

✦ PLAYBOOK · COST OPTIMIZATION

Caching For AI Systems

What to cache in an LLM application — embeddings, retrieval results, responses and provider prompt prefixes — and how to key, scope, expire and measure each cache safely.

FORMAT	LEVEL	READ TIME	UPDATED
Playbook	Intermediate	12 min	Oct 2026

OVERVIEW

What this playbook covers

Caching is one of the most effective ways to cut cost and latency in AI systems, and one of the easiest ways to serve the wrong answer to the wrong person. This playbook covers each cache layer in an LLM application, how to build keys that include every input that affects the result, how to scope caches to users and permissions, when semantic caching is worth its risk, and how to tell whether a cache is actually helping.

KEY TAKEAWAYS

- Cache at several layers: embeddings, retrieval, model responses and provider prompt prefixes each catch different repetition.
- A cache key must include everything that changes the result — model, prompt version, parameters, index version and permission scope.
- Never share cached results across users or tenants unless the result is provably identical for all of them.
- Semantic caching trades correctness risk for hit rate; restrict it to low-risk, stable content with tight thresholds.
- Set TTLs by how fast the underlying truth changes, and invalidate on events where you can.
- Measure hit rate, savings and stale-serve incidents per cache — not just overall cost.

WHO IT'S FOR

- Backend and AI engineers
- Platform and infrastructure teams
- Engineering leads owning AI cost
- Security reviewers

HOW TO USE IT

Use section 01 to decide which caches you need, 02–03 to design keys and scoping, and the checklist in 06 before enabling any cache that returns model output to users.

CONTENTS

01 What to cache
02 Cache keys and invalidation
03 Per-user and permission scoping
04 Semantic caching and its risks
05 TTLs and provider prompt caching
06 Measuring hit rate and readiness

01 / LAYERS

What to cache

An LLM request passes through several expensive steps. Each can be cached independently, with different risks.

01 Embed the query → 02 Retrieve & rerank → 03 Assemble prompt → 04 Call the model → 05 Post-process & return

CACHE	WHAT IT STORES	RISK	TYPICAL WIN
Embedding cache	Vector for a given text and model	Very low — deterministic	Avoids re-embedding unchanged documents and repeated queries
Retrieval cache	Ranked chunk IDs for a query and filters	Medium — stale on index updates	Skips search and rerank for repeated queries
Provider prompt cache	Processed prefix of a long prompt	Very low — managed by the provider	Lower input cost and latency for stable prefixes
Exact response cache	Model output for identical inputs	Medium — scoping and staleness	Skips the model call entirely
Semantic response cache	Output reused for similar inputs	High — wrong-but-similar answers	Higher hit rate on repetitive FAQ traffic

02 / KEYS

Cache keys and invalidation

Most caching bugs are key bugs: a key that omits something that changes the result. Build keys from an explicit list, not from whatever is convenient.

01

Normalize inputs

Trim whitespace, normalize Unicode and casing where it does not change meaning, and sort structured fields before hashing.

02

Version everything

Include model ID, prompt template version, parameters such as temperature, and index or corpus version.

03

Scope first

Start the key with tenant and permission scope so a missing component fails closed rather than leaking.

04

Hash, don't store, raw text

Use a cryptographic hash of the canonical input as the key; store the plain text only where you need it for debugging.

Invalidation strategies

STRATEGY	HOW IT WORKS	USE WHEN
Version bump	New prompt, model or index version changes every key	Deploys and re-indexing
Event-driven purge	Source update or permission change deletes affected entries	Documents and ACLs change unpredictably
TTL expiry	Entries expire after a fixed time	Underlying truth changes gradually
Manual purge	Operator clears a namespace	Incidents and bad cached answers

FIELD NOTE

Version bumps are the safest invalidation

When the prompt, model or index changes, changing the version in the key makes every old entry unreachable at once. No purge job, no race conditions — old entries simply age out.

03 / SECURITY

Per-user and permission scoping

A cached answer built from documents one user can see must never be served to a user who cannot see them.

UNSAFE SHARING
Key is just the question text
Answer cached from a manager's documents
Same question from a junior employee hits
Restricted content leaks via the cache

SCOPED CACHING
Key includes tenant and permission scope
Scope is a hash of the user's effective groups
Users with identical access share entries
Permission changes alter the scope hash

01 **Classify the content**

Public, tenant-wide, group-restricted or personal. Only public and tenant-wide results share broadly.

02 **Compute a scope key**

For restricted content, hash the set of groups the user belongs to that affect retrieval.

03 **Never cache personal data globally**

Answers that include the user's own records are cached per user or not at all.

04 **Re-check on read for sensitive data**

For high-risk content, verify the cited documents are still accessible before returning a cached answer.

04 / SEMANTIC

Semantic caching and its risks

Semantic caches match on meaning rather than exact text. That raises hit rate — and the chance that “similar” is not “same”.

How semantic caches go wrong

- Questions that differ in one decisive word — “cancel” vs “renew”, a different product, a different date — embed close together.
- Answers that depend on the user’s context are reused for someone whose context differs.
- Thresholds tuned on one traffic mix become unsafe when the mix shifts.
- Errors are silent: the user gets a fluent, confident answer to a slightly different question.

Guardrails if you use one

- [ ] Restrict to low-risk, stable domains such as public FAQs and product information
- [ ] Use a strict similarity threshold, tuned on labelled pairs of same and different questions
- [ ] Scope by tenant, permission and language like any other cache
- [ ] Optionally verify a hit with a cheap check model before serving it
- [ ] Exclude queries with entities, numbers or dates that differ from the cached query
- [ ] Log every semantic hit with both queries so mismatches can be reviewed

> A fast wrong answer is still a wrong answer. Cache what you can prove is the same.
>
> SPLENDENSLABS ENGINEERING PRINCIPLE

05 / EXPIRY

TTLs and provider prompt caching

Expire cached data on the timescale of the truth behind it, and structure prompts so provider-side caching can work.

Starting TTLs (tune to your data)

CACHED ITEM	STARTING TTL	INVALIDATE ALSO ON
Document embeddings	Until content changes	Content hash or embedding model change
Query embeddings	Days to weeks	Embedding model change
Retrieval results	Minutes to hours	Index update or permission change
Deterministic task responses	Days	Prompt or model version change
Generative answers on live data	Minutes, or do not cache	Any source update

PROMPT CACHE

Stable content first

Put system instructions, tool definitions and shared documents at the start; variable content last.

PROMPT CACHE

Keep prefixes byte-identical

Timestamps, request IDs or reordered tools early in the prompt break the prefix match.

PROMPT CACHE

Check provider rules

Minimum prefix lengths, cache lifetimes and pricing differ by provider; read cached-token counts in responses to confirm hits.

06 / MEASUREMENT

Measuring hit rate and readiness

A cache that is never hit costs memory and complexity; a cache that serves stale answers costs trust. Measure both.

METRIC	DEFINITION	USE
Hit rate	Hits ÷ lookups, per cache and per feature	Is the cache worth having?

METRIC	DEFINITION	USE
Savings	Tokens, cost and latency avoided by hits	Translate hit rate into value
Cached-token ratio	Cached input tokens ÷ total input tokens	Provider prompt cache effectiveness
Stale or wrong serves	Hits later flagged by feedback or review	Safety of the cache

KEYS

- [ ] Keys include model, prompt, parameters and index versions
- [ ] Tenant and permission scope first in every key
- [ ] Inputs normalized before hashing

FRESHNESS

- [ ] TTL set per cache from the data's change rate
- [ ] Event-driven purge on content and ACL changes
- [ ] Manual purge procedure documented

SAFETY & MEASUREMENT

- [ ] Semantic cache limited to low-risk content, if used
- [ ] Hit rate, savings and stale serves on a dashboard
- [ ] Cache hits marked in traces

✦ GET IN TOUCH

Let's Talk

We design caching layers for RAG and LLM applications that cut cost and latency without leaking data or serving stale answers. Bring us your traffic patterns.

- ✦ Cache layer and key design review
- ✦ Permission-scoped and semantic caching
- ✦ Hit-rate and savings instrumentation

BOOK AN AI STRATEGY SESSION →

EMAIL
hello@splendenslabs.com

WEB
splendenslabs.com

OFFICE
Sona Incubation Foundation, Salem