# Conflict-Free Team Synchronization via Per-Writer Shards

kgai maintainers

kgai.dev · team@kgai.dev · August 2026

**ABSTRACT**

Sharing a decision memory across a development team normally means either operating a server or merging files, and merging files means merge conflicts. We describe the synchronization design of kgai, a decision memory kept locally by AI-assisted development teams, in which textual merge conflicts cannot occur by construction. Every writer appends immutable, content-addressed decisions to its own shard, and synchronization uploads write-once segment objects to an S3-compatible object store the team owns, with no server component and no client-side synchronization state. Because the replicated object is a grow-only set of immutable events with a deterministic total order, every replica converges to a byte-identical graph, which the system can verify. We position the design against the CRDT literature, identifying the convergence result we borrow and the automatic merging of application state we deliberately avoid. Genuinely contradictory decisions are not merged but surfaced as a first-class visible state, a branch, resolved by one new recorded decision. Synchronization is opt-in, and we state the design's operational limits.

## 1. Introduction

A decision memory becomes most valuable at the moment it is shared, because the decisions a developer needs are usually someone else's. Sharing, however, is where memory systems acquire their operational weight. The common designs either centralize writes behind a server, which someone must run and everyone must trust, or replicate files between machines, which reintroduces the merge conflict as a routine cost of collaboration.

This report describes the synchronization design of kgai, a local-first store of a team's engineering decisions, which takes neither path. Synchronization is opt-in, runs against an S3-compatible object store that the team itself owns, involves no server component of ours, and is structured so that a textual merge conflict cannot occur at all. The guarantee is not the product of a clever merge algorithm but of an arrangement in which nothing is ever in a position to conflict. We present the data model that makes this possible, the wire protocol, the argument for conflict freedom, the design's relationship to the literature on

replicated data types, and the treatment of the one kind of conflict that is real and must not be hidden, the semantic kind.

## 2. The Data Model

The unit of replication is a decision, an immutable event that records a structural choice, its author, its date, and its rationale, and whose identifier is a hash of its content. Decisions are appended to a log that is never rewritten, an arrangement whose systems benefits are argued well in the immutability literature [8]. The log is not one file but a set of per-writer shards. When an installation initializes it mints an identifier of its own, and from then on it appends to its shard and no other, so every file in the store has exactly one writer, ever.

Locally, the log is the source of truth, and the queryable graph is a derived projection, rebuilt from the log deterministically. Replication therefore concerns logs alone. No database is ever merged, because the database is disposable.

## 3. The Segment Protocol

Synchronization speaks a stateless segment protocol against the object store. On push, an installation uploads a write-once object holding its next batch of events, keyed by its installation identifier, its position in its own shard, and the cumulative event count. On pull, it lists the segment keys of other writers and downloads exactly those segments whose cumulative counts exceed the length of its local copy of each shard.

Two properties of this protocol carry weight. First, there is no client-side synchronization state to lose or corrupt. What to push and what to pull is recomputed on every run from the local shard lengths and the counts encoded in the keys. Second, downloads are verified before they land, both against the content hashes of the events and against the hash-chain continuity of the shard, so a corrupted or truncated segment is rejected rather than absorbed.

Every synchronization ends with a full rebuild of the local projection in canonical order, never an incremental patch. An event pulled from a teammate may belong earlier in the canonical sequence than events already applied, and property updates that resolve by write order would drift apart across installations if patched in arrival order. Rebuilding from the whole log in canonical order restores identical state everywhere, and it stays cheap because the live element graph is deliberately kept small.

## 4. Why Textual Conflicts Cannot Occur

A merge conflict, in the version control sense, requires two writers to produce divergent versions of one artifact. The construction removes every precondition. No two installations ever write the same shard,

segment objects are write-once and never modified, and events themselves are immutable and content-addressed. The synchronized state of the team is simply the union of all writers' shards, and set union over immutable events is commutative, associative, and idempotent. Arrival order cannot matter, repetition cannot matter, and there is no artifact anywhere in the system of which two divergent versions can exist.

Union alone would still permit replicas to disagree about the derived graph if projection order were ambiguous. It is not. Events carry Lamport clock values [7], and projection replays them in the total order given by the clock value and the content hash, an order every replica computes identically and whose construction is the subject of the ordering report in this series (TR-2026-02). Two stores that have pulled the same events therefore hold byte-identical graphs, and the system can verify this directly by comparing canonical export digests. The consistency model is eventual in the standard sense [6], with the addition that convergence comes with a checkable witness.

## 5. Relation to the CRDT Literature

Readers of the replicated data types literature will recognize this construction, and precision matters about exactly what is borrowed and what is refused.

What we borrow is the oldest and simplest result. A grow-only set of immutable elements, merged by union, is among the elementary state-based CRDTs, and its convergence follows directly from the theory [1, 2]. Our replicated object is exactly such a set, with the Lamport order supplying a deterministic linearization for projection on top of it. The ancestry of the wider approach is also plain. Reconciling per-writer logs after disconnected operation goes back to Bayou [4], and building availability on an object-versioned, coordination-free store was demonstrated at scale by Dynamo [5].

What we deliberately avoid is the part of the field that has absorbed most of its recent effort, the automatic merging of concurrently edited application state. A replicated JSON document that merges concurrent edits from all parties without losing any of them is a genuine achievement [3], and it is the correct tool when the replicated object is the artifact itself, a document, a canvas, a data structure. Our replicated object is a history of engineering judgments, and we hold that two contradictory judgments must never be merged into one by an algorithm. kgai therefore applies automatic convergence only beneath the level of meaning, in the ordering and application of events. One narrow exception sits at the boundary and should be named: concurrent updates to the same descriptive property of an element resolve by the canonical write order, a last-writer-wins rule at property granularity. Both writes remain in the log, but the projected value is chosen by order rather than by intent, which is the standard price of that rule.

## 6. Semantic Conflict as a First-Class State

What automatic merging would have hidden, the design surfaces. When two writers take the same element in different directions within the same window, neither decision supersedes the other, and once

their logs merge that element carries two competing head decisions. We call this a branch, and it is the only real conflict the system recognizes.

A branch blocks no synchronization and signals no storage error. It is a visible, queryable state. A dedicated query lists every branched element with both heads, and resolution is performed in the vocabulary of the system itself: a single new decision recorded on the element supersedes both heads. History keeps the branch and its resolution for good, so the disagreement survives in the record as information rather than being smoothed away.

The contrast with Bayou is a useful close. Bayou resolved update conflicts through application-supplied merge procedures executed by the system [4]. Here the resolving artifact is a reasoned decision rather than a procedure, authored by a person or their agent, carrying its own rationale, and subject to the same immutability as everything else. The system's role ends at detection and preservation. Judgment stays with the team.

## 7. Operational Properties

Synchronization is opt-in. A store with no configured remote is complete and fully functional on one machine, and nothing leaves it. Configuring a remote is a single setting naming a bucket and prefix, with credentials resolved the standard way for the storage provider. Once configured, synchronization runs automatically in the background of working sessions, throttled, and without blocking any foreground work.

Because installation identity determines shard ownership, a store directory copied wholesale to another machine would let two machines extend one shard and silently fork its hash chain. The system detects this case, fails loudly instead, and provides an explicit operation that gives a copied store a fresh identity. For the same reason a store belongs to one user on one machine, and colleagues on a shared host must hold separate stores and let synchronization merge them.

The transport has been validated with concurrent writers, races on write-once segment arbitration, and copied-store fork detection, and exercised against stores of one million decisions on both a self-hosted S3-compatible service and production S3. Performance measurement is out of scope for this report and belongs to the scale report in this series (TR-2026-09).

## 8. Limitations

The design's honesty requires naming its edges. Convergence is eventual, and between synchronizations teammates hold different logs and receive different answers, a window that widens for developers who work offline. The bucket is a single point of administrative trust: any writer with append access to it is believed, and the integrity checks verify shape and continuity, not authorship. Signing decisions for zero-

trust remotes appears on the project roadmap, and this report makes no commitment about it. The last-writer-wins rule for concurrent property updates, discussed in Section 5, resolves by order rather than intent. Branches require human or agent attention, and a team that never consults the conflicts listing accumulates unresolved ones. Finally, the write-once segment protocol is designed for object stores, and the guarantees described here are claimed for that transport as deployed, not for every storage backend a team might improvise.

## 9. Conclusion

We have described a synchronization design in which conflict freedom is obtained by construction rather than by resolution. Per-writer shards leave no artifact that two writers could touch, write-once segments in a team-owned object store take the place of a server, statelessness does away with fragile bookkeeping, and a deterministic total order turns a union of logs into a byte-identical graph on every machine. From the CRDT literature we take the elementary convergence of a grow-only set and decline the automatic merging of meaning, because in an engineering record a contradiction is a fact worth keeping. The result is a synchronization layer that a team adopts by naming a bucket, owns end to end, and can verify, with its one honest conflict class made visible and its resolution recorded in the same immutable history as everything else.

## References


[1] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. 2011. Conflict-free Replicated Data Types. Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011), Lecture Notes in Computer Science, Vol. 6976. Springer, 386-400.

[2] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. 2011. A Comprehensive Study of Convergent and Commutative Replicated Data Types. INRIA Research Report RR-7506.

[3] M. Kleppmann and A. R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. IEEE Transactions on Parallel and Distributed Systems 28(10), 2733-2746.

[4] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995), 172-182.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007), 205-220.

[6] W. Vogels. 2009. Eventually Consistent. Communications of the ACM 52(1), 40-44.

[7] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7), 558-565.

[8] P. Helland. 2015. Immutability Changes Everything. ACM Queue 13(9). https://queue.acm.org/detail.cfm?id=2884038