DESIGNS · NOV 9, 2026

Reference Data Cache

By James Kunley · jameskunley.com/designs/reference-data-cache/

A design for reusing frequently read, rarely changed reference data across operations, with per-process memory backed by an optional shared cache.

#design #caching #distributed-systems

Contents

- 1. Problem Summary
- 2. Acceptance Criteria
- 3. Out of Scope
- 4. Glossary
- 5. Design
  - 5.1 Access Patterns
  - 5.2 Cache Reader
  - 5.3 Cache Owner
  - 5.4 Values, Metadata, and Admission
  - 5.5 Policies and Read Outcomes
  - 5.6 Timeouts, Retries, and Recovery
  - 5.7 Invalidation and Change Visibility
  - 5.8 Keyed Storage and Batch Loading
  - 5.9 Failure Policy and Reporting
  - 5.10 Integration and Verification
- 6. Rejected Alternatives

1. Problem Summary

Reference data supplies shared definitions and lookup values used to interpret, validate, classify, or enrich other data. It is usually read often and changed infrequently. Examples include codes, labels, categories, units, supported values, their relationships, and larger collections of records addressed by identifier.

It is useful across a system:

- **Database queries:** Joins, identifier resolution, filtering, sorting, and grouping.
- **Application logic:** Validation, eligibility, calculations, and rule inputs.
- **APIs:** Response enrichment, supported-value lists, and representation mapping.
- **Consumers:** Common definitions for services, workers, scripts, and command-line tools.
- **User interfaces:** Labels, controls, selection validation, and value interpretation.
- **Background processing:** Message, import, export, and batch enrichment.
- **Search and read models:** Index attributes, projections, and aggregates.
- **Reporting and analytics:** Consistent classification, grouping, and descriptions.

Without reuse, database queries repeatedly join reference tables, servers build the same values in memory for each request, and consumers repeatedly fetch them from endpoints. Each operation then discards its copy.

Database buffering may reduce disk reads, but query execution, transfer, encoding, parsing, and allocation still recur as demand grows. Concurrent requests and application instances can repeat this work simultaneously, multiplying load on the same dependencies.

Some consumers need a complete value or dataset; others need selected records by identifier. Fetching every record wastes memory and source work when only a few are used. Retaining data for every encountered identifier can also grow memory use as the workload touches more records.

Reference data can change. Consumers need those changes to become visible, while acceptable data age and tolerance for outdated values vary by data type.

Dependencies may fail, respond slowly, or never complete while useful values remain in memory. Expiry can then reduce availability, and repeated acquisition or recovery attempts can worsen an outage. Applications need to choose whether to wait or use retained values within an acceptable age, with bounded waiting and recovery work and explicit failure when neither can satisfy the request.

2. Acceptance Criteria

Consumer Capabilities

- Consumers can register data sources, read cached values with optional freshness metadata, and invalidate cached entries. Support local caching with optional caller-supplied shared storage.
- Support loading a complete dataset as one value, including objects, arrays, or scalars, and loading selected records by identifier. Consumers choose the loading pattern for each data type.

- Support individual and batch reads and invalidation by identifier. Report an outcome for every distinct requested identifier, preserve successful outcomes when others fail, and distinguish confirmed absence from acquisition failure.
- Shared storage supports reading, population with expiry, and deletion across different backends, protocols, client libraries, and encodings. Use bulk reads when available; storage without bulk reads remains supported.
- Requesting freshness metadata does not change acquisition behavior. Include freshness, age bounds, and delivery reason for successful values and confirmed absence.
- Reject unknown data types, duplicate registration, and invalid settings explicitly. Registering a source performs no source or network access and leaves no partial registration on failure.
- Allow data-type policies to override client defaults. Later changes to supplied settings or defaults do not alter registered policies; reject settings that would leave dependency waits unbounded.

Reads and Failure Recovery

- Fresh local reads avoid dependency access. Concurrent requests for the same entry reuse acquisition work; different entries have independent coordination.
- Consumers can wait for fresh acquisition, allow eligible stale data when refresh produces no fresh result, or receive eligible stale data immediately while refresh runs in the background. Waiting for fresh acquisition when refresh is due is the default.
- Preserve previously successful values through expiry and refresh, subject to retention limits and invalidation. Stale delivery requires a finite age limit and explicit permission; failures cannot extend that limit or create substitute values.
- Immediate stale delivery occurs before that entry's background refresh code runs. During cooldown, return an eligible fallback or recorded failure without waiting for cooldown to end.
- Shared-storage failure leaves time to attempt the configured source. A shared miss proceeds to source loading without retrying the miss.
- Bound the duration of source loading and shared reads, writes, and deletes, including background population and cleanup. These bounds still hold when dependency timeouts or cancellation fail.
- Limit retries by attempt count, elapsed time, and delay between attempts. After a failed or stale refresh, apply the configured cooldown; a later request can retry after it even if earlier dependency work never settles.
- Reads and invalidation settle within their original budgets while the runtime remains responsive. Timing out, cancelling, or replacing obsolete work does not wait for physical cancellation to complete.
- Late results cannot replace newer local state or trigger obsolete retries or new shared writes. Cancelling one caller does not interrupt other callers or newer work.
- Shared population, cleanup, and reporting failures do not undo successful local delivery. Background work and pending notifications remain bounded.

- Accept valid empty collections, zero, and false as successful values. Reject absent value payloads and malformed cache metadata explicitly; neither represents confirmed record absence. Application-value validation remains caller-owned.
- Bound retained records, active acquisition, and batch size for loading by identifier. Local admission rejection returns immediately without retries or cooldown, and later calls can try again. Evicted records can be loaded again.
- Copies preserve source observation, version information when available, ordering, and delivery bounds. Copying cannot reset age or make upstream stale data fresh; only fresh acquisition counts as recovery or permits shared population.

Invalidation

- Consumers can invalidate a complete cached dataset or selected identifiers. Invalidation clears targeted local values, stale fallbacks, and obsolete work without dependency access; when shared storage is configured, also attempt bounded shared deletion and report its outcome.
- Subsequent reads use the configured source under normal freshness and failure policies. Invalidation requires neither a new source capability nor proof that the value or its observation time changed.
- Temporary bypass of shared storage during invalidation ends after bounded recovery work, subject to retention limits. It cannot become a permanent condition for future reads.
- Other clients continue to follow their freshness and stale-age limits. Immediate cross-client notification and coordinated invalidation remain application responsibilities.

3. Out of Scope

- Arbitrary query/response caching, source writes, automatic query rewriting, and atomic snapshots across multiple keys.
- Enumerating, preloading, or invalidating every key of a keyed type. Callers supply explicit keys; omitted keys never mean “all keys.”
- Cross-process loading locks, event publication/subscriptions, invalidation broadcasts, or globally synchronized freshness.
- Scheduled refresh and prewarming without a request. Request-triggered background refresh is included.
- Byte-based memory budgets and eager expiry cleanup. Keyed entry-count limits, eviction, and bounded cache-owned work remain in scope.
- Durable storage, confirmed shared persistence, and draining background writes on shutdown.
- Provisioning, credentials, pools, hidden transport queues, reconnection, and physical cancellation. The cache still bounds its own I/O waits.
- Application authorization, tenant resolution, and automatic isolation boundaries.

4. Glossary

Data and Interfaces

Term	Meaning
Reference data	Read-mostly definitions or records used to interpret, validate, or enrich other data.
Data type / domain	A registered dataset with one loading pattern and policy. The logical failure field `domain` identifies it.
All / keyed	Load a complete value as one unit / load selected records by identifier.
Key / entry	A record identifier / one cached unit, identified by data type and, in keyed mode, key.
Client / reader / owner	One cache instance / a client with local storage / a client with local and shared storage.
Source / loader	The configured database, endpoint, or other data supplier / application-provided acquisition logic. A source may itself serve cached or replica data.
Tenant / representation	An application isolation boundary / a requested variant of a value. Distinct boundaries or variants need distinct cache identities when they affect results.
Adapter / backend	The component connecting shared-cache operations to a storage service / that storage service.
Array / map / object / scalar	An ordered sequence / a collection of key-associated results / a structured value / a single value. These are logical shapes, not language types.
Batch / bulk read	Multiple keys requested together / acquiring multiple shared entries through one adapter operation.
Identifier / batch normalization	Application-defined canonical identity / converting individual or batch input into a sequence with one occurrence per equal key.
Namespace / encoding / schema	An identity scope preventing collisions / a portable representation / the rules defining valid value structure.
API / HTTP	Application programming interface / Hypertext Transfer Protocol. No particular API syntax or protocol is required.

Values and Freshness

Term	Meaning
Snapshot / envelope	A value with its provenance / the structure carrying that value and its metadata.
Metadata / provenance	Information describing a result / its source observation, version, and inherited delivery bounds.
Candidate / expiry	A snapshot offered for validation and selection / the end of normal TTL reuse, which need not delete a retained value.
Upstream / replica	A dependency supplying a result to this cache / a copy of source data that may lag its origin.
Source version / validation time	An optional revision identifier / a conservative timestamp for the source observation behind a snapshot.
Hit / miss / confirmed absence	An entry exists at a cache layer / no entry exists there / the configured source reports no record. A hit still needs validity and freshness checks.
Fresh / stale / hard-expired	Eligible for normal reuse / due for refresh and requiring stale permission / beyond the latest permitted delivery time.
Non-fresh outcome	A failed refresh or an upstream-stale result. Cancellation, supersession, and rejected admission are separate control or admission outcomes.
TTL / hard age limit	Time to live: the normal reuse or storage duration / the maximum permitted age of a returned value.
Blocking read	Wait for fresh acquisition when refresh is due, within the request budget. It does not guarantee authoritative source truth.
Fallback / stale-if-error	A retained value eligible for delivery / using it after refresh yields no fresh result and failure policy permits.
Stale-while-revalidate	Return eligible stale data immediately, then refresh in the background.
Negative cache	A time-limited record of confirmed absence, distinct from a cache miss or failure.
Promotion / population	Copying a shared snapshot into local storage / writing a freshly acquired snapshot to shared storage.
UTC epoch time / monotonic clock	Portable time measured from 1970-01-01 00:00:00 UTC / an elapsed-time clock unaffected by wall-clock corrections. UTC means Coordinated Universal Time.

Term	Meaning
Clock skew allowance	The declared bound on differences between participating clocks, subtracted conservatively from incoming age and freshness limits.

Acquisition and Coordination

Term	Meaning
Acquisition / refresh / attempt	Obtaining a candidate / a coordinated acquisition cycle, including retries / one loader or adapter invocation.
Single-flight / waiter / subscription	Sharing one current refresh for an entry / a caller waiting for it / that caller's interest in a particular key's shared work.
Atomic transition / cache coordination	A state change observed as one indivisible decision by competing calls / the mechanisms enforcing those decisions and sharing current work.
Runtime / scheduling / execution capacity	The execution environment / when work is allowed to run / the resources needed to make progress and enforce deadlines.
Preemption	Interrupting work that is already executing; logical timeout completion does not guarantee this capability.
Budget / deadline / timeout	The total allowed duration / its cutoff / completion of cache-owned waiting when the cutoff is reached.
Loader reserve / watchdog	Time preserved for source loading after shared access / an independent deadline check covering the whole logical operation.
Generation / refresh identity / token	Markers identifying invalidation state / a particular acquisition / whether an attempt or key result can still be accepted.
Supersession / control outcome	Ending obsolete logical work / a caller cancellation or supersession rather than an acquisition failure.
Logical completion / physical cancellation	Releasing cache-owned coordination and completing callers / actually stopping dependency work. One does not imply the other.
Cancellation signal	A request to stop an attempt; receiving or sending it does not prove dependency work has ended.
Background work / cleanup	Work that does not delay successful delivery / best-effort removal of invalid or obsolete shared entries.

Term	Meaning
Current / pending candidate	Work or data whose identity is still accepted / a replacement snapshot waiting for its allowed write opportunity and original deadline.

Failures, Limits, and Integration

Term	Meaning
Failure code / stage / cause	A normalized failure category / where it occurred / the original local dependency error.
Classifier / comparator	Application policy deciding retry and stale eligibility / application logic ordering comparable source versions.
Backoff / jitter	Delay between retries / random variation of that delay.
Cooldown / retryAfter / recovery	Suppression of new refreshes after a non-fresh outcome / its monotonic end time / acceptance of a fresh acquisition after degraded operation.
Admission limit / dependency capacity	The cache declines new logical work before dispatch / a dependency reports resource saturation. Only the latter follows ordinary retry policy.
Retention / eviction / least recently used	Keeping local entry state / removing it to enforce a limit / choosing the entry with the oldest last access.
Logical slot / physical permit / pool	Capacity for cache-owned work / permission held for actual dependency work / a bounded collection of reusable dependency resources.
I/O / transport	Input/output with a dependency, including database or network access / the connection and request mechanisms performing it.
Reporting sink / coalescing	An application-provided destination for operational notifications / replacing redundant pending notifications with bounded summaries.
Invalidation / shared bypass	Discarding held state and obsolete work / temporarily acquiring without reading shared storage. Invalidation does not prove source data changed.
Authoritative validation / read-after-write	Confirming truth at the application’s authoritative source / the application’s required visibility of completed writes to later reads.
Eventing / stampede	Application-owned change notification / correlated client reads producing a burst of loading work.

Term	Meaning
Circuit breaker / hysteresis	A dependency-protection policy that changes request eligibility after failures / separate thresholds or delays that prevent rapid policy-state changes.
Conditional write / prewarming	A write accepted only when a backend condition holds, such as an expected version / loading data before requests need it.

5. Design

Behavior, metadata meanings, policy defaults, and failure guarantees define the implementation contract.

Names, language types, error representations, concurrency mechanisms, and encodings may follow the target language and organization's conventions while preserving these contracts. Field labels and outcome names below describe logical information; they do not require a particular API or wire format.

5.1 Access Patterns

Use one library with two registration modes. Both support objects, arrays, and scalar values. `all` loads, replaces, and invalidates the complete value as one unit; `keyed` does so per requested identifier.

Registration completes locally without external I/O. Implementations choose their calling and concurrency models; reads, invalidation, and background work must still meet the deadlines and delivery ordering in §5.6.

Choose the mode once per data type; freshness, retry, failure, and shared-storage settings remain type-level policies. Values, cooldown, coordination, and invalidation belong to individual entries. The reader and owner flows below apply to either mode; §5.8 covers keyed batching and capacity.

Operation	`all`	`keyed`
Read	Return the complete value.	Read one identifier or a batch; return a map of per-key outcomes.
Read with metadata	Return the complete value with metadata.	Add metadata to found and not-found outcomes; keep acquisition errors distinct.
Invalidate	Clear the complete entry.	Clear one identifier or a batch; return per-key invalidation outcomes.
Source loading	Receive operational context and return the complete value.	Receive an array of keys and operational context; return an outcome for each key.

A keyed call always returns a map, including for a single key. Each key reports a found value, confirmed absence, or an acquisition error. Successful keys remain available when others fail.

Identity

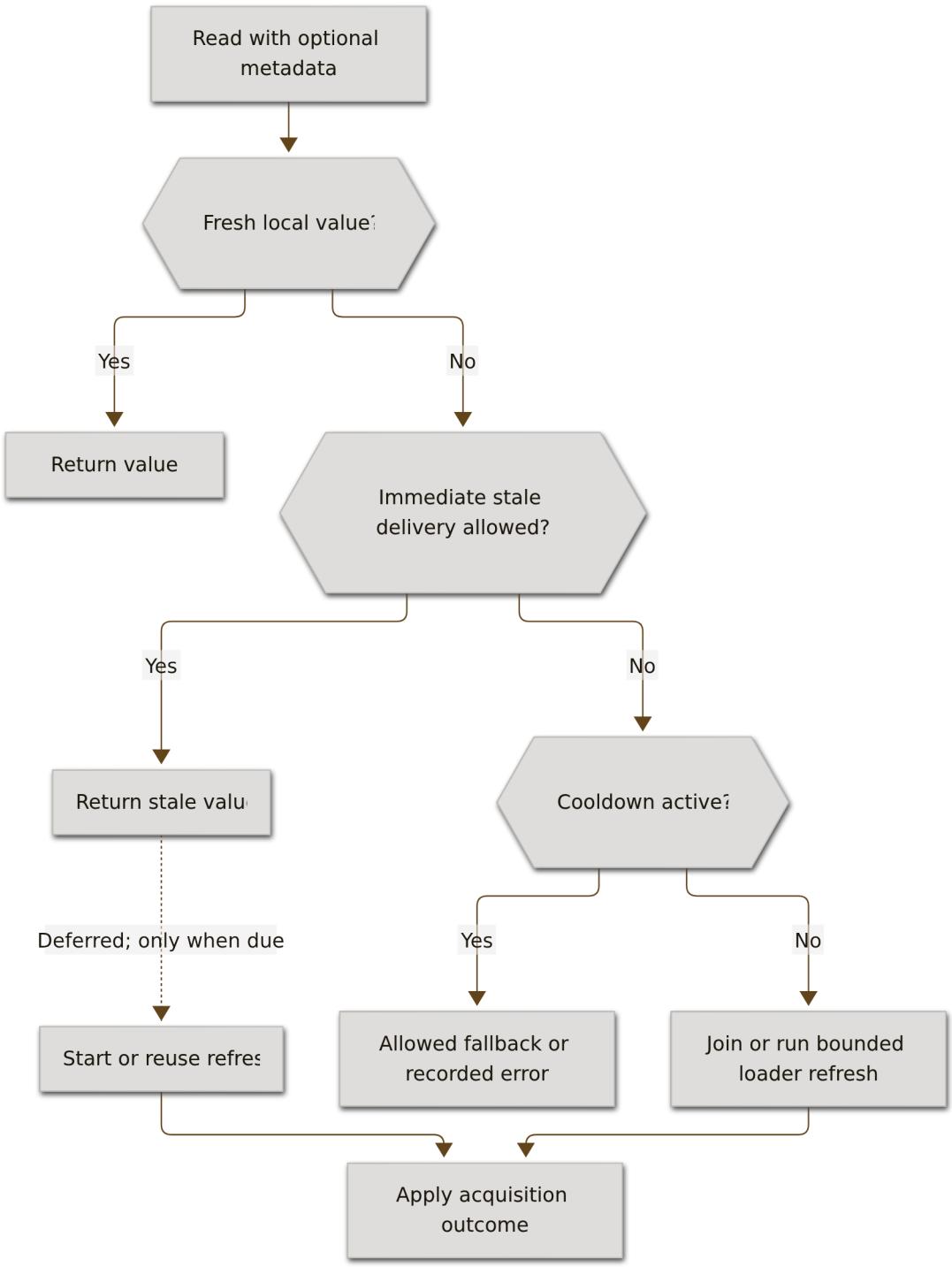
- Names and keys are exact and case-sensitive. Keys are nonempty strings or integers that round-trip without loss through the chosen local and shared representations; numeric `1` and string `"1"` are distinct. Applications perform any required identifier normalization before lookup.
- Owners require an explicit shared namespace. Encode namespace, type, mode, and key without collisions; clients sharing an identity must agree on value and metadata meanings, schema, encodings, and time units.
- Callers include tenant, representation, or version in identity whenever these change the value; a type/key must not depend on unrepresented request context.

Input Validation

- Reject unknown types, requests incompatible with the registered loading pattern, invalid keys, and oversized batches before I/O.
- Deduplicate keys in first-occurrence order.
- For a valid keyed type, an empty array returns an empty map without I/O.
- Map iteration follows that normalized order, independent of completion order.
- Acquisition errors are per-key outcomes; invalid call arguments reject the whole call.

5.2 Cache Reader

Create a reader once and reuse it across requests. Each registered loader supplies data from a database, endpoint, or another configured source. Apply this flow independently to each requested entry.



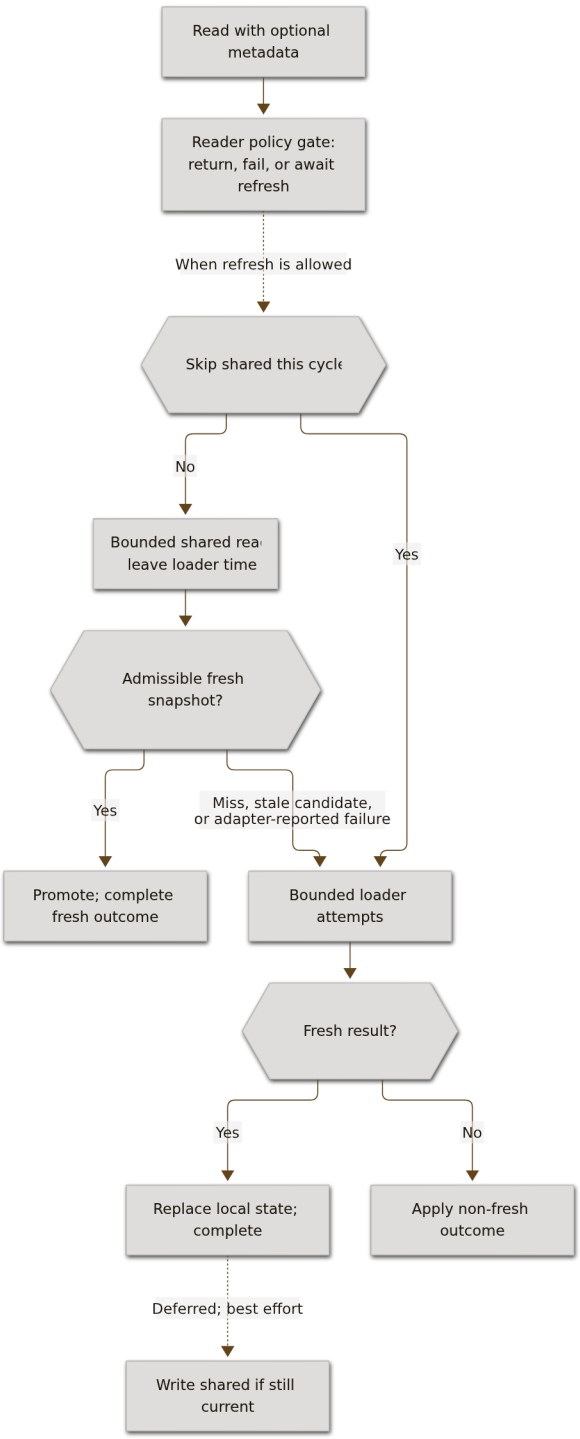
- The client records shared background work before returning and dispatches its body only after delivering the immediate result. Concurrent reads reuse that work; dispatch uses the target runtime's scheduling model.
- No read sleeps through cooldown.

Mixed Keyed Reads

- A keyed call returns one map. If some keys need acquisition, the map waits for those keys within the call's budget; immediate entries do not wait for their own refresh.
- Defer their background work until map delivery, while acquisition for the other keys proceeds.
- Recheck age and upstream delivery bounds before returning the map; an ineligible result becomes a per-key failure without restarting the budget.

5.3 Cache Owner

An owner applies the reader's delivery policy, then checks shared storage when refresh is needed.



Shared population never delays successful delivery. Every promotion, retry, and write must pass the current-work checks in §5.6.

Shared Reads

- Keyed owners use the adapter's bulk-read capability when available; adapters can use native bulk reads or pipelining.

- Bulk reads accept encoded shared-entry identities and operational context, returning an outcome for each identity: a hit carrying its stored snapshot, an explicit miss, or a dependency error.
- Without bulk-read support, use bounded per-entry reads with the same outcomes. Single reads remain the minimum adapter contract so simple backends need no bulk implementation.
- Bulk results must explicitly cover every requested key. Apply §5.8's batch validation and retry rules: preserve valid hits/misses, treat omitted outcomes as invalid data, and retry only failed keys within `sharedRetry.read`.
- A whole-call failure affects its unresolved keys.
- Bulk attempts obey the same per-entry deadlines, shared-stage cutoff, loader reserve, and late-result checks as single reads; changing call shape never adds attempts or time.

5.4 Values, Metadata, and Admission

Entry Contracts

These contracts describe one entry, independent of language or backend. A question mark denotes optional information; an absent hard age limit may use any explicit representation that preserves its meaning.

Each keyed loader outcome is a source result, a cached result, or a dependency error. Successful outcomes wrap a record or explicit absence in the same provenance envelope:

Contract	Fields
Loader context	`{ attemptStartedAt, deadline, signal }`: UTC attempt start, monotonic logical deadline, and per-attempt cancellation signal.
Source result	`{ kind: source, value, observedAt?, sourceVersion? }`.
Cached result	`{ kind: cached, result }`, carrying the upstream read result below.
Stored snapshot	`{ value, validatedAt, sourceVersion?, upstream? }`. Optional `upstream` contains `{ stale, freshUntil, serveUntil, reason }`.
Read result	`{ value, validatedAt, freshness, freshUntil, serveUntil, sourceVersion?, reason }`. `freshness` is `fresh` or `stale`; an absent `serveUntil` limit means no hard cap.
Failure	`{ domain, key?, stage, code, attempts, cause?, operationId }`. Public delivery reasons contain only `{ code, stage }`; original exceptions stay local.

`upstream.stale` is boolean; `reason` is null or `{ code, stage }`. Portable freshness/age deadlines are finite UTC times, except an explicitly absent `serveUntil` limit. Envelope validation checks these fields, not the value's application schema.

- Loaders and adapters report only a dependency failure code (§5.9) and optional cause.
- The cache builds the full `Failure`, adding the type, key, stage, attempt count, and operation identity; supplied code cannot choose cache-control outcomes such as `admission-limit`.

Values and Absence

- An all-mode value or a found record may be empty, zero, or false; missing or null payloads are invalid.
- Keyed envelopes carry an explicit presence flag and, when present, the record value. Never infer absence from an omitted key or a failed request.
- Validate this presence wrapper as part of the envelope; application validation still owns the value schema.
- Shared storage keeps the wrapper; public reads expose §5.1's outcomes. Reads requesting metadata include it beside each found/not-found status; errors carry the normalized failure instead.

Provenance

- For source results, set `validatedAt` to the attempt start or an earlier supplied `observedAt`. Do not use completion time to make a slow read appear younger.
- This records an observation of the configured source, not proof of the newest global revision. Replicas remain supported; a known older source observation must be reported accurately, and applications needing origin freshness must obtain that guarantee from their loader.
- Cached results preserve validation time and version. Set `upstream.stale` from the returned freshness and copy its deadlines/reason; direct source snapshots omit `upstream`.
- Endpoint success does not justify a new timestamp.
- Shared adapters store the envelope unchanged. Implementations document whether local values are shared, copied, or immutable. Callers must prevent mutation that could alter other reads and use adapter-compatible encodings.

Clocks and Admission

- All portable times are finite UTC epoch milliseconds. Local waits, TTL intervals, and cooldown use a monotonic clock.
- Client-stamped local observations need no cross-host allowance; supplied observation times and cached/shared metadata require an explicit finite `clockSkewAllowanceMs` (zero is allowed).
- Reject wrong field types, non-finite times, validation times too far in the future, and freshness deadlines later than hard deadlines.
- Past deadlines indicate expiry, not corruption.

- Subtract the skew allowance when converting incoming age limits and deadlines to remaining local time. Portable age guarantees depend on the declared clock bound.

Shared result	Action
Valid, not upstream-stale, within inherited freshness/age bounds, and not known older	Promote as fresh.
Valid stale present data within the age allowance	Keep as a fallback candidate and try the loader.
Hard-expired, missing, or adapter-reported error	Try the loader; never serve the hard-expired candidate.
Invalid envelope	Report, attempt bounded cleanup, and try the loader.

Candidate Ordering

- When both source versions are comparable, use the optional data-type comparator. Otherwise prefer the later observation, retaining the existing candidate on a tie.
- Observation time does not order data revisions. An older shared candidate cannot replace a newer retained one; try the loader instead.
- Comparator errors fail as policy errors. The same rules select among offered stale candidates.
- Backend storage TTL does not override these bounds. Expiry of an earlier local copy alone does not disqualify a shared snapshot; promotion can start a local interval within the preserved restrictions.

5.5 Policies and Read Outcomes

An all-mode source-only reader needs effective `ttlMs` and `refreshTimeoutMs`; keyed types also require the limits below. Type settings override client defaults and apply uniformly to their keys. Shared-storage settings fall back to adapter defaults. Copy policies at registration.

Freshness

Setting	Rule or default
`ttlMs`	Required finite, nonnegative local reuse interval.
`readMode`	`blocking` by default; also `stale-if-error` and `stale-while-revalidate`.
`maxValueAgeMs`	Finite positive bound required for stale serving; optional for blocking.

Setting	Rule or default
`clockSkewAllowanceMs`	Implicit zero only for client-stamped local observations; explicit before portable metadata is accepted.
`sharedPolicy.ttlMs`	Adapter-supported storage lifetime, at least the applicable local TTL.

Timeouts and Recovery

Setting	Rule or default
`refreshTimeoutMs`	Required finite positive read/refresh budget.
`invalidateTimeoutMs`	Owner deletion budget; defaults to the refresh budget. Reader invalidation has no I/O.
`loaderReserveMs`	Required owner reserve, positive and no greater than the refresh budget.
`retryCooldownMs`	Defaults to the refresh budget; finite nonnegative override, with zero disabling suppression.
`loaderRetry`, `sharedRetry.read/write/delete`	Default one attempt. Effective attempt/total timeouts are finite; extra attempts require finite base/max backoff delays.
`reportTimeoutMs`	Notification lifetime, defaulting to the refresh budget.
Failure classification and source-version comparison	Optional local callbacks; no I/O or deferred decisions.

Keyed Limits

Setting	Rule or default
`maxEntries` (keyed)	Required positive integer; bounds retained entry state using least-recently-used eviction.
`maxBatchSize` (keyed)	Required positive integer; maximum distinct keys in one public call or loader invocation.
`maxActiveKeys` (keyed)	Required positive integer, at least `maxBatchSize`; bounds current entry refreshes.
`notFoundTtlMs` (keyed)	Defaults to zero (no negative reuse); finite nonnegative TTL for confirmed absence.

Timeout Defaults

- A reader's loader attempt and total limits default to its refresh budget.
- For owners, the loader attempt defaults to the reserve and its total to the refresh budget; shared-read limits default to the budget minus reserve.
- Zero shared allowance bypasses that stage. Shared writes default to the refresh budget; deletes and cleanup default to the owner invalidation budget.
- The loader attempt must fit the reserve.

Registration and Budget Rules

- Overrides replace a complete retry-policy object; omitted fields use these derived defaults.
- Attempt counts are positive integers; one disables retries.
- Reject missing required settings, infinite budgets, negative/reversed backoff limits, and unsupported timer durations.
- Explicit timeouts remain clipped by their operation and parent limits.
- Every job's budget starts at creation, including deferred/pending time. Values are workload settings, not universal latency targets.

Value Lifetimes

For present values, calculate remaining lifetimes at population:

```
hard remaining = min(
    validatedAt + maxValueAgeMs - UTC now - skew allowance, if configured,
    upstream serveUntil - UTC now - skew allowance, if present)

fresh remaining = min(
    ttlMs,
    upstream freshUntil - UTC now - skew allowance, if present,
    hard remaining)
```

- Keep monotonic deadlines locally and expose conservative UTC bounds in results.
- Upstream-stale data cannot become fresh through promotion. Without a hard bound, blocking mode provides TTL reuse without an end-to-end age guarantee.
- Check the hard limit when selecting a response, including after waiting for refresh.
- Zero TTL disables normal local reuse: a newly observed source result may satisfy its waiting read once, within any hard age limit. Acquisition completion never recursively refreshes it.
- For example, data observed and populated at 12:00 with a ten-minute TTL, one-hour age limit, and zero skew becomes due at 12:10 and unusable at 13:00. Failures move only the next-attempt time.

Read Outcomes

Current acquisition result	State change
Fresh	Replace eligible local state, clear failure/cooldown, and permit shared population.
Upstream stale	Select an eligible candidate, record `upstream-stale`, and start cooldown; no shared write or recovery claim.
Failure	Retain the previous value, record the failure, and start cooldown.

- Blocking waits and rejects non-fresh results.
- Stale-if-error waits, then may use a candidate when classification permits.
- Stale-while-revalidate returns an eligible candidate first; the later result governs subsequent reads.
- No candidate, denied permission, or a hard age limit means explicit failure. A denied fallback failure disables further stale delivery until fresh recovery.
- Supersession terminates old work without updating newer state or cooldown. Caller cancellation ends only its waiter. These are control outcomes, not acquisition results.
- The separate stale modes resemble stale-while-revalidate and stale-if-error, without requiring HTTP.

5.6 Timeouts, Retries, and Recovery

- The cache owns current tasks, retry timers, pending candidates, and waiters.
- Loaders/adapters own physical requests, pools, cancellation, and transport repair.

Operation Deadlines

- Before every loader or adapter invocation, arm a client-owned timeout, including background writes and cleanup deletes.
- Use the shortest remaining attempt, operation, and parent budget.
- A separate watchdog covers the whole refresh or owner invalidation.
- Joining callers keep their original deadlines; nothing restarts a budget.

Completion

- One completion path closes the attempt token, clears timers and logical ownership, and settles waiters exactly once.
- Timeout does not await cancellation, old I/O, reporting, or cleanup. Request cancellation separately and ignore late outcomes.
- Late completion handlers must not retain active locks, waiter lists, or snapshot payloads.

- End owner shared reads by the earlier operation cutoff or `refresh deadline - loaderReserveMs`. If no shared time remains, go directly to the loader.
- Adapter saturation is a classified failure, not a cache-owned capacity lock.

Retries and Cooldown

`maxAttempts` includes the initial call. Before retry `n`, starting at one, use full jitter:

```
delay = random(0, min(maxDelayMs, baseDelayMs * 2^(n - 1)))
```

- Allow other work to run even when retry delay is zero; check monotonic deadlines before dispatch.
- Misses do not retry, and malformed shared data goes to cleanup/loader.
- Bound dependency-client retries within the same effective limits; otherwise attempts multiply across layers (AWS retry guidance).
- Every current non-fresh acquisition sets `retryAfter = completion + retryCooldownMs`.
- During cooldown, return allowed fallback or the recorded error immediately.
- The first later request can start a new bounded refresh even if the old dependency operation never completes. There are no request-independent recovery loops.
- A fresh result clears cooldown; a stale upstream result does not.

Current Work and Cancellation

- Decisions that change registration, admission, invalidation state, candidate selection, or completion must be atomic relative to concurrent calls.
- Do not hold cache coordination while waiting, performing I/O, or running supplied code. Recheck current-work identity after supplied code returns and before committing state.
- Each entry has one current refresh, one logical writer with one replaceable latest candidate, and at most one cleanup job. Keyed entries also share the type-level work limits in §5.8.
- Generation and refresh identity checks stop obsolete population, retries, and writes. A newer accepted snapshot supersedes older write identities.
- All jobs release their cache-owned slots at deadline/supersession; pending candidates keep their original budgets.
- Already-issued remote effects cannot be retracted.
- An already-aborted read rejects before scheduling. Later cancellation removes only that waiter; recursion and cross-entry loader cycles still obey the original watchdog.
- External background refresh, cancellation, and reporting bodies must not run ahead of an immediate response.

- Supplied code must not block or exhaust the execution capacity needed to complete cache operations and enforce deadlines. Timeout bounds assume that capacity remains available; they do not promise exact scheduling or preemption of arbitrary supplied code.

Physical Resource Limits

Timeout releases cache coordination, not physical resources. When loader calls never finish, repeated refresh cycles accumulate stalled physical calls for as long as demand and the outage continue.

For equally timed, independent single-entry cycles, the rough rate is `actual attempts ÷ (actual cycle duration + cooldown) × active entries` calls per unit time. Keyed batching changes calls per cycle.

Neither logical work limits nor this rate are a fixed ceiling on outstanding physical calls; dependency limits or effective cancellation must stop accumulation.

The configured attempt count caps one cycle; timeouts are not minimum cycle durations. Adapters must limit their own queues/resources and recover transports; the cache cannot contain arbitrary supplied code.

5.7 Invalidation and Change Visibility

Local and Shared Invalidation

Invalidation clears the targeted entries this client holds; it does not prove what a source will return. Apply the following independently to each entry:

1. Advance the generation, discard local value/fallback/cooldown, and supersede old work. Settle affected waiters with `superseded` without waiting for I/O.
2. A reader completes locally. Its next read uses its normal loader and admission rules.
3. An owner also deletes the shared entry under its original timeout/retry budget. While owner deletions are pending, reads bypass shared storage.
4. Each owner deletion finalizes once, advances the generation again, clears refills and intermediate failure/cooldown state, and supersedes affected work. Report its own success or failure. After the last pending deletion, mark one post-invalidation refresh to bypass shared storage.

- A keyed invalidation returns `{ status: "invalidated" }` or `{ status: "error", error }` per key; a delete error still leaves the local entry cleared. Invalidating an unknown but valid key is allowed.
- Overlapping calls share a bounded per-entry delete coordinator: each caller keeps its deadline, the entry remains bypassed while a current delete is pending, and repeated calls cannot accumulate an unbounded queue.

Shared Bypass

- Consume that one-cycle bypass when the next current refresh starts. It invokes the configured loader under normal rules; completion or failure ends the preference.
- A later invalidation creates a new generation and its own marker. No authority-proof flag or timestamp threshold can remain latched after this cycle.
- A cached endpoint or replica result remains acceptable under normal policy, even with a validation time older than invalidation. The discarded local copy cannot be used as fallback, but newly acquired data may have the same value.
- Failed shared deletion is reported; after the bounded bypass, an old shared entry may be encountered until its storage expiry.
- Strong read-after-write requirements belong in the application's loader/version contract.

Cross-Client Visibility

Other clients discover changes on subsequent reads under their TTL, refresh, and stale-age limits. Expiry makes acquisition due; it does not synchronously update every client. Stale delivery stays within the hard age bound.

- Applications needing immediate notification publish their own change events and invalidate affected entries on each client. They own source-update ordering, delivery, duplicates, authorization, and any claim that every client has changed.
- Invalidation is lazy: loader spikes require correlated reads afterward, not merely receiving events.
- Applications can coordinate or spread refresh work; delaying invalidation itself weakens an immediate-invalidation requirement.
- The cache supplies no event bus, cross-process loading lock, or global freshness barrier.
- Already-dispatched writes/deletes can still alter shared storage after local invalidation. Strong remote ordering requires backend conditions or application coordination.

5.8 Keyed Storage and Batch Loading

Loading

1. Normalize the keys, return fresh local hits, join existing refreshes by key, and admit new refreshes within `maxActiveKeys`. Apply the common read policy per entry.
2. Owners use the bounded shared-read contract in §5.3; only unresolved keys reach the batch loader.
3. A reader sends newly admitted cold keys from the call in one loader batch.
4. An owner first finishes their bounded shared-read stage, then batches the remaining misses.
5. Dispatch new keys without waiting for joined refreshes to finish or adding a batching delay; the caller still awaits the joined keys in its result map.

For example, overlapping requests for `[a, b]` and `[b, c]` reuse `b`'s current refresh; only `c` needs new work in the second request.

Partial Failures

- A loader always receives an array, even when only one key remains.
- It must return one outcome for every requested key.
- Omitted or malformed outcomes are `invalid-data` for those keys; unexpected keys are reported and never cached.
- A batch call that fails as a whole fails its unresolved keys.
- A malformed whole response fails all unresolved keys.
- Accept valid per-key results independently.
- Retry only unresolved, retryable keys under their original budgets and attempt counts; never reload successful keys merely because another failed.
- A confirmed absence is a successful result, not a retryable miss. Multiple found records do not imply an atomic source snapshot.

Batch Deadlines and Cancellation

- A batch transport attempt uses the earliest remaining member deadline and applicable attempt/operation limit.
- Expiry or invalidation closes only the affected key's token, not other keys' refreshes; ignore that key's late outcome. An already-dispatched batch may continue for remaining keys within their original budgets.
- Ending one token need not abort the shared transport; a transport timeout still closes that attempt for all unresolved members.
- The result map settles when its keys settle or the call's original deadline expires, preserving eligible completed results and marking unresolved keys with errors.
- Caller cancellation rejects that call and detaches its subscriptions without cancelling other waiters.

Admission Limits

- Callers split sets larger than `maxBatchSize` explicitly.
- When `maxActiveKeys` is reached, a new miss gets an immediate `admission-limit` outcome or permitted stale fallback; fresh hits and existing refresh joiners still work.
- No refresh was admitted: do not queue, retry, record an acquisition failure, or start/extend cooldown. Preserve existing entry state, including stale eligibility; the next call can try admission again.
- Timeout or supersession releases logical slots even if physical I/O continues.
- The same count bounds current shared writers, deletes, and cleanup jobs separately. Keep at most one replaceable pending write per retained entry, with its original deadline.

- At these local limits, skip/report best-effort side effects as `admission-limit`; do not queue new deletion/cleanup jobs. Rejected shared deletion is reported after local invalidation and leaves the one-cycle shared bypass in place.
- These are cache-owned bounds, not transport pool limits (§5.6).

Eviction

- `maxEntries` covers retained positive values, negative values, stale fallbacks, failure/cooldown records, and idle invalidation markers.
- Evict the least-recently-used entry; eviction does not invalidate other clients or delete shared data.
- Active coordination is separately bounded by `maxActiveKeys` and cannot be pinned by hung I/O. An active entry may retain its candidate until its deadline, so the count limit is not a byte limit or an exact process-memory ceiling.
- Use unique job identities across eviction and recreation: an old completion must never match a newly created entry.
- Eviction may drop a failure/cooldown record or an idle one-cycle bypass; these preferences are best effort after eviction, not permanent tombstones.
- Applications requiring deletion across eviction need source/version guarantees or successful shared deletion.

Missing Records

- An admitted fresh confirmed-absence result clears that entry's old positive fallback.
- With `notFoundTtlMs > 0`, use the lifetime formula in §5.5 with that TTL instead of `ttlMs`, clipped by inherited bounds and any hard age limit. Never serve stale absence or extend it on failure.
- With zero, satisfy the current read once and retain no reusable negative value.
- A stale cached absence is not a fallback; continue loading and never turn it into fresh absence.
- Best-effort shared population replaces the old entry with the fresh absence envelope; its storage expiry must not exceed the remaining negative lifetime.
- With negative caching disabled, attempt bounded shared deletion only if this refresh saw a positive shared entry or the shared read failed, was bypassed, or was inconclusive. Skip deletion after an explicit shared miss or absence.
- Concurrent remote changes and late writes remain subject to the existing ordering limits. Failure of either remote operation may allow another client to see an older positive value under normal policy.
- Creation of a previously absent record needs invalidation or expiry of its negative entry.

5.9 Failure Policy and Reporting

Classification

- Normalize backend exceptions into the codes below; unmapped errors remain `unknown`.
- `capacity` means dependency-reported saturation; `admission-limit` means the cache declined local work before dispatch.
- The classifier receives the failure and stage and returns a completed local decision on retry and stale eligibility, with no I/O or deferred result. Failure to produce a valid decision becomes `policy-error`.
- Cooldown is independent of either decision.

Code	Retry default	Stale default / action
`timeout`, `transport`, `unavailable`, `capacity`	Within budgets	Allowed by read mode; physical recovery stays with the dependency owner.
`admission-limit`	No; no cooldown	Allowed by read mode; return immediately without changing acquisition state.
`throttled`	No immediate retry	Allowed; apply cooldown. HTTP statuses/Retry-After require explicit normalization.
`upstream-stale`	No	Allowed by read mode; it is not recovery.
`corrupt`, `invalid-data`	No	Denied; shared corruption may fall through to a valid loader result.
`unauthorized`, `configuration`, `unknown`	No	Denied by default.
`policy-error`	No	Denied; cannot classify itself permissively.
`aborted`, `superseded`	No	Denied; affect only their calls, not newer cooldown/state.

- Data-type overrides may change ordinary error eligibility, not policy errors or cancellation/supersession controls.
- Classification cannot make `admission-limit` retryable or start cooldown; it may restrict its stale permission.
- Intermediate shared failures do not start entry cooldown when loading succeeds.
- Valid empty data is success. Shared-write and cleanup errors cannot undo local delivery.

Reporting

- Report terminal operations once with identity, type, optional key, stage, attempts, code, and local cause.

- Distinguish optional attempt events; coalesce stale/recovery transitions instead of reporting every read. Per-read metadata carries delivery status.
- Reporting is deferred and never awaited by a data operation.
- Keep one current notification and one replaceable pending item per type, not per key. Aggregate keyed summaries to a bounded size; do not create an unbounded key list or per-key notification queue.
- `reportTimeoutMs` closes the logical gate even if the sink never settles; later notifications can try again without renewing old deadlines.
- Drop expired items, handle late failures, and do not report sink errors recursively through itself.
- The sink owns physical resources and remains subject to the execution-capacity rule in §5.6.

5.10 Integration and Verification

Integration

- For join replacement, acquire the complete all-mode value once, or collect the required identifiers and make bounded keyed calls. Enrich the selected records in memory; avoid a loader call per record when batching is available.
- Preserve existence, filtering, ordering, grouping, pagination, and response meaning. Blocking cache mode is not a substitute for authoritative validation when the application requires current truth.
- Reuse clients across requests and forward envelopes through cached endpoints.
- Loaders keep their normal contract after invalidation; the cache neither forces a bypass of every upstream cache nor certifies a global update.
- Applications set freshness bounds and implement immediate notification outside the library.

Use controlled clocks/scheduling and dependency fakes, then real adapter tests. Verify the following without treating this plan as measured or implemented behavior:

Reads and Freshness

Scenario	Required observation
Both modes; objects, arrays, scalars; all read modes; cold, fresh, stale, hard-expired, empty, zero, and false values	Correct value or explicit error; no invented fallback or false miss.
Same-entry concurrency and independent entries	One current refresh per entry; joiners do not extend deadlines or block unrelated entry coordination.
Immediate stale delivery; slow external setup	Caller receives data before refresh/reporting code executes.

Scenario	Required observation
Upstream stale, older candidates, source recovery, and invalid data	Correct candidate selection; only fresh outcomes replace recovery state or reach shared storage.
Zero TTL, long acquisition, exact boundaries, skew, clock shifts, and metadata copying	No recursive refresh or age reset; waits remain monotonic and admission conservative.

Failures and Recovery

Scenario	Required observation
Any loader/shared read/write/delete/cleanup never settles	Cache-owned deadlines complete the job without physical-cancellation acknowledgment.
First load hangs; next load succeeds after cooldown; old work later settles	Recovery without restart; no late population, retries, state changes, or unhandled late failure.
Concurrent state transitions and supplied callback re-entry	Atomic admission and completion; capacity limits hold, obsolete results cannot commit, and cache coordination is released before supplied code runs.
Shared read hangs while loader is healthy	Reserved time reaches the loader.
Every error class, malformed decisions, recursion, retry delay, and cancellation	Correct retry/fallback/cooldown; original budgets hold, timers clear, and other waiters continue.
Reporting never settles	Data calls continue; notification gate expires and later delivery is possible.
Adapter saturation, broken cancellation, restart, and no accessible snapshot	Bounded cache outcome; transport repair and memory durability are not falsely guaranteed.
Local admission full, then freed; dependency saturation	Immediate `admission-limit`, zero attempts, no cooldown/state change, and next-call admission; dependency `capacity` follows retry policy.

Invalidation

Scenario	Required observation
Reader/owner invalidation; failed/hung/overlapping deletes	Correct local clearing, bounded one-cycle owner bypass, and exactly-once finalization.

Scenario	Required observation
Invalidate then load from a cached endpoint or replica	Normal eligible result accepted without new authoritative-proof capability or a later timestamp requirement.
Invalidation during load/backoff/write; repeated writer replacement	Superseded callers settle; no obsolete new effects or unlimited task queue.
Change with no event, then an application event; correlated client reads	Normal TTL/stale behavior until local invalidation; no inferred broadcast; potential stampede visible.

Keyed Batches and Bounds

Scenario	Required observation
Scalar/array, duplicate/empty/invalid keys, mode mismatch, and oversized input	Stable map shape/order, no duplicate acquisition, and validation before I/O.
Mixed local/shared hits, overlapping batches, partial failure, and missing/extra loader keys	Per-key outcomes; one current load per key; retries never repeat completed keys.
Key invalidation, cancellation, or timeout during a batch	Only affected subscriptions/tokens end; other keys can succeed; original deadlines hold.
Negative hits, new records, stale absence, and failed absence population/deletion	Explicit absence; no old local fallback or stale negative reuse; remote visibility limits remain clear.
Partial bulk shared results, omitted outcomes, and whole-call hangs	Valid hits/misses preserved; only failed keys retry; loader reserve and original budgets hold with either adapter shape.
Repeated confirmed absence after explicit shared misses, positive hits, or failed reads	No redundant delete after a known miss; bounded cleanup only for observed/uncertain positive entries.
Retention churn, full active/writer/delete capacity, and eviction during outstanding work	Bounded retained/logical state, prompt admission outcomes, no old-identity population, and reload after eviction.

Measure calls, latency, attempts, stale age, cooldown suppression, pending cache jobs, and retained memory under normal and outage loads. Observe physical dependency resources separately; a completed cache call does not prove its remote work stopped.

6. Rejected Alternatives

Alternative	Reason and accepted trade-off
Fetch the source every time	Repeats unchanged acquisition; still appropriate for operations needing current authoritative truth.
Shared cache only	Retains network work per read; local copies accept duplicated memory and independently timed updates.
Await shared-write confirmation	Couples successful acquisition to another I/O result; best-effort writes accept later shared misses.
Arbitrary query caching	Adds query identity and invalidation dependencies; keyed mode targets explicit record identifiers only.
Separate all/keyed libraries	Duplicates freshness, failure, metadata, and recovery contracts; mode-specific entry/loading rules fit one library.
Single-key-only loaders	Causes one source call per cold record; the batch contract also handles a single key.
Distributed loading/invalidation inside the library	Adds locks and event-delivery obligations; applications own immediate cross-client coordination.
Renew TTL on failure	Can retain old data indefinitely; fixed age bounds separate freshness from retry timing.
Hold physical permits until calls end	Can permanently park an entry; cache state releases at timeout while adapters own resources.
Require authoritative proof after invalidation	Excludes ordinary cached/replica loaders; applications enforce stronger source/version guarantees when needed.
Queue readers until cooldown/capacity recovers	Adds waiting state; return permitted stale data or a bounded outcome instead.
Full circuit breaker	Adds thresholds/hysteresis beyond cooldown and single recovery acquisition; applications may wrap dependencies.
Mandatory version-conditional writes	Requires backend/version semantics beyond read/write/delete; stronger remote ordering is an extension.
Scheduled refresh/prewarming	Adds idle work and lifecycle scheduling; the first request may wait when no retained value is eligible.

© 2026 James Kunley. All rights reserved. Latest version: jameskunley.com/designs/reference-data-cache/