processing [42] [43] . This dramatically speeds up *research assistants* or *document Q&A* apps, since the model isn't re-reading the same background text on every query. In frameworks like LlamaIndex, which build an index of documents, caching can be used to store intermediate results (like node embeddings or prior query answers) to expedite future queries. In all these cases, the integration of caching ensures that **expensive prompt components (tools, docs, examples)** are only paid for once and then "remembered" for subsequent use.

Lifecycle and Update Patterns

When to Cache: Deciding *what and when* to cache is a crucial design consideration. Caching every single prompt indiscriminately may not be efficient – for prompts that will never repeat, storing them just adds overhead. Good caching policy starts by identifying **high-value prompts**: those that are costly (long prompts consuming many tokens) and/or likely to be reused. A common guideline is to cache large static prefixes or contexts. For instance, OpenAI's system only enables prompt caching on prompts longer than 1024 tokens [27], because below that, the performance gain is negligible. Similarly, Anthropic's Claude requires a conversation to reach 1024 tokens before it even allows creating a cache checkpoint [44]. In your application, you might decide to cache prompts that contain lengthy instructions or backgrounds (even if the user's question part is short). Repeated **system messages**, few-shot examples, or documentation chunks are prime candidates for caching [45] [46]. On the other hand, prompts that contain highly dynamic data – e.g. a prompt that includes the current timestamp or a user's name in every request – are poor candidates for caching (each prompt would be unique). It can be useful to preprocess prompts to **extract variables**: for example, instead of caching the entire prompt with a user's name, refactor the prompt to a constant template plus the name as input, so the template can be cached once [47] [48]. In summary, enable caching for prompts/parts that are *stable* across requests, and bypass or disable it for those that are always unique. Some frameworks let you specify this via code or metadata (e.g., Anthropic's `cache_control` can mark which parts to cache and which to skip) [21] [22]. Developers should also monitor cache usage over time to adjust what to cache – if certain queries never hit, maybe they shouldn't be cached at all.

Cache Invalidation Strategies: The classic hard problem of caching is invalidation – i.e. ensuring the cache doesn't serve stale or incorrect data after something changes. In prompt caching for LLMs, one must consider changes in *content* and *context*. Several strategies are used in practice:

- **Time-To-Live (TTL) Expiry:** Many provider-side caches use a TTL to automatically purge entries after a certain time. For example, Anthropic's *ephemeral cache* entries live for 5 minutes by default [22] [49] (and this is refreshed on each use, meaning an active item stays alive) [22]. Amazon Bedrock also uses a 5-minute TTL for Claude and other models' context caches [50]. This ensures that even if something changes, the cached data won't stick around forever. Some systems allow longer retention – OpenAI offers an "extended" prompt cache for certain models, retaining up to 24 hours by offloading cache to disk [51]. In your own caching layer (like a Redis cache you control), you might set a TTL for certain keys. For instance, if you cache a response to "What are today's news headlines?", you might expire it after an hour so that tomorrow it doesn't give yesterday's news. TTL-based invalidation is simple and works well for data that naturally updates after a known interval.

- **Manual Invalidation & Versioning:** In cases where the correct answer can change unpredictably (or you deploy an updated LLM prompt), you'll want a way to manually bust the cache. One technique is **version tagging**. Suppose you have a cached prompt that includes a knowledge base or policy that gets updated periodically – you can include a version number in the prompt (e.g. "Instructions v2.0: ...") [52] [53]. When you update your content, you increment the version, which effectively changes the prompt key and prevents using the old cached result. This way, the cache naturally invalidates anything with the old version string. Another manual approach is to provide an API or admin tool to flush certain cache entries when data changes. For semantic caches, invalidation might mean removing vectors related to a certain document or topic if that document was updated. The **cache-aside** pattern is helpful: the application can choose not to use the cache for certain operations if it knows the data changed (e.g. skip cache and recompute the answer, then update the cache). In frameworks like LangChain, you can always call `llm.clear_cache()` or simply not set the cache during a run where fresh output is needed.

- **Capacity Eviction (LRU/FIFO):** Caches have finite storage, so even if content is still valid, old entries might need eviction to make room for new ones. A common policy is **Least Recently Used (LRU)** – evict the entry that hasn't been used in the longest time [54] [55]. This works on the principle that if something hasn't been accessed in a while, it's less likely to be needed again soon (which often holds true). LangChain's in-memory cache for example allows a `maxsize` setting; once the number of items exceeds that, the oldest items are dropped automatically [54]. GPTCache also supports LRU eviction when you set a max cache size, as well as a FIFO option (first-in, first-out) [56] [55]. Choosing an eviction policy depends on usage patterns – LRU is usually a sane default for interactive systems. In some cases, you might evict based on other metrics; e.g., you could remove entries that are expensive to store (very large prompts) sooner, or use a combination of time + size (e.g., anything older than 1 day). It's important to log cache hits and misses to ensure your eviction policy isn't throwing away items that were actually still useful.

- **Granular Invalidations:** Hierarchical caching shines here. By caching parts of prompts separately, you can invalidate just the portions that changed. For instance, if you cached a system prompt and a document snippet independently, and the document updates, you can drop the document's cache while still reusing the system prompt's cache. Anthropic's multi-breakpoint caching implies this: each cached segment is handled individually, so you could, say, replace the cached "recent updates" segment daily while keeping the long-term segments [19] [20]. Another mechanism is using **cache keys that include context identifiers**. Imagine a cache key template like `Answer[{doc_id}][{user_question}]`. If doc_id changes (i.e., the user asks about a different document), it's naturally a different key. If the document's content changes, you would assign it a new doc_id (or content hash) so that queries start populating a new set of keys. This way, you don't serve answers based on outdated content. In summary, structure your caching strategy so that it aligns with the granularity of updates in your data: cache things at a chunk size or scope that makes sense for invalidation when that thing updates.

Cache Updates & Aging: Once a cache entry is stored, how is it maintained over time? We've covered automatic expiration and eviction. Another aspect is **refreshing** caches. Some systems proactively refresh certain cache entries before they expire if they are critical. For instance, if you know a particular prompt is extremely common (say the system prompt every request uses), you might periodically re-run it through the LLM to update the cache (especially if the LLM's behavior might drift or you have a non-deterministic model with temperature – though in many caching scenarios, we cache only when using deterministic settings). In most LLM usage, deterministic outputs (temperature 0) are preferred for caching, since you want the same answer each time. If using temperature > 0 (stochastic outputs), some cache systems choose not to return a cached answer at all, or only do so under certain conditions. For example, GPTCache lets you configure a threshold with the model's `temperature` to decide when to bypass cache – at high temperature (more randomness), it's more likely to skip the cache to get a fresh variation [57] [58]. Over time, you should also consider **cache aging policies**. If an entry hasn't been accessed in a long time, even if it hasn't hit TTL or size limits, you might want to clear it out (to save space or because it's likely irrelevant now). In practice, a combination of LRU (which inherently ages out old entries) and TTL (which guards against indefinitely storing things) keeps the cache healthy. Monitoring metrics like *cache hit rate*, *average latency improvement*, and *cost savings* helps decide if your caching strategy needs adjusting. For instance, if you see a lot of cache misses for semantically similar queries, you might loosen the similarity threshold to cache more aggressively [59]. Or if your cache hit rate is very low, perhaps the content is too dynamic and you'd disable caching for those parts to avoid overhead. The lifecycle of cached prompts should thus be managed with a balance: keep the cache "fresh" enough to be correct and useful, but "sticky" enough to capture long-term repeated knowledge.

Architectural and Conceptual Models

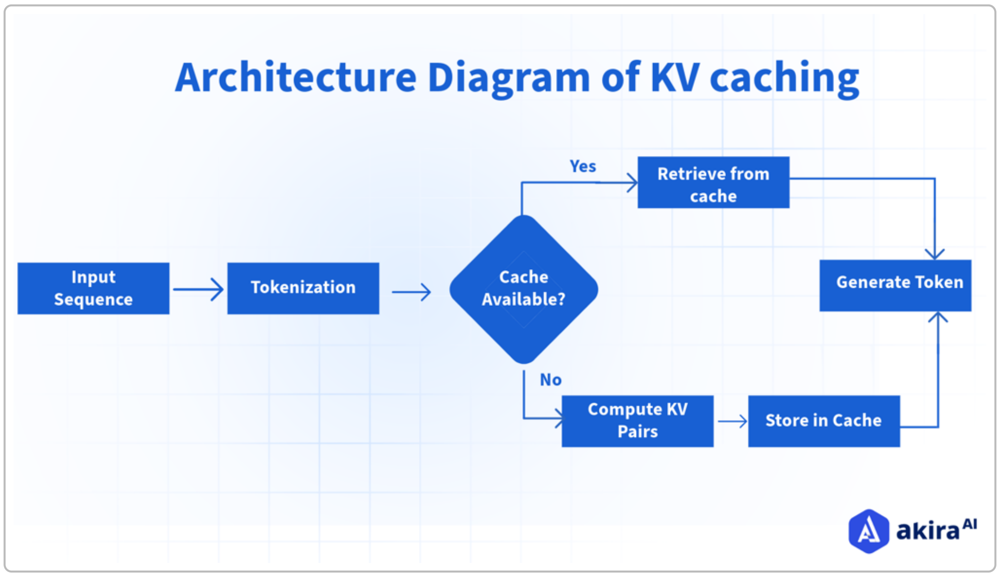
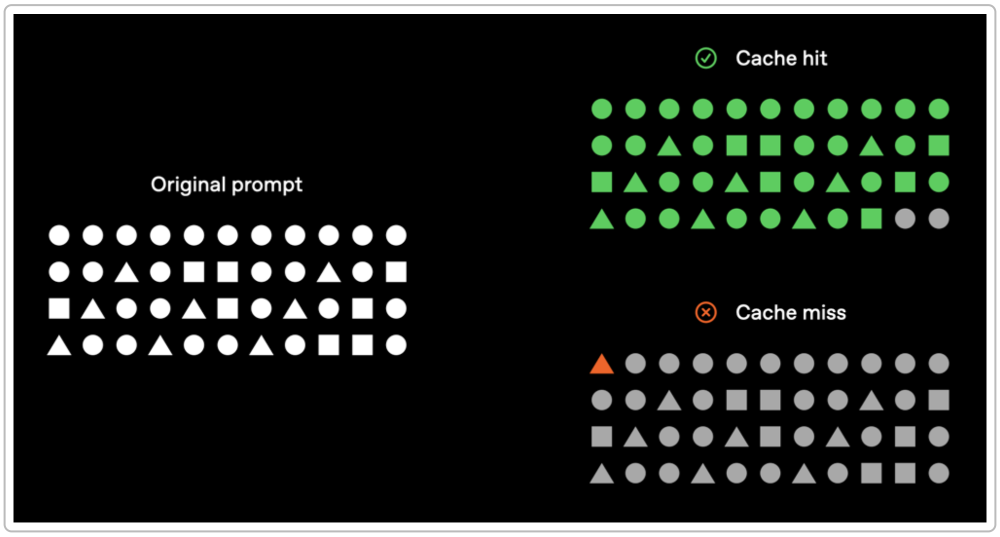


Illustration of caching within an LLM's processing. If a cached prefix state is available (green path), the model retrieves stored key/value tensors and can immediately generate output tokens using them, skipping the redundant computation. If no cache exists (red path), the model must process the entire input from scratch (e.g. compute all attention layers for the prompt) and then it stores the resulting state for future reuse [38] [39].

Client-Side Cache Layer: At the application architecture level, prompt caching is often implemented as an intercepting layer in front of the LLM API. The flow is straightforward: when a request comes in, the app first **checks the cache** (a local or network store) using a key derived from the prompt [60] [2]. If a cached result is found (cache hit), it can return the answer immediately without contacting the LLM service. If not (cache miss), the app calls the LLM as usual, gets the result, and then **updates the cache** by storing the prompt and response for next time [28]. This pattern fits into many web service architectures; it's analogous to caching database query results or HTTP responses. The cache can be as simple as an in-memory object in a single server, or a distributed cache shared across many servers. The latter is important for scaling: if you have multiple instances of your chatbot running, a distributed cache (like a Redis cluster) ensures that if one instance has seen a prompt, another instance can reuse that answer too. From a conceptual viewpoint, the cache layer sits between the *business logic* (the part of your app that decides what prompt to send) and the *LLM inference endpoint*. In designs like LangChain, the cache is integrated such that any call to the LLM goes through a cache check if enabled [5]. One must also consider the cache storage architecture: it might include a fast lookup index (for semantic search) and a storage backend for the data. For example, GPTCache's architecture has a **data manager** that encapsulates both a scalar store (for exact keys and metadata) and a vector store (for embeddings) [61] [62]. This modular design lets you plug in different databases depending on your needs, but conceptually it's still a component that the application queries *before* calling the LLM.

Model/Provider-Side Cache: In addition to (or sometimes instead of) client-side caching, modern LLM providers implement their own caching at the model inference level. This often takes the form of *context caching* – the model remembers the computation for a prefix of the prompt so that it can skip that part next time. Architecturally, this is achieved by storing the model's internal state (the key/value tensors from Transformer attention layers) after processing the prefix [63] [64]. When a new prompt arrives that starts with the same prefix, the service can **retrieve the cached state** and initialize the model with it, rather than recompute those layers. The OpenAI platform, for example, routes prompts by a hash of their prefix to ensure requests with the same prefix go to the same server where the cache resides [36]. If the prefix was seen recently on that server, it's a cache hit and the model only needs to process the *new* part of the prompt (appending to the cached states) [65] [37]. This yields much faster time-to-first-token and lower compute cost. In effect, the provider-side cache is like giving the model a **memory of recently seen contexts**. Google's PaLM (Gemini) calls this *"context caching"*, where developers explicitly create a cached content object on the server, then attach it to subsequent requests [66] [67]. Anthropic's Claude requires developers to send a special header to enable caching and mark sections of the prompt with `cache_control` flags [21]. In these cases, the architecture includes an API for cache management – e.g. creating and referencing cached content by an ID. Amazon Bedrock's implementation (for models like Claude or Amazon's own Nova models) introduces the concept of *cache checkpoints* in a conversation: you can insert a checkpoint after a certain token count, and the model will then preserve all state up to that point for reuse in the next turn [38] [68]. From a system design perspective, provider-side caching means the **inference servers** themselves have an ephemeral store (often in GPU memory or attached high-speed storage) where they keep recent key/value states. These are typically not accessible directly by users, but you influence them by how you structure prompts (identical prefixes = likely cache hit). One important conceptual point: *provider caches do not cache the output tokens*, only the *prefix interpretation*. So the model will still generate fresh output for each request, but it does so faster/cheaper because it didn't redo the earlier part of the prompt. The output can differ if there's randomness, but the cost is lower. This kind of caching is particularly beneficial for long conversations or big context usage, where each new query has, say, 90% the same tokens as the previous – the savings are huge. It effectively treats the static portion of your prompt as if it were part of the model's context memory.



Prompt caching requires identical prefixes for a hit. In this conceptual visualization, the system prompt and lengthy context (green blocks) remain the same between requests and thus can be cached as one unit; the only new content in the second request is the final user query (blue block), which the model processes on top of the *cached state. If any part of the prefix differed (additional orange block in the "miss" case), the cache would not apply* [30] [69] .

Hierarchical Caching Design: Complex LLM applications often have multiple layers of prompts or chains, and caching can be applied at each layer. For example, consider a chatbot that first uses an LLM call to summarize recent conversation (to condense context), then another call to answer the user's question using that summary. Both of those calls could benefit from caching: the summarization prompt (if it's triggered frequently with similar content) and the final answer prompt (especially if users ask repetitive questions). Hierarchical design can also refer to multi-tier caches in a deployment. One might use an in-memory cache on each server for super-fast hits on very hot prompts, and a shared persistent cache for less frequent prompts. The in-memory (L1) cache might have a small capacity but very low latency, while the persistent (L2) cache can store much more and survive restarts. Many web architectures use this pattern (e.g. an LRU memory cache + Redis as backup). In prompt caching, this ensures that the *first few* repeats of a prompt might hit the same server's RAM cache, and even if that server goes down or the item ages out of RAM, the next repeat on any server can still find it in Redis. Another hierarchical aspect is caching *within prompts*. As discussed, providers like Anthropic let you define multiple cache segments in one prompt [22] [21] . You can think of this as a hierarchy: level 1 cache = the first segment, level 2 = first+second segment, etc. If a later request shares only the first segment but not the second, it might hit on that first segment's cache but have to recompute the second. The concept is similar to how a CPU might have L1, L2, L3 caches of increasing size and lower speed – here a prompt can have sections cached at different lifetimes. Designing your application to take advantage of this means grouping your prompt into logical parts. For instance, *"system instructions"* (never change) could be one block, *"product documentation"* (updates weekly) another, and *"recent announcements"* (changes daily) a third [19] [70] . This way, if the daily segment changes, you only lose the cache on that one, while the first two segments still benefit from caching.

Design Principles: When incorporating prompt caching, a few guiding principles help maximize effectiveness. First, **stability and consistency** in prompts is key. If your prompt is constructed dynamically in different orders or formats each time, you will undercut the cache. It's recommended to structure prompts in a fixed order and format so that the static parts are byte-for-byte identical across requests [71] [72] . For example, don't randomly shuffle example questions in a few-shot prompt – pick a canonical order. Similarly, avoid including non-deterministic elements (like the current date, unless necessary) in the cacheable section. If something must change, isolate it to the end of the prompt or wherever the cache is meant to break. Second, design for **observability**. Treat the cache as an important component: monitor metrics such as cache hit rate, token savings, and latency improvements [73] [74] . This can be done by logging when a cache was used and how much latency or cost was saved (OpenAI's API even returns fields like `cache_read_tokens` to indicate how many tokens were served from cache [75] [76]). These metrics will guide you in tuning your caching (e.g., if hit rate is low, maybe your prompts aren't repeating as much as you thought, or your semantic threshold is too strict). Another principle is **fall back gracefully**: always handle cache misses correctly (they should trigger an LLM call). Also, be cautious during development and prompting experimentation – it's often wise to disable caching when you are prompting iteratively, to avoid confusing results. Some developers set up cache to ignore prompts that contain a special keyword or when in a "dev mode." Finally, remember that **caching doesn't alter the model's output**; it's an optimization behind the scenes. So your application logic should remain correct whether a response came from cache or from the model. By adhering to these design principles – consistent prompt structuring, monitoring, careful invalidation – caching becomes a robust layer in the architecture rather than a risky shortcut.

Framework and Library Examples

LangChain: As a popular framework for building LLM applications, LangChain includes prompt caching utilities to save costs during development and production. In LangChain's Python library, you can enable a global cache with `langchain.globals.set_llm_cache(...)`. For example, to use an in-memory cache: `set_llm_cache(InMemoryCache())` is a one-liner that will cause all LLM calls to first do a cache lookup [25] [77]. Under the hood, LangChain creates a key from the prompt text plus an identifier of the LLM model & parameters (so that, say, a GPT-4 response isn't incorrectly served to a GPT-3.5 prompt) [78] [79]. On a miss, it calls the LLM and then `update()`s the cache with the result. Several cache backends are provided: `InMemoryCache` (fast, ephemeral), `SQLiteCache` (stores to a local SQLite database file), and `RedisCache` (stores in a Redis instance for distributed use) [26] [80]. By default these are *exact-match* caches. Indeed, LangChain's docs note that the basic cache will miss if there's even a minor difference like an extra period or emoji [3]. To address this, LangChain introduced a **Semantic Cache** option: e.g. `RedisSemanticCache` which integrates a vector embedding and similarity search on top of Redis [11] [81]. With one line, you can swap the cache to semantic by providing an embedding model and setting a distance threshold for hits [82] [83]. LangChain does not enforce a particular eviction policy internally – if using SQLite or in-memory, you can specify a `maxsize` (for InMemoryCache) and it will evict oldest entries once that many items are stored [54]. For more advanced policies or persistence, using Redis or a custom cache is recommended. In practice, LangChain users often employ caching during development to avoid re-running expensive prompts while tweaking chains. It's also useful in testing: you can cache LLM outputs so your unit tests don't call the API each time. LangChain's integration with caching frameworks (like GPTCache or Portkey) means you can also plug those in if needed, but for many cases the built-in caches suffice to dramatically speed up apps (the LangChain author notes caching can be *"0 to 100 (ms)"* kind of improvement, making LLM apps feel snappier) [84].

LlamaIndex (GPT Index): LlamaIndex focuses on connecting LLMs with external data (documents, databases) and had its own mechanisms to cache LLM calls, especially those involved in index construction or querying. In earlier versions, one could enable caching in the `LLMPredictor` so that identical questions to the index wouldn't trigger duplicate LLM lookups. However, as of late 2023, LlamaIndex's maintainers shifted away from a custom cache and now recommend using external caching solutions. The **Portkey** integration is one such solution: Portkey is a service (and also a library) that can act as a caching and routing layer for LLMs. LlamaIndex's `Portkey` class can be used to wrap your LLM – you provide a mode (like single or multi-user), an API key or base URL for Portkey, and it takes over caching of prompts [85] [86]. Under the hood, Portkey can do things like record each prompt/response, handle duplicate requests, and even route to different model variants, but in the context of caching it ensures repeated calls with the same input get the stored answer quickly. There's an open-source server called **Rubeus** which implements Portkey's API, allowing you to self-host the caching layer [87] [88]. The Portkey approach is a bit different in that it can combine caching with other features (like monitoring, as noted – Portkey provides a dashboard of prompts). For those who prefer purely open-source and local caching, LlamaIndex documentation suggests using GPTCache in tandem [89]. In fact, GPTCache has an adapter for LlamaIndex. For example, when building a `KnowledgeGraphIndex` or `VectorStoreIndex`, you can configure GPTCache so that any call to the LLM (for generating node text or answering a query) first checks the cache. If you ask a question that's been asked before, LlamaIndex will fetch the cached answer almost instantly, instead of recomputing it. This can be a huge win in a Q&A system where users might ask similar questions, or when running evaluations. One user noted that with caching enabled, they could iterate on their index prompts faster without incurring costs each time – essentially **development-time caching** to save on API calls. It's worth mentioning that caching in data-index scenarios might need careful invalidation: if the underlying documents update, you might need to clear related cache entries to avoid serving outdated answers.

GPTCache: GPTCache is an open-source library dedicated to prompt caching, created by Zilliz (who are known for the Milvus vector database). It's designed to be a **one-stop solution** supporting exact, semantic, and even more advanced caching strategies. The architecture of GPTCache is modular: you choose an embedding function (OpenAI, Hugging Face, etc.), a storage backend for cache entries (it supports SQLite, MySQL, Postgres for metadata), and a vector store for similarity search (FAISS, Milvus, etc.) [61] [62]. GPTCache can be as simple as in-memory or as distributed as you need. By default, if you call `cache.init()` with no arguments, it sets up a basic local cache. With minimal setup, you can direct your OpenAI API calls through GPTCache. For example, it provides a drop-in replacement for `openai.ChatCompletion.create` – you import `gptcache.adapter.openai` and use that, and it will handle caching behind the scenes [90] [91]. One of GPTCache's strengths is **semantic caching** configuration: you can plug in different embedding models and even different similarity evaluation metrics (vector distance, cosine similarity, or even a custom function) [92] [93]. It also supports **hierarchical cache chains**: you can configure a `Cache` object with a `next_cache` – for instance, first an exact match layer, then a semantic layer [94] [95]. This means it will try the cheap exact lookup, and only if that fails, do the embedding search. In terms of eviction and limits, GPTCache allows setting a max size for the cache and an eviction policy; currently LRU and FIFO are supported, and it mentions possibly adding time-based eviction in the future [56] [55]. Developers have reported substantial improvements using GPTCache. According to its documentation, a high cache hit rate can reduce average **latency by up to 100×** in heavy scenarios [96]. This is plausible when, say, a response that normally takes 2 seconds from the API can be fetched from a local store in 20 milliseconds. Cost savings are of course a big motivation – GPTCache's README highlights that by cutting down repeated calls, you reduce token usage and avoid hitting rate limits [97] [98]. An example given is that GPTCache can make a demo app feel *instantaneous* after the first query, whereas without caching each query would incur the full model latency. GPTCache is also developer-friendly in that it can simulate an LLM for testing: you could "prime" the cache with some inputs and outputs (either real or mock), and then during development your app can run against the cache without even calling the real API – useful for offline testing or integrating into CI pipelines. It's integrated with LangChain and LlamaIndex (as noted above), making it relatively straightforward to drop into existing projects.

Other Frameworks and Tools: Beyond the above, the ecosystem has embraced prompt caching in various ways. **OpenAI's own API** now implicitly caches prompts on their side (with no action needed by the user) [29], but they also allow an optional `prompt_cache_key` parameter to customize the hashing/routing for better hit rates in certain cases [36]. **Anthropic's Claude API** requires developers to opt-in via a beta header and then use the `cache_control` fields in the prompt to mark cacheable content [21]. This gives more manual control – for instance, you can decide that the first system message and the first few user messages are worth caching in a support chatbot. Anthropic even allows caching *tools* and function definitions similarly [40] [41]. **Google's Vertex AI (PaLM)** has a "Context Cache" for their Gemini models, where you explicitly create a cached context and then attach it to model calls [66] [67]. This is a slightly different workflow – you manage cached content IDs – but conceptually the same prefix caching. On the enterprise side, **Amazon Bedrock** supports prompt caching for many foundation models it hosts. As described, Bedrock's approach (especially with models like Claude on Bedrock) uses ephemeral 5-minute caches and up to 4 checkpoint segments [50] [44]. Amazon's documentation and blogs emphasize how this can yield **85% faster responses** and **90% cost reduction** for the cached portions [35] [99]. They encourage structuring prompts to maximize reusability – e.g., keeping a large document context static while users ask multiple questions about it [100] [101]. Meanwhile, community tools like **Humanloop** (which provides an LLM orchestration platform) describe prompt caching in their guides as a best practice for production apps [102]. Even **OpenRouter** (an API that proxies to multiple LLMs) advertises caching to reduce costs for repeated calls [103]. In summary, virtually all modern LLM frameworks and APIs recognize prompt caching as a vital optimization. Whether through built-in features or integrations with libraries like GPTCache, developers have many options to implement caching. The result is more efficient applications: faster responses for the user, lower bills for the developer, and the ability to scale LLM systems to larger workloads by not re-doing the same work over and over. As LLM usage grows, we can expect caching layers to become even more intelligent – perhaps adaptive semantic caching, cooperative caching across different apps, or new frameworks that analyze prompt patterns to auto-cache effectively. But even today, prompt caching stands out as a **practical technique** that every GPT-based application can leverage for better performance and throughput [104] [105].

---

[1] [2] [25] [26] [60] [77] [80] [84] Cache Usage in LLMs: LangChain Cache and OpenAI Prompt Caching | Pedro Medinilla Bohorquez - Pedromebo
https://www.pedromebo.com/blog/en-llm-prompt-caching

[3] [4] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [59] [81] [82] [83] How to Save Costs and Improve Latency in LLMs: Semantic Cache with LangChain and Redis | Pedro Medinilla Bohorquez - Pedromebo
https://www.pedromebo.com/blog/en-llm-semantic-cache

[5] [6] [54] [78] [79] Caches | LangChain Reference
https://reference.langchain.com/python/langchain_core/caches/

[17] [18] [19] [20] [42] [43] [45] [46] [47] [48] [52] [53] [70] [71] [72] [73] [74] Prompt Caching: Complete Guide | Product Builder | Product Builder
https://www.productbuilder.net/learn/prompt-caching

[21] [22] [40] [41] [49] [66] [67] [69] [75] [76] Prompt Caching with OpenAI, Anthropic, and Google Models
https://www.prompthub.us/blog/prompt-caching-with-openai-anthropic-and-google-models

[23] [24] [55] [56] [61] [62] [92] [93] [94] [95] [96] GPTCache Quick Start — GPTCache
https://gptcache.readthedocs.io/en/latest/usage.html

[27] [28] [29] [30] [36] [37] [51] [65] Prompt caching | OpenAI API
https://platform.openai.com/docs/guides/prompt-caching

[31] [32] [33] [34] Slash Your AI Costs by 80%: The Complete Guide to Prompt Caching | by Ashish Kumar Singh | AI Update in Your Pocket | Medium
https://medium.com/artificial-intelligence-update-in-your-pocket/slash-your-ai-costs-by-80-the-complete-guide-to-prompt-caching-e9e09acf02b2

[35] [38] [39] [44] [50] [63] [64] [68] [99] [100] [101] Amazon Bedrock Prompt Caching: Saving Time and Money in LLM Applications | Caylent
https://caylent.com/blog/prompt-caching-saving-time-and-money-in-llm-applications

[57] [58] [90] [91] [97] [98] GitHub - zilliztech/GPTCache: Semantic cache for LLMs. Fully integrated with LangChain and llama_index.
https://github.com/zilliztech/GPTCache

[85] [86] [87] [88] [89] What's the recommended way for caching in llama_index? · run-llama llama_index · Discussion #10189 · GitHub
https://github.com/run-llama/llama_index/discussions/10189

[102] Prompt Caching - Humanloop
https://humanloop.com/blog/prompt-caching

[103] Prompt Caching | Reduce AI Model Costs with OpenRouter
https://openrouter.ai/docs/guides/best-practices/prompt-caching

[104] [105] Langchain Prompt Caching | IBM
https://www.ibm.com/think/tutorials/implement-prompt-caching-langchain