When Corrected AI Answers Return: A Consistency Contract for Human-Reviewed Systems

Andrew Koski

6 November 2026 | Working paper · Version 1.0 | Draft prepared 6 November 2026

Abstract

A human correction can be acknowledged while an older AI result remains capable of reappearing through a cache, delayed worker, or secondary read path. This is a consistency problem with direct implications for user trust. I propose a revision-based contract in which a result's eligibility depends on the current correction state of the subject to which it applies. The paper distinguishes cache expiry, invalidation, and atomic rejection of stale writes, then describes a bounded protocol and its assumptions. An illustrative execution shows why deletion alone is insufficient. A proposed evaluation measures stale-result exposure, correction visibility, availability, and cost under controlled interleavings and failures. The proposal applies to reviewed extraction, catalog identity, classification, and recommendation records. It does not claim that human correction retrains a model, removes all copies of data, or makes the corrected answer true.

Keywords: human review; cache consistency; revision fencing; stale writes; AI systems

Explore the paper

1. The position
2. Why deletion alone fails
3. Define the correction contract
4. A bounded implementation model
5. Scope, identity, and partial correction
6. Proposed evaluation and failure tests
7. Conclusion
References

1. The position

A user corrects an AI answer and receives confirmation. Later, the old answer returns. From the user's perspective, the system has ignored the correction. From the system's perspective, several individually reasonable operations may have occurred in an unfortunate order. I argue that acknowledging a correction should carry a defined visibility guarantee, not merely confirm that one database write succeeded.

The intended audience includes engineers building reviewed extraction, catalog enrichment, moderation support, and other products where machine suggestions become stored records. The recurring issue is the relationship between a human decision and previously computed results. The model may be unchanged throughout. The failure can occur entirely in ordinary application infrastructure.

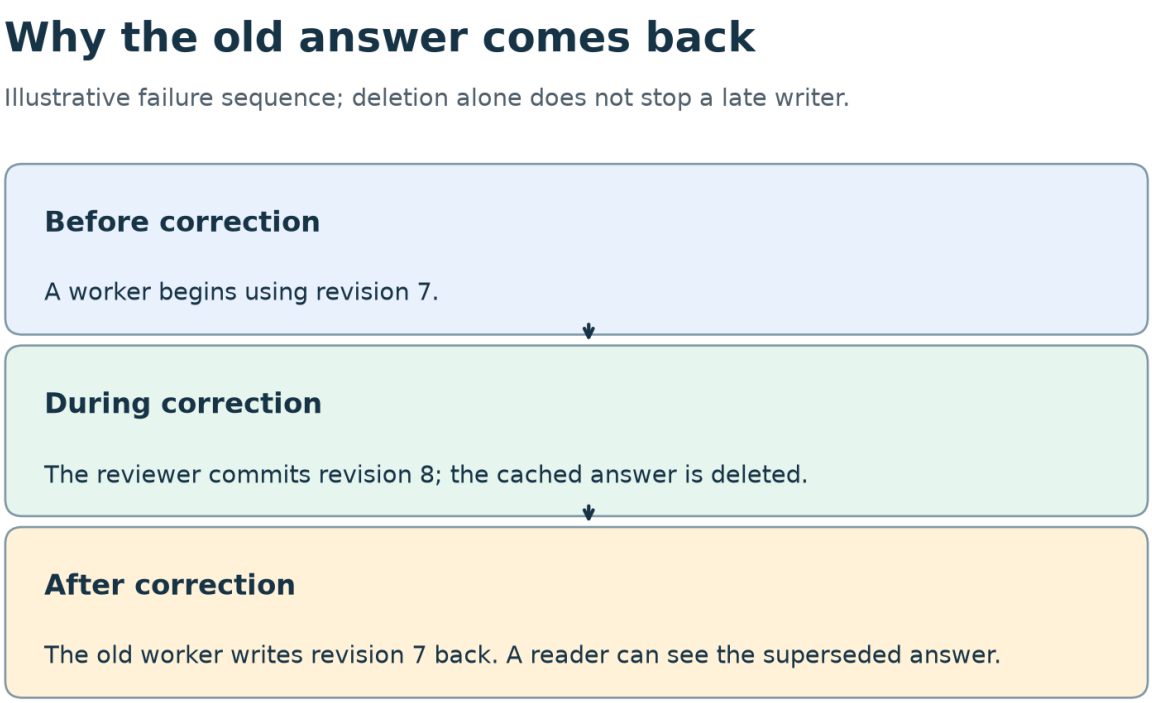


Figure 1. Illustrative race: a late worker can restore an answer after cache deletion.

2. Why deletion alone fails

Consider a worker that begins computing an answer against revision 7. A reviewer then corrects the subject and the application deletes its cached result. The worker finishes after the deletion and writes its old answer back. A later reader receives that answer. The correction and deletion can both have succeeded while the user-visible behavior remains wrong.

This is an illustrative execution, not a new experimental result. It explains why a time-to-live value only bounds retention under particular assumptions; it does not establish that a correction takes effect immediately. A shorter expiry can reduce the period of exposure while still allowing stale results to return repeatedly.

Distributed systems distinguish event order from wall-clock timestamps. Lamport's treatment of the happens-before relation is foundational to reasoning about such executions. [1] The protocol below uses a subject revision allocated by an authoritative store. It does not use timestamps from independent machines as evidence that one decision supersedes another.

3. Define the correction contract

Let s identify a subject and r(s) its current authoritative revision. A generated result carries the revision at which it was produced and the evidence and policy versions relevant to its meaning. The proposed rule is: a result can be accepted for the subject only when its revision equals the current revision at the point of the acceptance decision.

A correction atomically advances r(s) and records the accepted correction or rejection state. An asynchronous worker retains the revision it observed at the start. Before publishing its result, it must compare that revision against current authority in an operation that cannot race with a correction

commit. A read path must also validate eligibility or obtain its answer through a boundary that already provides the same guarantee.

The strong contract proposed here is that a read begun after a correction has been acknowledged cannot return a superseded revision. An overlapping read may complete with the older revision if it is ordered before the correction under the declared consistency model. If the product requires every already-open client to update immediately, an additional delivery and client-refresh contract is needed. Server-side revision checks alone do not provide it.

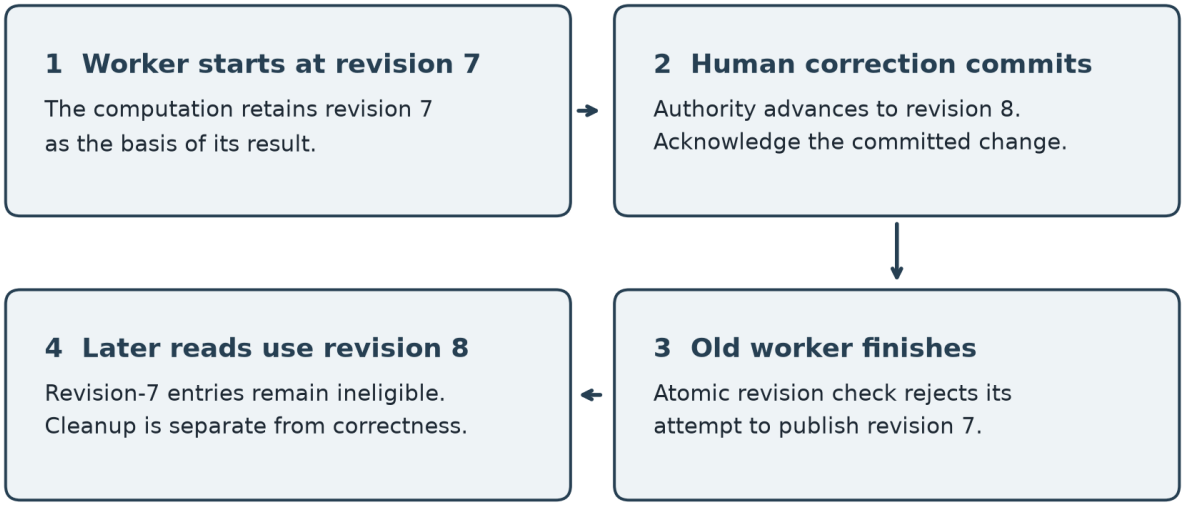


Figure 2. Illustrative stale-write interleaving. Revision checks must be atomic with acceptance; deletion alone does not reject a late writer.

4. A bounded implementation model

One implementation keeps the subject revision and accepted answer in the same transactional database. A worker attempts a conditional update whose predicate includes the expected revision. A zero-row update means that its evidence has been superseded; it must discard or recompute the result. The correction and answer acceptance participate in the same serialization mechanism. This description is a protocol proposal, not a claim that a particular database isolation setting automatically satisfies it.

A cache can be keyed by subject and revision. After obtaining the current revision from authoritative storage, a reader may reuse the corresponding immutable cache entry. Old entries may remain physically present but are no longer eligible. This moves cleanup out of the correctness path at the cost of an authoritative revision lookup. Caching that lookup without a stronger coherence mechanism weakens the visibility guarantee.

A compare-and-set operation on one cache does not automatically coordinate a separate database. Redis documents WATCH as an optimistic locking mechanism for Redis transactions. [2] That can support a single-store design, but it does not make an external database update and a Redis write atomic together. Cross-store designs must specify their authority, failure handling, and recovery rather than infer safety from the presence of a transaction in either component.

When authority is unavailable, a product must choose between refusing the strong guarantee and temporarily refusing the dependent answer. Returning an old entry while calling it current would violate the contract. Some products may reasonably provide a clearly marked stale suggestion, but that is a different service level and should not authorize actions requiring current review state.

5. Scope, identity, and partial correction

A revision must attach to the right subject. Correcting a catalog product does not necessarily correct every photograph, seller listing, or physical item linked to it. Conversely, correcting only an image cache can leave a derived catalog record stale. The dependency boundary should identify which claims are invalidated and which remain supported.

A coarse subject revision is easier to reason about but may invalidate unrelated fields. Fine-grained revisions can reduce recomputation while introducing dependency-management complexity. The proposed evaluation should include both. A system should not create a complete corrected snapshot from a few review fields unless the remaining fields have independently valid evidence.

Human approval does not retrain the recognition model. Rejection does not prove which alternative answer is true. Cache invalidation is not deletion of training data or removal of every downstream copy. These distinctions keep the promised correction behavior narrow enough to implement and verify.

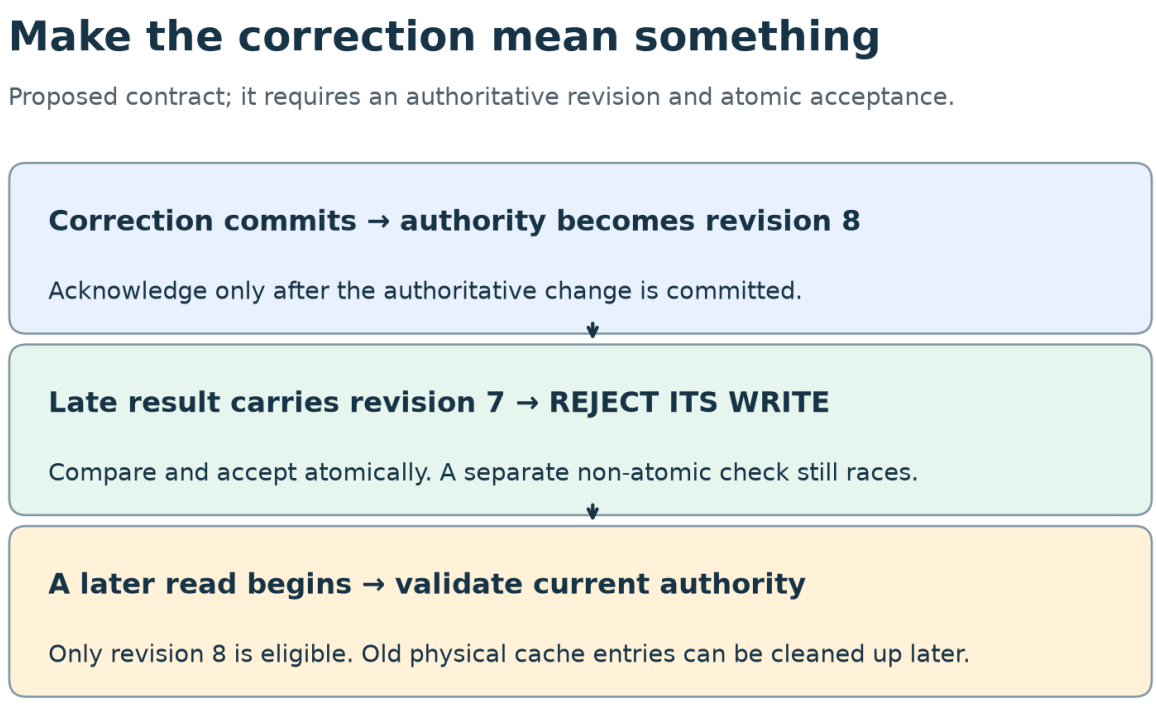


Figure 3. Proposed revision contract. Rejection must be atomic with acceptance.

6. Proposed evaluation and failure tests

Compare deletion-only invalidation, a short-expiry cache, and revision-checked acceptance under the same workload. Use a deterministic scheduler to reproduce workers finishing before and after correction, repeated corrections, retries, delayed messages, and concurrent readers. Record operation

histories with subject IDs and revisions so that violations can be detected independently of final database state.

The primary safety measure is the proportion of reads begun after acknowledged correction that return superseded results. Report absolute violations as well as rates; a rare event may still invalidate a strict contract. Secondary measures include time to acknowledgment, availability during authority failures, latency, cache hit rate, recomputation, and stale-entry storage overhead.

Test process crashes before and after every state transition. Include secondary caches, search indexes, background enrichment, and browser-held responses within the declared scope. A system can satisfy its primary read path while a search endpoint reintroduces the same defect. Clock skew and message reordering should not change correctness if the design truly relies on revisions rather than timestamps.

The protocol fails its stated guarantee if any allowed execution returns a superseded answer to a read begun after correction acknowledgment. Practical usefulness is a separate question: even a correct protocol may be unsuitable if its authority dependency makes the product unacceptably slow or unavailable. Both conclusions should be reported. No performance advantage is assumed in this paper.

7. Conclusion

I regard correction visibility as a product contract that deserves the same precision as the original prediction. An AI system does not become trustworthy merely because it offers a correction button. The application must define which prior decisions that correction supersedes and prevent them from silently regaining authority. Revision checks are one concrete way to express that requirement. Their value depends on a complete boundary, explicit consistency assumptions, and tests that examine the interleavings where stale answers return.

References

[1] Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), 558–565. https://lamport.azurewebsites.net/pubs/time-clocks.pdf

[2] Redis (n.d.). Transactions: Optimistic Locking Using Check-and-Set. Official documentation, accessed 6 November 2026. https://redis.io/docs/latest/develop/using-commands/transactions/