Total Order Without Coordination: Lamport Clocks and Content Hashes in a Multi-Writer Log

kgai maintainers

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

ABSTRACT

A decision log written by many independent machines and synchronized through an object store under the team's own control, with no coordinating server, must nevertheless replay identically on every machine. This requires a total order over events that every replica computes for itself, from the events alone, regardless of when the events arrive. We describe how kgai obtains such an order. Every event carries a Lamport logical clock value, and ties are broken by the event's cryptographic content hash, which embeds the identity of the writing installation. The resulting order over pairs of clock value and hash is total, deterministic, and computable without communication. We explain why wall-clock timestamps are excluded from the ordering entirely, what the total order guarantees for replay, and what it deliberately does not decide. Concurrent contradictory decisions about one element are not resolved by the order. They surface as explicit branches that a later decision supersedes, so arbitration over meaning stays in the record rather than in the transport.

1. Introduction

kgai records a team's engineering decisions as immutable events in an append-only log, and derives from that log the graph its readers query. The event model report in this series (TR-2026-01) describes the events themselves. This report concerns their order.

Order would be trivial with a central server assigning sequence numbers. kgai deliberately has none. The system is local-first. Each installation writes to its own shard of the log on its own machine, and teams move shard contents between machines through an S3-compatible object store of their own, carried as write-once segments, without a coordination service and with nothing resembling synchronization state kept on any client. Events therefore arrive at each replica in arbitrary order and at arbitrary times, and two installations may write concurrently without either knowing of the other.

Yet every replica must replay the log to the same graph. The projection includes operations whose outcome depends on order, so replay order cannot be left to arrival order, which differs between machines. What is needed is a total order over all events that is a pure function of the events themselves.

Any two replicas holding the same set of events must sort them identically, whatever the history of how those events reached them.

kgai's answer combines two classical ingredients. Every event is stamped with a Lamport logical clock value [1], and ties are broken by the event's content hash, which embeds the identity of the writing installation. Section 2 states the requirements precisely. Section 3 explains why wall clocks are excluded. Sections 4 and 5 develop the clock and the tiebreak. Section 6 states what the order guarantees for replay, and Section 7 what it deliberately leaves undecided.

2. The Setting and the Requirements

The log is sharded by writer. Each installation mints an identifier at initialization and appends only to its own shard, one event per line of newline-delimited JSON. One writer per shard means synchronization is a union of append-only files. Two installations recording in parallel cannot produce a textual conflict, because they never write to the same file.

Synchronization runs over an object store. A push uploads one write-once object carrying the installation's next batch of events, named by a key that encodes the writer and the cumulative event count. Nothing needs remembering between runs, because comparing the counts in those keys with the local shard lengths determines what to push and what to pull. Every download is checked before it enters the local log, event by event against the content hashes and shard by shard against the continuity of the hash chain.

From this setting, four requirements on the ordering follow.

First, determinism. The order must be a function of the event set alone. Two replicas with the same events must produce the same sequence.

Second, no coordination. An installation must be able to stamp an event while offline, with no server to ask and no round trip to any other writer.

Third, stability under partial knowledge. A replica never knows whether more events exist elsewhere. The relative order of two known events must not change when a third event arrives, otherwise every synchronization could reorder history.

Fourth, totality. The projection applies operations whose results are order-sensitive, so no two distinct events may be incomparable.

3. Why Wall Clocks Are Excluded

The obvious candidate, ordering events by wall-clock timestamp, fails these requirements in ways both classical and specific to this system.

The classical objections are Lamport's [1]. Physical clocks on independent machines drift, skew, and are corrected by their operating systems, sometimes backwards [7]. An ordering keyed to them can place an event before its own cause, and two machines can honestly disagree about which of two events came first. Systems that have relied on loosely synchronized clocks for ordering have had to layer reconciliation machinery on top precisely because timestamp order and causal order come apart [5].

The objection specific to kgai is sharper. The decision event's date field is deliberately backdatable. A team importing decisions it made in years past is instructed to record them under their true historical dates, so that the visible timeline reflects the team's real history rather than the day the tooling arrived. A field that is designed to be backdated cannot key replay order. An import performed today would insert events that sort before everything already replayed, and it would do so differently on machines that received the import at different times.

kgai therefore splits the two roles cleanly. Wall-clock dates are provenance, carried for human readers and for history views. The replay order is keyed by a logical clock that no writer has any reason, or any interface, to manipulate.

4. Lamport Logical Clocks

Lamport's construction [1] assigns each event a counter value with a simple discipline. A process increments its counter at each local event, attaches the counter to what it emits, and on receiving information from another process advances its counter to exceed the largest value it has seen. The result is a partial order guarantee. If one event can causally influence another, the first carries the smaller value.

In kgai, every event is stamped with such a clock value when it is appended, and the store maintains the clock across synchronization, so events written after a pull are stamped above the values that arrived in it. The guarantee this buys is exactly the one that matters for a decision log. When a writer records a decision after having seen another decision, for instance a decision that supersedes one pulled from a teammate, the later record carries the larger clock value and replays after the record it builds on.

What the clock does not provide is totality. Two installations writing concurrently, each unaware of the other, can stamp different events with equal values. Lamport's paper already observed that the partial order can be extended to a total one by breaking ties with an arbitrary but fixed rule over process identity [1]. Vector clocks extend the construction to capture causality exactly, at the cost of state proportional to the number of writers [2], and Section 7 returns to why kgai does not need that strength.

5. The Content Hash as Tiebreak

kgai breaks ties with material the event already carries. Every event's identifier is a cryptographic hash of its content, and the hashed content includes the identity of the writing installation. Replay sorts the union

of all shards by the pair of clock value and hash.

This is Lamport's fixed arbitrary tiebreak, instantiated with three useful properties.

It is total. Two events with equal clock values were written by different installations or differ in content, so their hashes differ and the pair comparison always decides. Distinct events never compare equal.

It is computable from the event alone. The tiebreak needs no registry of writers, no configuration, and no communication. Any replica, including one that has never seen a given writer before, sorts that writer's events identically.

It is tamper-evident for free. The same hash that orders an event also verifies it during synchronization, so the ordering key and the integrity check are one value [3].

The tiebreak is arbitrary in the sense Lamport meant. Between two genuinely concurrent events, the hash order encodes no meaning, no priority between writers, and no claim about real time. It is a coin flip that every replica flips identically, which is all a replay order requires. Arbitration that carries meaning is handled elsewhere, as Section 7 describes.

6. What the Total Order Guarantees

The order underwrites one central guarantee. Two stores that hold the same events produce identical graphs when they replay, byte for byte, regardless of the order in which the events arrived at each. kgai makes the guarantee checkable rather than asserted. A canonical export of the graph yields a digest, and matching digests across machines witness convergence.

The mechanism is the state machine approach [6]. Replicas applying the same deterministic operations in the same order reach the same state. Classical state machine replication spends a consensus protocol to agree on the order. kgai removes that expense by construction. The event set needs no agreement, because shards are append-only and their union is well defined. The sequence needs no agreement, because the sort key is a pure function of each event. Agreement is replaced by arithmetic.

Order sensitivity in the projection is what makes this necessary rather than pedantic. Property writes are last-writer-wins per key, so two events setting the same property on the same element must be applied in the same order everywhere, or replicas silently diverge. Supersession chains, which record which decision is the current word on an element, are likewise built in replay order. The projection report in this series (TR-2026-03) describes how the projection additionally makes replay idempotent and tolerant of partially transferred logs, and why the graph is rebuilt in canonical order after synchronization rather than patched incrementally.

An alternative design could seek convergence without any total order, by restricting the data model to operations that commute, as conflict free replicated data types do [4]. kgai's operation set is deliberately

not commutative. Last-writer-wins property updates and supersession are order-sensitive because the domain is order-sensitive. A decision record is precisely about which considerations came in which sequence. Imposing a deterministic order preserves that expressiveness while keeping the convergence guarantee.

7. What the Order Does Not Decide

The total order is a replay order, not a judgment. Three boundaries deserve plain statement.

It does not arbitrate meaning. When two writers concurrently make contradictory decisions about the same element, both replay, in hash-decided sequence, and neither supersedes the other. The element then has two competing head decisions. kgai surfaces exactly this situation as a conflict, a branch in the decision history rather than an error, and it is resolved by a person or agent recording one new decision that supersedes both heads. The resolution is itself an event, so the disagreement and its settlement both remain in the record. Semantic arbitration is kept in the log, where it is auditable, rather than delegated to a transport level rule that would silently discard one side.

It does not claim to reflect real time between writers. Between causally unrelated events, the sequence is stable and meaningless. Readers who want the human timeline read the date field, which is carried for that purpose.

It does not detect concurrency. Lamport clocks cannot distinguish concurrent events from ordered ones, a gap vector clocks were designed to close [2]. kgai does not consume clock values to find conflicts. Concurrency becomes visible structurally, as two heads on one element with neither superseding the other, which is the only concurrency the domain cares about. The clock is thereby left with a single job, providing a causally consistent, coordination-free sort key, and the hash completes it into a total order.

8. Conclusion

A multi-writer decision log with no server needs an order that every replica derives independently and identically. kgai obtains one from two values each event already carries. The Lamport clock preserves causal sequence where it exists, and the content hash extends the partial order to a total one while doubling as the event's integrity check. Wall clocks are excluded from ordering because they are unreliable in general and deliberately backdatable here. The resulting order makes replay a deterministic function of the event set, which is what allows independent machines to hold byte identical graphs and prove it by comparing digests. What the order refuses to do is as important as what it does. Contradictions between concurrent decisions are surfaced and resolved in the record, not settled by a sort key.

References

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

[2] F. Mattern. 1989. Virtual Time and Global States of Distributed Systems. Proceedings of the International Workshop on Parallel and Distributed Algorithms, North-Holland, 215-226.

[3] R. C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. Advances in Cryptology, CRYPTO '87, Lecture Notes in Computer Science, Vol. 293. Springer, 369-378.

[4] 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.

[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] F. B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. ACM Computing Surveys 22(4), 299-319.

[7] M. Kleppmann. 2017. Designing Data-Intensive Applications. O'Reilly Media.