ChatGPT

Architecture Overview

High-level pipeline of our system from event creation to final state. All operations (data updates, policy changes, trust updates) are captured as signed events, forming a causal DAG (left, dashed box). These events are then deterministically linearized (topologically sorted with a tie-break) and replayed through an authorization filter ("deny-wins") to produce a convergent, policy-compliant state (right).

Our architecture treats **all changes** – both data edits and access policy modifications – as the same fundamental object: a **signed, hash-linked event**. Each event is appended to a distributed log and references prior events, creating a **directed acyclic graph (DAG)** that captures causal ordering [1] [2]. Peers exchange these events via gossip; even if nodes work offline, their operations later integrate into the DAG once shared. The **causal DAG** ensures that every operation's dependencies are known, and no event is applied before its predecessors. In effect, once all dependencies are satisfied, the events can be viewed as a **topologically sorted sequence** that extends a consistent history [3]. This gives us a notion of "time" or order without any central coordinator.

Crucially, each event is cryptographically **signed** by its author and **hash-linked** to its parents (prior events it builds upon). This makes the log *tamper-evident*: any alteration to past events would break the hash chain, and any illegitimate event (e.g. forged or modified) is detectable via its signature and hashes. In essence, the event DAG serves as an audit trail – **provenance** is ensured, and all replicas can verify the integrity and origin of events [4]. Policy changes (like granting or revoking permissions) are themselves events in the log, so **trust updates are distributed in-band** rather than out-of-band. This means the system inherently propagates authorization information (e.g. new user roles or revocations) just like data edits, using the same eventual delivery mechanism. It avoids any reliance on a central authority at runtime: trust decisions travel through the same channels as data.

Event Model and Causal Ordering

Events. Each event represents a single operation and includes: *(a)* a unique identifier (we use a cryptographic hash of the event's contents as the ID), *(b)* a set of one or more parent references (hashes of earlier events that this event causally depends on), *(c)* a payload describing the data mutation or policy change, and *(d)* the author's digital signature (and any accompanying credentials). The parent pointers form the **hash-linked DAG** structure. Typically an event will reference the latest events that the author knows about; this implicitly encodes causality (happened-before). If an event has two parents that were concurrent, it "joins" those branches and therefore *knows* about both. By construction, the DAG has no cycles (an event can never indirectly reference itself), and it grows monotonically as new events attach.

Causal DAG. The partial order defined by the DAG is our baseline for consistency: if event E2 is a descendant of E1 in the graph, then E1 causally happened-before E2 (E2 acknowledged E1 as prior). Events that are on different branches with no path between them are **concurrent** – they happened without knowledge of one another [5]. The DAG enables *causally consistent delivery*: a node will not process an event until it has all of that event's ancestors (its dependencies) in its local log. This way, every replica eventually sees events in an order that respects causality. We leverage cryptographic hashes as pointers, which means peers cannot "fake" dependencies – an event can only refer to a parent if it actually had that parent's hash, ensuring **integrity** of the history [6]. If an event arrives out of order, it's simply buffered until missing parents arrive [1], guaranteeing that whenever we process an event, we have its causal context.

The causal DAG by itself does not pick a winner between concurrent events, but it provides the scaffold on which we can define a deterministic total order. In summary, the event DAG gives us two key properties: **(1) Causal consistency** – all replicas eventually see any causally related events in the correct order; and **(2) Tamper-evidence and provenance** – thanks to hash linking and signatures, the log's history cannot be altered or forged without detection. These properties set the stage for the next step, where we impose a deterministic linear execution order over this DAG.

Deterministic Total Order Construction

To ensure that every replica ultimately *applies* events in the same sequence, we define a **deterministic total order** over the DAG. This is essentially a procedure to linearize the partial order. We perform a *topological sort* of the DAG (which respects all causal dependencies) and then break any ties between concurrent events in a fixed, repeatable way [7]. In our implementation, the tie-break rule can be as simple as comparing the events' unique hashes or another globally consistent attribute (such as a logical timestamp combined with author ID). This rule is deterministic: given the same set of events, every correct replica will produce *identically ordered* sequences. There is no coordination or leader needed – each node independently computes the sort, yet they all agree because the input DAG and the ordering criteria are the same.

Example: In the architecture diagram, events **e1** and **e2** are concurrent (no causal link). Our deterministic sorter might compare their hashes and decide that **e1** comes before **e2** in the total order. Event **e3**, which had dependencies on both e1 and e2, naturally comes after them. The result is a sequence [e1, e2, e3] (or [e2, e1, e3] if the tie-break favored e2 – but every replica will make the same choice). This deterministic replay order is the cornerstone of achieving **Strong Eventual Consistency**: if no new updates occur, all replicas will process the same events in the same order and thus converge to the same state [8] [9]. By ensuring a uniform conflict resolution order, we avoid divergent outcomes [10] – any conflicting concurrent operations are resolved in the same way everywhere.

It's worth noting that our approach of choosing a total order by hashing (or any fixed criterion) means the selection is *arbitrary but consistent*. Deterministic tie-break rules have been proposed in prior work (e.g. using lexicographic order of event IDs) to merge concurrent operations in CRDTs [11]. This doesn't attempt to capture "real time" – it simply imposes a stable logical ordering. As a result, even if two events were truly simultaneous or originated without knowledge of each other, all replicas will eventually agree on which one to apply first. We sacrifice a bit of real-time fairness for consistency; this is acceptable because our focus is on *eventual* state agreement and policy correctness rather than real-time ordering guarantees.

With the total order in hand, we now have what looks like a single, sequential log of all operations. We call this the **execution order**. However, this sequence may include operations that *should not be allowed* under the system's security policy (for example, an edit made by a user who *at that moment* didn't have permission). In a centralized system, such an operation would have been rejected at submission time [12]. In our decentralized setting, it might tentatively appear in the log (since there was no central authority to prevent it), but we will rectify that in the replay. The next stages focus on enforcing access control **after the fact** in a consistent manner across replicas.

In-Band Trust and Policy Events

TrustView. We maintain a structure called the **TrustView** that represents the current authorization state (which users/keys are trusted and what permissions they have) at any point in the log. The TrustView is essentially derived from all **policy events** in the history. For example, if an event grants Alice edit access, the TrustView will reflect Alice as an authorized editor from that point onward (until perhaps another event revokes it). Rather than relying on an external PKI or central database of roles, we distribute trust information *in the same log as other events* – this is what we mean by **in-band trust distribution**. The initial TrustView can be bootstrapped by a *genesis* event (signed by a system owner) that establishes the founding permissions (e.g. who the first admin is, or which root keys are recognized). Thereafter, any changes to access control (adding a user, changing roles, revoking credentials) are recorded as events.

Each replica can compute the TrustView by scanning through the total order and updating a set of **issuer keys and their status/roles** [13] [14]. We include in events any needed verifiable credentials – for instance, an admin adding a new user might attach a certificate of that new user's public key, signed by the admin. When processing such an event, a node will verify the signature on the certificate and the event itself. If valid, the TrustView is updated to mark that new user key as active (and possessing whatever role was granted). Similarly, a revocation event (say, an admin removes a user or a key is reported compromised) will update the TrustView by marking that identity as *revoked* or lowering its permissions. The TrustView thus encapsulates the *set of valid identities and their privileges at the current execution point*. We design it so that all information needed to verify trust is carried alongside the events – for example, if membership is federated or cross-organization, we might bundle **verifiable credentials** attesting to a user's group membership or attributes, all signed by trusted issuers. Peers incorporate those credentials into the TrustView as they appear.

Because trust changes are part of the event log, they are propagated reliably and quickly to all replicas (subject to the same eventual delivery guarantees as data ops). This ensures that *policy decisions (e.g. revocations) made by one replica will eventually be known and enforced by all others.* There is no out-of-band delay; as soon as the revocation event is received, a replica will incorporate it. The TrustView at different replicas will remain logically consistent since they build it from the same ordered sequence of events. Any difference in local knowledge is temporary and resolves once events propagate.

Authorization Checking. With an up-to-date TrustView, the system can **validate each event's legitimacy** as it is replayed. In the total order replay process, when we reach an event, we check: *Did the TrustView (i.e., the set of policies in effect) at the moment of this event allow it?* This could involve checking the author's role and the operation type. For example, if the event is a data edit by user *U*, we ask: "At this point in the log, is *U* supposed to have permission to do this?" If yes (the TrustView says *U* is authorized for that action), then the event is provisionally allowed to execute. If no, then this event violates the current policy.

Notably, the "moment" of the event is defined by our total order position. Since we replay events in sequence, by the time we get to an event, we have processed all earlier events (including any policy changes that happened before or concurrently with it, which are now earlier in the order). Thus, the TrustView we hold reflects *all causally prior policy decisions*. If a policy change was concurrent with the event, its position relative to the event in the total order determines which one takes effect first. We conservatively treat concurrent revocations in a **"deny-wins" fashion**: if, by the time we consider event *E*, there exists any policy event earlier in the total order that revokes *E*'s author or permission, *E* will be considered unauthorized (even if in real-time the author might not have known of the revocation). This approach aligns with common access-control practice where an explicit deny or removal takes precedence over any allow [15] [16]. It ensures that **revocations propagate globally** – once a user is removed in one branch, any other branch's actions by that user are eventually nullified.

Authorization Epochs and the Deny-Wins Rule

One way to conceptualize the replay is to divide the history into **authorization epochs** separated by policy-change events. An "auth epoch" is a segment of the total order during which the set of active permissions (the TrustView) remains static. When a policy event (like a role change) occurs, it starts a new epoch with a different TrustView. Within a single epoch, all operations are governed by the same set of rules. In our implementation, we increment an *epoch counter* whenever we apply a trust or policy event, and tag subsequent events with the current epoch ID. This is mainly for auditing and easier reasoning – it lets us say "Event *E* was executed under policy epoch 5, which had these active admins and users."

When we reach a new epoch, we also retrospectively know that any concurrent events that were not ordered before the policy change might effectively belong to the new policy regime. For example, suppose an admin revokes user *U* at event R, and *U* had an offline edit event X that is concurrent (neither knew about the other). In the total order, let's say our deterministic sort places R (the revocation) before X. Then by the time we process X, the TrustView has *U* marked as revoked – we will **deny** event X. If instead the tie-break placed X before R in the order, then at the moment of X, *U* isn't revoked yet, so we would tentatively allow X. But when we later process R, we realize *U* should be removed. Our algorithm will then **re-evaluate** any events by *U* that came after R *in real time*. In practice, because we compute a final total order, we handle this by one pass of filtering: event X would not be adjacent to R (since R came later in order), so X was applied, but we detect at R that "U is now revoked." We then mark any of U's remaining future events for removal. This is effectively a *rollback* of X, since X is in the log earlier. We implement this by performing the replay in two phases: first pass to identify unauthorized events (or we allow tagging events with the epoch of their author's validity), and a second pass to actually apply only the authorized ones. In simpler terms, **the deny-wins rule retroactively purges any operations that a concurrent or prior policy change would forbid**. We find that in practice, a second pass (or an online check with slight lookahead) is sufficient to catch these cases.

The **deny-wins filter** is thus applied as we replay the events in order. Concretely, for each event in sequence: - If it's a trust/policy event, update the TrustView (and epoch). - If it's a data operation, consult the current TrustView to check if the author is permitted to do this operation *in the current epoch*. If yes, apply it to the state; if not, ***deny*** it – i.e. skip this event (do not apply it).

A denied event does *not* alter the state; effectively it is as if that operation never happened (except that it remains in the audit log, marked as rejected). All replicas do this uniformly, so unauthorized actions are consistently ignored everywhere. This approach provides a clear **Policy-Safety** guarantee: the state that results from replaying the log contains no effects of unauthorized operations – it's as if those were never accepted at time of execution [17] [18]. Yet, unlike a centralized system that rejects the operation upfront, our system may only finalize this decision after the fact, once the relevant policy knowledge (e.g. the revocation event) is available. Peers must treat authorization decisions as *tentative* until the log is sufficiently complete [18]. This optimistic execution allows work to proceed offline and without coordination, at the cost that some work might later be rolled back. In our design, if a user was wrongly assumed to have access and made changes, those changes will be rolled back deterministically on all replicas once the correct policy catches up.

It’s important to highlight that this filtering does not violate causality or integrity. Because policy events are part of the same DAG, a user’s subsequent edits might depend on their earlier (now-denied) edits. If an edit is denied, any later event that *causally* depended on that edit may become a no-op as well or invalid. However, since the user was revoked, presumably those dependent events were also by the removed user (or at least would also be unauthorized). Our replay can therefore consistently drop not just the one event but any downstream events that lack a valid causal history (we ensure an event that depends on an unauthorized event is itself treated as unauthorized due to broken trust chain). In practice, we haven’t encountered pathological cases – the deny-wins rule, combined with always requiring an unbroken chain of trust from the beginning of history, handles the logical dependencies. If something does slip through (say a collaborative edit by two users where one input was invalidated), the CRDT conflict resolution would handle it gracefully, likely favoring the effect of the authorized user.

Convergence and Correctness Guarantees

By design, our system achieves **eventual convergence** and **policy safety** *without runtime coordination*. We assume an asynchronous network where messages (events) are delivered reliably but with arbitrary delays. Under this assumption, we guarantee the following key properties:

- **Convergence (Consistency):** All correct replicas that have received the full set of events will *converge to the same application state.* In other words, the algorithm is deterministic and commutative: given the same multiset of events, every node’s replay (sorting + deny-wins filtering + CRDT application) produces an identical outcome. This holds because any two replicas will eventually have an identical event DAG (they exchange all events), and they both compute the same total order and apply the same filter rules. The conflict resolution (order + filter) is *uniform across replicas* [10] . Thus, no matter how events were generated or interleaved in real time, all nodes end up with a single coherent sequence of authorized operations. This is a **strong eventual consistency** guarantee – after quiescence, state is uniform everywhere [19] .

- **Policy-Safety:** The final converged state reflects a execution in which all operations obeyed the access control rules in effect. Any operation that was not authorized (according to the policy events and trust state in the log) has been purged from the executed sequence. We formally state this as: *if a policy (deny) event occurs (e.g. revoking a permission), then no operation violating that policy will appear in the final state.* The deny-wins replay ensures that the effect of any unauthorized event is erased. This property is analogous to the **Principle of Authorization** for distributed systems [17] – an event only changes the state if the state (as determined by preceding and concurrent events) allowed it. Our system upholds this principle in an eventual sense: once all relevant information is available, it will be as if no invalid action ever took place. This required us to sometimes retroactively undo actions, but it guarantees the integrity of the final outcome.

- **Integrity and Auditability:** Every event in the log is digitally signed and hash-linked, so malicious tampering (omitting events, forging events, or reordering them outside the rules) will be detected. Parties can audit the log and verify, for each event, a chain of trust: e.g., “User X’s edit at time T was accepted because at that point in the log, X was an active editor (granted by admin Y in a prior event), and here is the cryptographic proof of Y’s authority.” The log provides non-repudiation – if a user performed an action, their signature is attached. Even events that were denied still remain in the log record (marked as denied), which is useful for forensics or accountability. However, these denied events have no effect on the replicated state. The hash-linked structure ensures that if someone tries to prune or alter history (to hide an unauthorized action, for instance), all participants can notice the discontinuity [6].

Proof Sketch: *Convergence* follows from the fact that the replay procedure is a pure function of the set of events. All replicas eventually have the same set of events, and they use an identical deterministic algorithm, so they must arrive at the same result state [9]. The use of a total order ensures a single serial history is considered, and the CRDT apply step ensures that even if some operations commute, treating them in one fixed order yields the same state (effectively we are doing state machine replication without consensus, using the log's hash ties to prevent divergence). *Policy-safety* is maintained because for any event that would violate policy, there is at least one policy (deny) event in the log that precedes it in the execution order (if not, then it wasn't actually unauthorized). When that policy event is applied, our rules guarantee the offending event is filtered out. Even if the offending event came earlier in order, the moment the policy event appears, the system recognizes the violation and the final state is adjusted to exclude the unauthorized effect. This corresponds to finding a fixed-point in the self-referential conditions of authorization in a distributed timeline [20] [21]. Our algorithm's outcome can be seen as the **most restrictive valid interpretation** of the event set – it leans toward denying uncertain operations to preserve safety. Finally, because all replicas reach the same set of authorized events, *liveness* is preserved for authorized actions: any action that *should* be allowed (one that adheres to policy) will eventually appear in the state (assuming its event was propagated), since nothing will filter it out.

In summary, by making minimal assumptions (an eventually synchronous network, secure signatures, and an initial root of trust), we achieve **offline-capable, convergent access control**. Users can make progress offline or in parallel, and any conflicts – such as concurrent revocation vs. update – are resolved after the fact by a deterministic rule (deny-wins) rather than coordination. The cost of this approach is that some work may be wasted (rolled back) and that convergence might be delayed until policy information has caught up (convergence latency). However, our design favors availability: the system never globally locks or rejects operations upfront, so it's highly resilient to network partitions and downtime. Eventually, when all events have been exchanged, every replica will agree on a single, **policy-compliant** state of the data.

---

[1] [3] [4] [5] [6] [7] [11] Notes on building a convergent, offline-first Access Control CRDT ~ p2panda
https://p2panda.org/2025/08/27/notes-convergent-access-control-crdt.html

[2] [12] [13] [14] [17] [18] [20] [21] The Magic of Distributed Access Control
https://robin.town/blog/distributed-access-control

[8] [19] Understanding Data Consistency: Types, Examples, and Use Cases
https://medium.com/@ali75mnf/understanding-data-consistency-types-examples-and-use-cases-36ffdf14344a

[9] [10] [PDF] Replicated Data Types: Specification, Verification, Optimality
http://www.cs.ox.ac.uk/people/hongseok.yang/paper/popl14-final.pdf

[15] [PDF] Akeeba Ticket System for Joomla!™ 4 - Akeeba Ltd
https://www.akeeba.com/download/ats/5-2-0/ats-guide-pdf.pdf

[16] IAMRoles (docx) - CliffsNotes
https://www.cliffsnotes.com/study-notes/4166856