ChatGPT

Change-Set Ledger: Tamper-Evident, Idempotent, Forensics-Ready System

Introduction and Goals

We are upgrading the current **draft → approve → apply** workflow into a **Change-Set Ledger** system that is tamper-evident, idempotent, and forensics-ready. In this new design, all writes proposed by AI agents/tools (as opposed to direct manual writes) are recorded as append-only change-sets. The ledger provides cryptographic integrity (every change is linked via a hash chain), prevents duplicate or conflicting updates through idempotency controls, and captures rich evidence for each change to aid forensic analysis. Key objectives include:

- **Tamper-Evident Audit Trail:** An append-only log where each entry's integrity is verifiable via a cryptographic hash chain, with periodic checkpoints anchored externally for tamper evidence [1] [2]. This ensures that any alteration in the change history will be detectable, fulfilling compliance and legal requirements for data integrity [3].
- **Idempotency & Concurrency Safety:** Each proposed change carries an **idempotency key** to ensure that repeating or concurrent requests do not produce duplicate entries or race conditions [4] [5]. This protects against accidental double-apply (e.g. an agent retry or simultaneous approvals) by treating repeated operations with the same key as a single logical change.
- **Deterministic Ordering per Entity:** Changes are ordered in a **per-entity change stream** to maintain consistent, sequential updates for each individual record/object. This deterministic ordering (enforced via sequence numbers or versioning) prevents out-of-order application of changes to the same entity even under concurrent workloads [5]. It also simplifies race-condition handling by localizing conflicts to the entity scope.
- **Replay, Rollback, and Forensics:** The ledger acts as a single source of truth of all state changes, enabling **event replay** to rebuild the application state or projections deterministically at any point in time. Each change record contains enough information (including before/after values or inverse operations) to support **rollbacks** via compensating entries [6]. Forensic investigations are facilitated by **Evidence Packs** attached to each change, capturing inputs, model outputs, logs, screenshots, and lineage metadata needed to understand why and how the change was made.
- **Integration with HITL Approvals and Observability:** The system integrates seamlessly with the human-in-the-loop **Approval Tray** (including multi-signature approvals) such that changes require the configured number of human approvals before apply. It also ties into our Observability framework by linking changes to trace IDs and logs. Every change is annotated with context (user or agent identity, trace/span IDs) so that one can trace a user action or agent decision through API calls and ledger entries [7] [8]. The evidence and lineage data will feed our existing evidence viewer "drawer" UI component, allowing one-click inspection of the proof attached to each change [9].

All components are designed with the **simplicity principle** in mind – the user experience remains a clear 3-step flow (propose, approve, apply) and developers will interact with a stable set of TypeScript interfaces (for hashing, appending, applying, etc.) that match our current stack. Under the hood, the ledger's

cryptographic and concurrency features operate transparently, so that agents and users do not need to perform extra steps beyond the usual workflow. In the sections below, we detail the architecture and integrity mechanisms, forensic tooling, handling of failure modes, and the reference implementation (code modules, schema, CLI, tests, runbooks, CI integration) for this Change-Set Ledger system.

Architecture Design

Per-Entity Change Streams & Ordering

Each **entity** (or aggregate record) will have its own logical change stream, ensuring that all modifications to a given entity are applied in a strict sequence. We maintain a global **ChangeSet** ledger table that records every proposed change, tagged by the target entity (e.g. `entity_type` and `entity_id`). Within a single entity's stream, changes are applied in deterministic order (by an auto-incrementing sequence or timestamp ordering with conflict resolution) to preserve consistency. This design aligns with event-sourcing best practices, which emphasize ordering of events that affect a specific entity's state [5]. It prevents interleaving changes from different entities from causing confusion, while still having a unified ledger for system-wide integrity verification.

To implement deterministic ordering, the system assigns each applied change a **monotonic sequence number** (for example, the `change_set_applied.seq` primary key) at insertion time. This sequence reflects the global order of committed changes. In addition, each change can carry an `entity_version` – a version number for that entity which increments with each change – to detect concurrent modifications. When applying a change, we enforce that the entity's current version matches the expected version in the change request (optimistic locking). If it doesn't, the ledger can reject or delay the change (to be retried after conflict resolution). This approach ensures that, for instance, if two agents concurrently propose edits to the same record, one will succeed first and increment the version, causing the second to detect a stale version and avoid a lost update [10] [11]. In effect, it addresses the classic time-of-check to time-of-use (TOCTOU) problem by performing a final check on the entity's state at the moment of apply.

Changes from different entities do not require strict global ordering relative to each other (they can be processed in parallel), but **within each entity** the ledger guarantees serializable updates. For example, if agent A proposes change X to entity `E1` and agent B proposes change Y to entity `E2` at the same time, both can be appended and applied without conflict. However, if both propose changes to the *same* entity `E1`, the first one to be approved and applied will increment `E1`'s version, and the second will detect a version mismatch and be handled (e.g. queued for retry or marked for review). The ledger thus achieves *ordered delivery per resource* similar to patterns used in reliable systems [12], ensuring each resource's event sequence is consistent even under concurrency.

Idempotency Keys and Duplicate Prevention

To guard against duplicate operations (whether due to user re-submission, network retries, or agent errors), the ledger uses **idempotency keys** on all change submissions. An idempotency key is a client-generated unique identifier attached to a state-altering request [4]. The ledger's append logic checks this key against past entries to decide how to handle the request:

- If a new change comes in with a key that was **never seen** before, it is treated as a fresh proposal and recorded normally.

- If a change comes in with a key that **matches an existing** change (meaning it's a retry of the same intended operation), the system will **not insert a duplicate**. Instead, it can return the existing ledger entry or ensure the operation is a no-op beyond the first application [13]. This guarantees that the operation will have been applied at most once, no matter how many times it was requested.
- If two *different* operations somehow arrive with the same idempotency key (which indicates a client error or misuse), the ledger treats that as an error – the second operation is rejected, since the key is supposed to uniquely identify a single operation [13].

We implement this by making the `idempotency_key` a **unique column** in the `change_set` table (scoped globally or per entity, depending on system needs). The `appendChange()` function (in `append.ts`) wraps the insert in a database transaction. If the same key is concurrently used, one transaction will succeed and the other will get a unique constraint violation – which we catch and interpret as "this operation was already recorded". The append function can then simply fetch and return the existing record for the caller, giving them the same result without duplicating the effect. This pattern follows established practices in robust API design [4], ensuring callers can safely retry operations without risk.

To handle the scenario of **simultaneous duplicates** (e.g. two identical requests hitting at nearly the same time on different app servers), we incorporate a **synchronization mechanism** to avoid a race where both check and both insert. The simplest approach is to rely on the database's atomicity: the first insert wins, the second fails on the unique index and is caught as described. Alternatively, we could implement a short **mutex or lock** keyed by the idempotency key (at an application level using a Redis lock or at DB level using `SELECT ... FOR UPDATE` on a key index table) to serialize duplicate attempts [14]. In this design, we lean on the database uniqueness constraint for simplicity – it is efficient and sufficient in practice.

Optimistic concurrency: In addition to idempotency for duplicates, the design uses an **optimistic locking** strategy for general concurrency control on each entity's state. Each entity can have a `lock_version` or `last_change_id` field that is included in the change-set proposal. The apply operation will verify that the stored version matches; if not, it aborts the apply with a concurrency error. This is akin to the approach used in financial ledger systems to avoid phantom reads and race conditions [15] [11]. The combination of idempotency keys and optimistic version checks provides robust **retry and race protection**: agents can retry failed operations safely, and conflicting concurrent edits are detected and can be resolved deterministically (either by queuing, merging, or manual intervention).

Workflow Integration (Draft → Approve → Apply)

The ledger is woven into the existing workflow as follows:

1. **Drafting a Change:** When an autonomous agent or tool wants to make a write (e.g. modify a record), it does **not** write directly to the primary database. Instead, it creates a *draft change-set* via the `appendChange()` API. This records the proposal in the `change_set` table with status "draft" (pending approval). The draft includes details like the target entity and the proposed modifications (e.g. fields and new values), along with the agent's identity and idempotency key. A corresponding Evidence Pack is attached at this point (capturing the agent's input, reasoning, and any relevant artifacts) to support the approvers in decision-making. If a similar draft was recently recorded (same idempotency key), the system will recognize it as duplicate and not create a new entry.

2. **Approval Tray & Multi-Sig:** The new draft appears in the **Approval Tray** UI for human reviewers. Based on configured policy, it may require single or multiple approvers (multi-signature). The `approvals` table tracks each approver's decision (e.g. who approved, when, and any digital signature or comment). If multiple approvals are needed, the change remains pending until the required number (or specific roles) have approved. We enforce multi-sig rules by checking the count/role of entries in `approvals` before allowing an apply. The Approval Tray UI and backend logic ensure that this step remains a simple "Review diff & evidence -> click Approve" process for each human, preserving the 3-step UX (draft, approve, apply) and not overburdening the user despite the underlying complexity.

3. **Apply (Execution of Change):** Once approved, the change can be **applied** to the actual system state. The apply operation is performed via the `applyChange()` function (in `apply.ts`). This function will first promote the change from draft to applied in the ledger: it generates a new entry in the `change_set_applied` ledger table referencing the original change-set, and computes the cryptographic hash linking it to the previous applied entry (more on hashing in the next section). The apply function then executes the change's effect on the primary database (or downstream systems) – for example, updating the record's fields as proposed. This update is done within a database transaction *together* with the insertion of the ledger entry, to guarantee atomicity (state change and ledger change occur or fail together). If any part of the transaction fails, it's rolled back so we never have a ledger entry without the actual state change or vice versa (avoiding partial apply scenarios). On success, the change-set's status is updated to "applied" and timestamped. The Evidence Pack may be enriched with any runtime data (e.g. logs collected during execution, confirmation IDs from external services, etc.) and is sealed. At this point, the change is considered committed and visible in any projections or user interfaces. The ledger's append-only hash chain now incorporates this change, meaning any tampering with this record or prior ones would be detectable by verification.

4. **Rejections & Cancellations:** If a draft change is rejected by a human (or times out waiting for approval), it remains in the `change_set` log as a record of a proposed-but-not-applied change. We mark its status as "rejected" and optionally include a reason. Rejected changes are **not inserted into the** `change_set_applied` **hash chain**, so they do not affect the cryptographic integrity of applied state. (They can be considered as *non-events* for state, though they are retained for audit/history.) If needed, we could also append a special "rejection" event in the ledger to explicitly signal that a particular proposal was voided, but typically logging the decision in the approvals table and keeping the draft record is sufficient. During replay of events to rebuild state, the replay tool will skip any change_sets that never reached applied status. This way, the chain of applied events reflects exactly the sequence of actual state modifications, while the system still retains a full audit trail of all attempted changes (applied or not).

5. **Observability Integration:** Each change-set (draft and applied) carries metadata that integrates with our observability and tracing system. For example, we attach a **trace ID** or span ID from our distributed tracing to the change record [8]. This allows us to correlate ledger events with application logs and metrics. In practice, if an agent ran a complex sequence of tasks leading to a change, all those tasks likely share a trace ID – by logging that ID in the ledger, an investigator can later retrieve all relevant logs (via our log aggregator) for the timeframe and components that participated in that change. Similarly, we record the agent's or tool's name and any session/user IDs involved, creating a lineage from a user action → agent decision → ledger entry [7]. This end-to-end lineage answers the "who/what/why" for each change: *Who* initiated it (user or which AI agent), *What*

was changed, and *Why* (with evidence like model outputs or references to source data). The evidence viewer UI (the "drawer" component) can use this information to display a rich, interlinked story for the change. For instance, a human can click a change in the UI and see in the drawer: the diff of the data changed, the agent's rationale (from the Evidence Pack), a screenshot of the state before/after if captured, and links to logs or other changes that preceded it in the lineage graph. This tight integration ensures the ledger enhances transparency without disrupting normal workflows.

Overall, the architecture ensures that agents operate within a governed change pipeline. They **emit proposed changes** rather than directly mutating state, and the ledger + approval system mediates these changes in a controlled, auditable manner. The design is compatible with our existing **multi-agent architecture** in that each agent simply invokes the ledger API when it wants to make a change. The ledger abstracts away concurrency concerns and logging – from the agent's perspective, it just submits a change and later gets a confirmation when applied (or a rejection). The underlying database is Postgres with Prisma ORM, so we define the schema in Prisma and use Prisma client transactions to implement the atomic append-and-apply operations. The approach works within our current stack (Node/TypeScript, Prisma/Postgres) and is deployable on our infrastructure (the ledger is just another module in our codebase and another set of tables in our Postgres database, plus some background jobs for anchoring and verification that we can run on our servers or CI).

Integrity: Hash Chains, Checkpoints, and Anchoring

A core feature of the Change-Set Ledger is **tamper-evidence** – the ability to detect if any change logs have been altered, deleted, or inserted out-of-band. We achieve this by leveraging cryptographic hashes to link entries in an append-only **hash chain**, and by recording periodic **checkpoints** (snapshots of the chain's state) that can be anchored to external systems.

Hash Chain vs. Merkle Tree Approach

For each applied change event, the ledger computes a SHA-256 cryptographic hash over that event's data and the hash of the **previous** event in the chain. By storing this hash in the `change_set_applied` record, we form a **hash chain**: each entry `n` has a field `hash` = H(data_n || hash_{n-1}), with hash_0 defined as a fixed genesis value. This means no entry can be altered or removed without breaking the chain of hashes. If an attacker modifies an old record (or inserts a fake record), the hash stored in the subsequent record won't match the recomputed value, and verification will fail, revealing the tampering. Hash chains are a well-known method (used in blockchains and secure logs) to ensure that the log is append-only and retrospectively immutable [16].

We considered both **hash chains** and **Merkle trees** for our integrity structure. A Merkle tree (as used in systems like Certificate Transparency logs) allows efficient proofs of inclusion and consistency by aggregating entries' hashes into a tree whose root is a short digest [16]. One could take all changes in a given period and compute a Merkle root. In contrast, a simple hash chain is linear: each new entry references the prior hash, without a branching structure. We opted for the **hash-chain approach** for its simplicity and because our primary verification operations are full-log or contiguous-range verifications, which a chain handles well. In a private, centralized ledger (with one authority writing the log), a straightforward hash chain is sufficient for tamper-evidence [1]. Merkle trees shine when you need to prove inclusion of an event to an external client without revealing the whole log, or to parallelize verification, but in our case the ledger is mainly verified internally or by auditors with full access. The

overhead and complexity of maintaining a dynamic Merkle tree (and storing proofs for each entry) weren't justified given our usage. That said, the two approaches are not mutually exclusive – one can treat the append-only log as a "blockchain" where each block's hash covers a batch of events (like a Merkle root for that batch) and still chain those, but we decided on one-event-per-block for granularity.

Concretely, when a change is applied, the `hash.ts` module will create a **canonical representation** of the change data and context, then take a SHA-256 digest. The data hashed for an event includes: - Key identifying fields (entity type and ID, change type), - The payload of the change (e.g. which fields were changed from what values to what new values), - Metadata like timestamps, the `idempotency_key`, the user/agent who initiated it, and a reference to the Evidence Pack (e.g. a hash or pointer to evidence blob manifest), - The **prev_hash** (the hash of the previous applied event, or a constant for the first event).

All this is serialized in a deterministic way (e.g. JSON with sorted keys or a Protocol Buffers message) to ensure that given the same content we always get the same hash digest (canonicalization prevents incidental differences like key order from altering the hash). The resulting SHA-256 digest becomes the `hash` field of the new record. We also store the `prev_hash` in the record for completeness (though it's implicit in the chain, it's useful for quick traversal and verification of a segment).

By verifying the chain from the beginning, we can prove the integrity of the entire log up to the latest entry. Each record's hash is essentially a commitment to the entire history before it (because it incorporates the previous hash, which in turn incorporates its predecessor, and so on). If the chain length is N and an attacker tries to alter record *k*, they would have to recompute hashes for all records k+1…N to cover their tracks. They cannot do that retroactively if we have anchored the earlier hashes externally, as described next.

Daily Checkpoints and External Anchoring

While a hash chain protects the log *internally*, a determined attacker with full database access could conceivably rewrite history by recomputing all hashes from some point onward (especially if they control all copies of the database). This is where **checkpoints** and external anchoring come in. The idea is to regularly take a snapshot of the ledger's state (in practice, the latest hash value at a point in time) and record it in a place that the attacker cannot quietly alter. Later, we can compare the current log against these checkpoints to detect inconsistencies.

We will implement **daily checkpoints**: at the end of each day (UTC midnight or a configured time), the system computes the hash of the latest applied change-set and stores it in a `daily_checkpoint` table along with the date. The `daily_checkpoint` entry might also include the index of that last change (so we know up to which record this checkpoint covers). This provides an immutable daily digest of all changes up to that day. The checkpoint table itself is small and append-only (one row per day). It is protected by the same database security, but an attacker altering it would be obvious (they'd have to remove or change entire entries).

However, to truly be tamper-evident to external observers, we **anchor** these checkpoints in an external medium outside of our primary database control [1] [17]. Several anchoring methods are possible: - **Public Blockchain Anchor:** Publish the daily hash on a public blockchain (e.g. Bitcoin or Ethereum) as a transaction or in a smart contract. For example, we could use an Ethereum transaction with the hash in calldata, or use Bitcoin's OP_RETURN field to store the digest [18]. Later, anyone can look at the blockchain

and retrieve our posted hashes. If our internal ledger is ever questioned, we refer to the public record; any discrepancy means the internal data was changed after-the-fact. This is a highly tamper-resistant option (the security of the anchor relies on the blockchain's integrity). The downside is cost and complexity (though posting a small hash daily is quite feasible). - **External Trusted Service:** Use a service or separate server to hold the checkpoints. For instance, send the daily hash to an immutable log service (like Amazon QLDB or an ElasticSearch index with WORM storage), or simply email it to a set of auditors, or even print it out (the old-school way was newspaper ads [19] !). The principle is to make it infeasible for an attacker to purge or alter all copies of the checkpoint. Even printing the hash daily provides strong evidence, because no one can later alter all distributed print copies [19] . - **Hardware or Offline Storage:** Write the checkpoint to a **WORM** (Write-Once-Read-Many) medium or an HSM (Hardware Security Module) which appends it to a secure audit log. WORM storage (like write-once optical media or certain S3 bucket modes) ensures once written, it cannot be changed [8] . This approach was noted as expensive for frequent short data like daily logs [20] , but is an option.

Our implementation can leverage an **automated job** `ledger-anchor` that runs daily and performs the anchoring. Initially, we might integrate with a public blockchain via an API (if cost allows) or use a simpler route like posting to a company-owned *audit webhook* (for example, a minimal service that simply stores these hashes in a secure append-only file or third-party timestamping authority). The `ledger-anchor` CLI can output the hash and allow hooking up any external process. We will also store the anchored hash and the transaction/reference ID in the `daily_checkpoint` table (fields like `anchored_at` timestamp and `anchor_txid` or reference). This way, we know which external proof corresponds to which day's hash (e.g., the Ethereum transaction ID).

By using external anchoring, even a **colluding admin scenario** is mitigated: if someone with full DB privileges tried to rewrite the log, the **publicly anchored hashes would not match** the recomputed ledger, exposing the attempt [2] . For example, suppose an admin tries to delete a change from yesterday. Today's anchor (posted on e.g. Ethereum) includes the hash that was computed with that change present. If they remove it and adjust later hashes, they would end up with a different end-of-day hash for yesterday – which would not match what's on Ethereum. That discrepancy is irrefutable evidence of tampering. In other words, anchoring moves our integrity from just *tamper-evident* (detectable if one has an original reference) towards *tamper-resistant*, because the attacker cannot easily reconcile their tampered version with the external checkpoints [21] [1] .

Verification Process (Full and Range Verification)

We provide tooling to **verify** the ledger integrity either in full or over a specific range. The `ledger-verify` CLI uses the `verify.ts` module to perform these checks:

- **Full Verification:** Starting from the earliest change (or a known genesis hash), iterate through each `change_set_applied` entry in sequence, recompute the hash chain and compare it to the stored `hash` field. If all hashes match, the chain is intact. This also involves verifying that the `prev_hash` of each entry matches the `hash` of the previous entry. If any mismatch is found, the tool will flag the specific sequence number and details. Full verification ensures end-to-end integrity but can be time-consuming if there are millions of entries (we address performance below). Usually, this would be run periodically or as part of an audit, not on every transaction.

- **Range (Segment) Verification:** We support verifying a subset of the ledger, for example between two checkpoints or for a particular date range. This might be used when we suspect an issue in a certain window or want to validate that a backup subset is correct. Range verification works similarly: given a starting prev_hash (which could be from a checkpoint as the "anchor" for that range's start) and an ending expected hash (from an end checkpoint), we hash through that segment and confirm the end result. One can verify daily segments independently if daily checkpoints are trusted. For instance, if each day's end hash is anchored, one could verify each day's events against that anchor in parallel, rather than one huge chain – although verifying continuously through the chain is straightforward if performance permits.

- **Checkpoint Verification:** The verify tool also cross-checks the chain against the `daily_checkpoint` table and external anchors. For each checkpoint in our DB, it recomputes what the hash at end of that day *should* be (by hashing up to that sequence number) and compares it to the stored value. If there's a discrepancy, it warns that historical data may have been altered. Optionally, if connected to the anchoring medium (e.g., able to query a blockchain or our external store), it can retrieve the recorded hash and ensure it matches as well. This ties our internal verification to the external reference.

The **cryptographic strength** of SHA-256 ensures that even a single-bit change in any record will produce a completely different hash chain thereafter. The chance of an undetected change (colliding with a valid hash) is astronomically low. Thus, the combination of hash chaining and external anchors gives high confidence in ledger integrity. This approach mirrors known solutions for secure audit logs [16] [1] and is inspired by designs like AWS QLDB (which uses a digest chain and allows verification) and Google's Trillian (Merkle trees for transparency logs), though scaled to our use case.

Below is an example diagram of an append-only ledger with a hash chain and external anchoring, similar to our design:

Figure: Append-only ledger chain with cryptographic hashes. Each new block (change-set) includes the hash of the previous block, forming a tamper-evident chain. The latest hash can be periodically checkpointed (e.g., daily) and anchored in a Trusted Storage or public blockchain, ensuring any attempt to tamper with past records can be detected by a mismatch with the anchored checkpoint.

In the figure above, if any block in the chain is modified, the hash links "break" and the discrepancy would be caught when comparing to the anchor in trusted storage [1]. Our implementation follows this model closely, with the `Trusted Storage` being represented by our daily external anchor.

Forensics & Lineage Features

Evidence Packs and Schema

Every change-set comes with an **Evidence Pack** – a bundle of data that provides context and proof for the change. The goal is to make each change auditable and explainable after the fact, which is crucial in a system where AI agents are making autonomous changes. The Evidence Pack for a change may include: - **Inputs**: The input that led to the change (e.g. a user prompt or a data file the agent processed). - **Model Outputs/Decisions**: For AI-driven changes, the relevant sections of the model's output or reasoning. For

example, if the agent is an LLM that generated text which was used to update a field, the generated text or summary is included. - **Logs and Traces**: Any logs from the agent's run or system trace that show the steps taken. This could include debug info, error messages encountered and handled, etc. We automatically attach the system trace ID as mentioned, so using that ID one can pull a timeline of events from our logging system [8] . We might also capture specific log excerpts if they are directly relevant (depending on how verbose we want the evidence). - **Screenshots or Data Snapshots**: If the agent operates in a UI or processes images, screenshots or copies of relevant images can be evidence. For example, if the agent is to change a product description, we might snapshot the old description or UI state before and after. - **Before/After State**: The ledger itself stores the "before" and "after" values of the changed data (for structured data changes). In the evidence manifest, we include a clear diff or summary of what changed. This is helpful for approvers (to see what they're approving) and for auditors (to quickly grasp the change). - **Lineage metadata**: Pointers to any prior changes or events that directly influenced this change. (This ties into lineage graph, discussed below.)

The Evidence Pack is stored in the `evidence` table and possibly external blob storage. We design it such that it can be packaged as a **manifest JSON file** plus associated binary blobs. For instance, for each change we can generate a `evidence_manifest.json` that lists all evidence items (with type, description, and a URI or hash), plus any meta info like the agent name, timestamps, etc. The blobs (images, large text logs, etc.) can be stored in cloud storage (e.g. an S3 bucket) with their URIs in the manifest. The `evidence.ts` module provides functions to **pack** and **unpack** these. Packing might involve uploading files to storage, encrypting them if necessary, and creating the manifest. Unpacking would fetch and decrypt files given a manifest.

Encrypted Blob URIs: We assume evidence may contain sensitive information (user data, internal prompts, etc.), so we allow for encryption. Each evidence blob can be encrypted client-side (by the agent or by our server when storing it). We then store an `encrypted_key` (for example, an envelope-encrypted key using a KMS) or some key reference along with the blob's URI. Only authorized systems (or the evidence viewer with proper permissions) can retrieve and decrypt the blob. The manifest might store just the URI and a hash of the plaintext (for integrity). The actual decryption keys would be managed securely (e.g., stored in a vault and accessible only to the evidence viewing service with audit logs). This ensures even if someone got access to the raw storage, they cannot read evidence without authorization [22] . Our evidence schema in the database (`evidence` table) could have columns like `type` (e.g. "screenshot" or "log"), `uri` (pointer to the blob), `content_hash` (SHA-256 of the plaintext for tamper detection), and `encrypted_key` (maybe encrypted via KMS).

For small evidence items (say short texts or diff summaries), we might store them directly in the manifest JSON or as fields in the DB for convenience. Larger items (images, long logs) remain as external references. The manifest JSON itself can be stored as a text field in the evidence table or generated on the fly by gathering all evidence rows for that change.

Our Evidence Packs are designed to be **compatible with the existing evidence viewer/drawer UI**. The viewer expects certain content: likely it knows how to display images given a URL, how to format JSON or text logs, etc. By using standardized fields and types (for example, marking an image with a thumbnail URL, or a log with a text snippet and a link to full content), the front-end can render them appropriately. The manifest JSON essentially acts as an interface: the UI can fetch it (through an API that authenticates and then possibly decrypts any needed pieces) and then iterate through entries to display. Because we include

lineage and manifest metadata, the viewer could also present links such as “Source: Change #1234” or “Next related change: #1236”, creating a navigable chain of related changes.

Lineage and Causal Links

Beyond just capturing evidence for an individual change, the system maintains a **lineage graph** that connects changes and evidence across the multi-agent workflow. The `lineage_edges` table serves to represent directed relationships between nodes (where nodes can be change-sets or evidence items). For example: - If Change B was generated as a direct consequence of Change A (e.g., agent 2’s action was triggered by the result of agent 1’s change), we record an edge `Change A -> Change B`. - If a particular Evidence item (say a report or analysis) was used as input for a change, we link `Evidence X -> Change Y`. Conversely, if a change produces an artifact that is stored as evidence and then that artifact is used in another change, edges can capture those dependencies. - In a human-in-the-loop scenario, if a user’s decision or an approval is influenced by some earlier output, we can capture that link as well.

The lineage graph enables **forensic traceability**: one can start from a given change and traverse backwards to see what led to it, or forwards to see what it influenced. This is extremely useful in a multi-agent system where a chain of automated steps and approvals might lead to a final outcome. For instance, consider an AI agent proposes an update to an article (Change 5), which was based on a summary generated by a previous agent (Change 3) that in turn got its data from an earlier data ingestion (Change 1). All these would be linked. An investigator can navigate this chain to audit the whole process, verifying that each step was sound.

We implement lineage tracking by inserting records into `lineage_edges` at appropriate points: - When a draft change is created by an agent as a result of some prior change, the agent’s code or orchestrator will call something like `recordLineage(parentChangeId, childChangeId)`. - If an Evidence Pack item comes from a prior change, we record that too. For example, if the evidence includes a summary text that was generated as part of an earlier change’s output, we create `parent_type = 'ChangeSet', parent_id = [that change], child_type = 'Evidence', child_id = [evidence id]`. - The schema is flexible: it has generic parent and child references, with type labels so it can link different kinds of objects (changes, evidences, perhaps external events). Typically, parent could be a ChangeSet or an external event, child could be ChangeSet or Evidence.

One specific lineage element we include is the **trace/span ID** (mentioned earlier) which effectively ties into the observability traces. While not a direct graph edge in our database, it acts as a *foreign key* into our logging system: an investigator can use the trace ID to fetch logs showing chronological relationships (like “action X happened, then action Y happened”). Combined with our explicit lineage_edges, we get both *synchronous* relationships (one change directly caused another) and *broader context* relationships (these events happened in the same workflow trace).

Lineage data is stored in a simple form that can be easily queried to visualize the graph. We might create a custom query or CLI to output a chain of events given a start point, or use a graph visualization tool if needed (even a quick script to output Graphviz or similar). The aim is to support questions like *“How did we end up with this change?”* and *“What was the chain of custody for this data?”*. This aligns with principles of audit & lineage in financial systems where you must link user actions to system state changes [7]. Our implementation ensures **“audit & lineage for every move”**, stitching together user actions, agent operations, and ledger entries via trace IDs and explicit edges [7].

Replay and Deterministic Rebuild

One powerful byproduct of storing all changes (with their data and inverses) is the ability to **replay** the events to reconstruct system state at any time. This is essentially an event-sourcing pattern: instead of relying on the current database snapshot alone, we can recompute it by starting from an empty state and applying each change in order [23] [24]. Our `ledger-replay` functionality serves multiple purposes: - **State Recovery:** In case of a catastrophic failure or corruption in the primary database, we can use the ledger to rebuild the entire state on a fresh instance. Because every applied change is recorded with its full details, replaying them in sequence should yield the same final state as the original. This can replace traditional restore-from-backup if needed, or be used to validate backups. - **Deriving Projections:** We might spin up new projections or data warehouses by replaying the log. For example, if we want to build a new search index or analytics database from scratch, we feed it the stream of changes. This ensures the derived systems are consistent with the source of truth and can be rebuilt whenever the schema changes (avoiding complex migration scripts) [24] [25]. - **Point-in-Time Queries:** We can answer "what was the state on date X?" by replaying up to the changes before that date. This is useful for forensic investigations or debugging issues that occurred in the past. Rather than storing full snapshots for every day, we can recompute on the fly. (We may also periodically store materialized snapshots for efficiency, see below.) - **Testing and Simulations:** Developers can use replay to populate a local dev database with realistic data by applying all events (or a subset). This yields a consistent state that mirrors production (minus secrets). Also, one can simulate "forks" by branching the event log – e.g., take all events up to yesterday, apply to a test environment, then try out a hypothetical new sequence to see outcomes, without affecting the real ledger.

The `replay.ts` module will implement deterministic reapplication of events. Each change event knows how to apply itself (and we ensure those operations are **pure and deterministic** – e.g., if the event says "set field A to value X", that is exactly what replay does, regardless of external context). We avoid any non-deterministic actions during replay: for instance, if an event originally involved a random number or time, the actual outcome is stored in the event, so replay uses that stored outcome rather than regenerating randomness. Similarly, if an agent used AI to generate text, the final chosen text is stored, so replay doesn't call the AI again (it just inserts the text). In essence, every non-deterministic decision is turned into data, captured in the change record or evidence, making replay a straightforward deterministic transformation of state.

We maintain **inverse operations** for each change to enable rollback or selective undo. In practice, when applying a change, we record the *before state* of any modified element. For example, if a change updates a field from "foo" to "bar", the change record will contain "before: foo, after: bar". If a change inserts a new record, the inverse would be "delete that record"; if a change deletes a record, inverse contains the entire deleted record data to allow re-insertion. By having these inverses, we could automate a rollback by creating compensating change events [6]. Indeed, the Azure architecture guide notes that in event sourcing, *"to undo a change, add a compensating event to the event store"* [6]. We follow the same principle – no destructive reversal, but rather an additional event that counteracts the effect. The ledger can provide a convenience to generate such an event if needed (for instance, if an admin decides to revert a particular change, we generate a new change entry that uses the stored before-state to set things back, with appropriate evidence noting it's a rollback of a previous change).

One challenge is if we want to "rewind" and reapply with different logic (say we discovered a bug in how changes were applied). In that case, we wouldn't literally replay to update production (because that would repeat the bug); instead, we might *fix the code* and then replay all events onto a blank state to get a

corrected state, then cut over to that. This is what the original event sourcing vision allows – you can retroactively change how events are interpreted and rebuild a new projection [24]. Our runbooks (discussed later) include a **re-mint procedure** for such cases.

To ensure replay is efficient, we will utilize **snapshots** and **partitioning**: - **Snapshots**: We can periodically take a snapshot of the entire state (or of large aggregates) after a certain number of events, and store that snapshot (in a backup or even in the ledger as a special event type). Then, replay can start from the latest snapshot prior to the target point, instead of from scratch, to save time. For example, if we snapshot the database state at the end of each month, then to reconstruct mid-August, we start from end of July snapshot and play events from August. Snapshots can be stored as part of Evidence Packs or in a separate snapshot store. Note that storing snapshots should be done carefully to not violate tamper-evidence: ideally we treat a snapshot as a derived artifact (with its own hash in evidence), not as a primary ledger entry that could break the chain. The snapshot's integrity can be ensured by including a list of event hashes it represents. - **Range selection**: The replay tool allows selecting a subset of events (e.g. only events of certain entity types or from certain date ranges) to rebuild only part of the system or answer a narrow question. If needed, we can parallelize replay by entity stream, since different entities' events are mostly independent (except where cross-entity consistency is needed, but our multi-sig and invariants ensure each event was valid in real-time).

The determinism and completeness of our ledger means **replays will always produce the same result given the same event log**. If there is any divergence, that indicates a problem (like a non-deterministic effect or an out-of-band change). We include assertions in the replay process to catch such issues – for example, we might recompute a hash of the final state and compare it to an expected hash if we had one. In a properly functioning system, rehydrating the events yields the current state exactly [26] [27]. This principle underpins the idea that the ledger is the primary source of truth, and the database is just a projection that can be discarded and rebuilt if needed [24] [28]. In practice, this gives us confidence to treat the production database as more flexible – if a schema migration goes wrong, we can reconstruct; if data is corrupted, we have a full history to refer to.

Rollback and Recovery

In forensic scenarios or remediation, one might need to **roll back** certain changes. There are a few modes of rollback: - **Point-in-time rollback**: restoring the entire system to a previous state (undoing everything after a certain time). This can be done by replaying events up to that time into a fresh instance and then swapping pointers or otherwise directing reads to that instance. It's essentially a time-based fork. However, in an always-forward ledger, we typically wouldn't remove events; rather, we'd use this for analysis or to reset a test environment. In production, instead of rolling back in place, one would issue compensating events as needed (so the ledger reflects what happened: e.g. "we backed out change X by doing Y"). - **Selective rollback (Compensating change)**: If one specific change (or a set) is deemed erroneous and we want to undo its effect while preserving later correct changes, we create new change events that counteract the wrong ones. For each change to undo, we look at the "before" data stored in that change and apply it as a new change. This new event goes through the same draft→approve (perhaps fast-tracked if urgent) and gets appended to the ledger as a legitimate correction. This way, the ledger does not lose history (the wrong change is still there, followed by a compensating change) [6]. During audits, both will be seen – which is acceptable and even desirable for transparency. The system state ends up as if the change never happened (if no other dependent events), or at least the data is corrected, and a clear audit trail shows how and when the correction happened. - **Operational recovery**: In cases of partial failures (discussed in failure

modes), we might need to complete or revert changes that were only partially applied. Our design's use of transactions prevents partial database commits, but consider an external side effect (e.g., an email sent to a user) that occurs right after the ledger entry. If the external side effect fails, we might issue a compensating change or mark the ledger entry with an error status and attempt the side effect again. Alternatively, if the external side effect succeeded but the ledger entry failed to record (less likely since we do ledger first, but imagine a network partition after side effect), we have a "ghost" change (not logged). Our observability and idempotency could catch this: perhaps the side effect system returns an ID which our evidence captured, so we could later insert a ledger entry for completeness if needed (with a special marker).

From a forensic perspective, we ensure **no deletion of ledger data**. Even if something is "undone", it's done via an additional record, not by scrubbing the original. This is critical for legal defensibility – auditors prefer seeing that a mistake happened and was corrected, rather than an attempt to pretend it never happened. Our runbooks (below) detail the procedure for handling corruption or needed rollbacks using these primitives.

Finally, all forensic-related data (evidence, lineage, etc.) is kept for as long as necessary to meet compliance. We may implement **retention policies** if needed (for example, maybe evidence images are kept encrypted for X years), but any deletion would itself be a logged event (with justification) to maintain trust.

Handling Failure Modes and Edge Cases

Even with a robust design, we must consider various failure scenarios and how the ledger system copes with them. Here we address **TOCTOU issues, clock skew, partial failures, and backfills**, among other potential problems.

Time-of-Check to Time-of-Use (TOCTOU) Consistency

TOCTOU issues arise when a system makes a decision based on a condition, but by the time it acts, the condition has changed. In our context, this could happen if an agent generates a change based on an earlier state, but the state is different by the time we apply the change. For example, an agent drafts a change "increase salary by 5%" for employee X, checking that X's current salary is $1000. If another change (perhaps manual) updated X's salary to $1100 in the meantime, a 5% increase should be $55 not $50; the original calculation is outdated. If we blindly apply it, we effectively under-pay relative to intention.

Our approach to TOCTOU is multi-layered: - **Precondition Checks in Change Data:** We allow a change-set to include an optional precondition (for instance, the expected previous value or version of the data). The apply logic will verify this precondition against the live state at apply time. If it fails, the apply is aborted or flagged. In the example, the change-set could carry "expected current salary = 1000". On apply, we see current salary is 1100, which doesn't match, so we **do not apply** and instead mark the change for reevaluation. This prevents silently applying changes on stale assumptions. - **Optimistic Locking (Versioning):** As described, each entity has a version number. The draft records the version it saw. When applying, if the entity's version has advanced, we know some other change occurred. The system can then either reject the change or, ideally, re-compute it. In simple cases (like a pure overwrite), rejection is fine; in cases where the change is a function of current state, we might want to recalc the difference. For now, we implement rejection or require manual intervention on version conflict to keep determinism (the agent could be invoked again to recompute its proposal on the new state, or a human could decide). - **Serializing by Entity (as discussed):** Because we queue changes per entity, often the conflicting scenario might be

naturally avoided – if one change is pending on X, maybe we do not approve a second until the first is applied, depending on workflow. However, since multiple proposals can exist concurrently for the same entity, we lean on the above mechanisms.

Thus, the ledger will catch TOCTOU races at apply time. Our UI could then notify the approver or agent that the change couldn't be applied due to intervening changes. If appropriate, it could auto-create a revised draft adjusting to the new reality (perhaps using the same idempotency key or a linked lineage to the original). The evidence pack for a change will also include context such as the state at the time of proposal (like "salary was 1000 when this recommendation was made"), so if a mismatch occurs, forensic analysis can see why the change ended up not applied.

Clock Skew and Ordering by Time

Using timestamps in a distributed system can lead to ordering anomalies if clocks are not perfectly synchronized. In our system, each change will have timestamps (drafted_at, approved_at, applied_at). If one server's clock is off, it might log a slightly incorrect time, but our *ordering* of events does not rely purely on those clocks. We use the database server's timestamp for ordering if needed (ensuring all events use a consistent time source) or, more robustly, the monotonic sequence number. The `seq` (or `id`) in `change_set_applied` effectively encodes order. We will not sort or anchor events by timestamp alone, we always use the insertion order. This avoids issues where, say, an event applied slightly later could have an earlier timestamp due to skew.

However, **clock skew** could affect daily checkpoint boundaries: e.g., what if an event at 23:59:58 on one node is recorded at 00:00:02 on the DB due to delay? It might end up in the next day's checkpoint. This is minor (just a boundary choice) but we define the cutoff by DB's notion of date. The daily checkpoint job will likely query "all events up to <today's date 23:59:59>". If clocks are off, an event might appear with a timestamp slightly into the next day. But since sequence is monotonic, even if an event "belongs" conceptually to day 1 but timestamped day 2, it doesn't break the chain – it will just be included in day 2's anchor. For integrity, that's fine. For semantics, it's rare and acceptable.

To mitigate skew further, we ensure NTP sync on all servers and rely on Postgres's timestamp when possible. Another approach is to use **Lamport timestamps or logical clocks** for ordering, but given we have a single ledger writer (the database) that serializes commits, a global logical clock is inherently present (the sequence ID). So clock skew is not a major integrity threat, just a logging nuance.

Partial Apply Failures

We have designed the apply process to be **atomic** where possible: the ledger entry and the actual data mutation occur in one database transaction. This means the classical partial failure (e.g., ledger says applied but data not actually changed, or data changed but ledger not recorded) is prevented at the database level. Using Postgres, we will perform, for example, `INSERT INTO change_set_applied ...; UPDATE business_table ...;` in one transaction. If any constraint fails or the database crashes mid-transaction, neither action takes effect.

However, not all operations are within a single database. Consider if applying a change involves: - Updating the database, and - Sending a notification or calling an external service (maybe to index the data or send an email).

If the external call fails after the DB commit, then we have a ledger entry and DB state updated, but the side effect didn't happen. That's a kind of partial failure from the system perspective. We handle this by **reordering side effects** and using outbox patterns: - Ideally, perform critical external side effects *before* committing the ledger, but that runs the risk that the ledger might then fail. Instead, we can do the opposite: commit the ledger+state, then have a reliable mechanism to do the external action, and if it fails, record that failure and possibly schedule a retry. This way, the ledger truth (state changed) is preserved. The external side effect can be linked to the ledger entry (e.g., evidence might note "email sent to user: success" or "failed, will retry"). We could even model the side effect as another type of event in the ledger (like an asynchronous event). - We incorporate a **retry mechanism for side effects**: e.g., a background job that sees a ledger entry with a "pending email" flag in evidence, attempts to send, and on success updates the evidence status. If ultimately fails, a human can intervene. This ensures consistency eventually, and the ledger still reflects exactly what was applied.

Another partial failure scenario is if applying a batch of changes (like an admin approves 10 changes at once) and the transaction fails halfway. But since we treat each change independently (one transaction per change apply), that won't occur – either each change commits fully or not at all, and they don't get batched unless intentionally.

System crashes or power loss during append/apply: Because database transactions are used, if the system crashes mid-apply, the transaction will roll back on recovery. The agent or orchestration would likely time out and maybe retry the apply, which is safe thanks to idempotency keys. We should be prepared that an apply could be executed twice in the worst case (if the first time it committed but the response didn't get back to the orchestrator). But again, idempotency would catch that (the second time, the change_set would already be marked applied, or the idempotency key would exist, so no duplicate effect).

Backfills and Historical Data

Backfilling refers to inserting events into the ledger after the fact, often to record historical data or changes that occurred outside the normal flow. There are a few scenarios: - We deploy this ledger system anew, and we want to import *past changes* (perhaps from log files or other records) to have a complete history. - An offline batch operation was done (say a script directly modified the DB for maintenance) and later we want to add those changes into the ledger for completeness. - Data from another system (legacy or external) is integrated, and we want to capture it as if it were ledger events.

Backfilling is tricky in an append-only chain since we cannot insert at arbitrary positions in the past without breaking the hash chain. Our approach is to **append backdated events** with special markers: - We can create a series of backfill events that are appended in the present, but carry a timestamp indicating their original occurrence time. They will have to be ordered in our chain at the point of insertion (the current end). To avoid confusing ordering, we may group all backfills with a special entity or tag. For example, we might do a one-time import: create a "genesis" block for initial state and a series of "historical" blocks for each past event, then a checkpoint, then proceed with live events. This is only feasible if we pause live writes during import. - More practically, if we need to add a historical record after go-live, we will insert it at the current tip of the ledger (so chain integrity remains). In its data, we mark `event_time = actual time in past` separate from the `applied_at` (which will be now). This way, it doesn't affect state (if the state change already happened earlier, we might not reapply it). In fact, for a pure backfill (for audit only), we might insert the ledger record with a flag that it should not be applied to state (because state already includes it). We could incorporate logic in replay to skip applying those or ensure

idempotency (e.g., the event might say “this is an import of an update that’s already reflected in state, do nothing on replay except verify consistency”). This is a bit complex but manageable with careful use of metadata. - If the backfill is initial data, one strategy is to treat the current database as the result of a bunch of past changes and backfill those changes. This could be done by diffing the current state against a baseline and manufacturing events. Alternatively, simply take a **snapshot event** at the start. For example, we can have event #0: “Initial state as of 2025-01-01 – inserted 1000 records” as a kind of synthetic event that establishes baseline. This event can be the parent of all actual changes after that. If we anchor its hash, we effectively anchor the initial state integrity as well.

Because backfills can confuse auditors if not clearly documented, we will keep a **clear record or manifest of any backfill operations**. The ledger can include evidence for backfilled changes explaining that they were inserted post-hoc to reflect earlier actions. We also likely require that backfill operations go through an approval (to ensure oversight, since they might be done by an admin user).

Another angle is performing **backfills on derived systems**: the Reddit excerpt notes that with an event log, *“Backfills are no longer weekend projects… select a replay window, start the job, and the log streams the exact slice you need”* [29] . In our case, if the core ledger has everything, we can populate other systems for specific historical ranges easily. This is more about using the ledger for feeding data warehouses, etc., which is a positive aspect.

In summary, our ledger design does not natively support inserting events in the middle of history, but we handle historical data by either capturing it as initial snapshot events or appending proxy events now that document past occurrences. The chain remains intact (since we never truly insert in the middle), and daily checkpoints/anchors continue normally from when the ledger went live. For events before ledger launch, we may anchor the genesis snapshot or simply rely on trust for prior data (depending on requirements).

Other Failure Modes

Some additional considerations: - **Approval delays or misses:** If an approval is significantly delayed, the draft sits pending. There’s a chance that by the time it’s approved, it no longer is valid (TOCTOU scenario) – which we already handle by version check. So those would be caught. We may also implement an expiry for draft changes to avoid applying very stale proposals (for safety). - **Multi-sig conflicts:** Suppose one approver approves a change and another rejects. The policy might dictate what happens (perhaps majority or all must approve). We ensure that only when the policy is satisfied does the apply proceed. If rejected by any required party, the change is marked rejected. There’s no race condition because the apply will only happen after a flag in `change_set` says “approved = true” and we set that in a transaction once conditions met. - **Database migration issues:** Changing the schema of ledger tables (e.g., adding a column) doesn’t break the chain integrity because the hash covers the data; adding a column with default doesn’t change old data’s hash calculation as long as we ignore it or treat default as null in hashing. If we do need to change how data is stored (say we compress some fields), we have to be careful that our hash computation still works over a canonical form. Ideally, we never alter the content of historical records. Any migration that would do so (e.g., data format change) should be done by appending events that update things, rather than in-place. If absolutely needed (like we realize a certain field wasn’t hashed but should be), we might recalc the whole chain and anchor that, but that undermines the immutability a bit. So we strive to design upfront to avoid such changes. - **Scaling and performance**: As the ledger grows, inserts might become heavy if not indexed properly, and verify or replay operations become lengthier. We mitigate with indexing by (entity_id) for quick per-entity queries, and perhaps partitioning the applied table by time (to make it easier to manage

older data). Verification of 1 million events, as required, needs to be optimized – a straightforward Python/TypeScript loop hashing 1M items is okay (1M SHA-256 operations might take on the order of seconds to a minute in optimized C, but in TS might be slower; we can chunk it or use a native addon if needed, or simply accept a minute or two runtime which is fine for an offline check). We will test and possibly implement a streaming verification where most heavy lifting is done in a single SQL query (e.g., using Postgres cryptographic functions) if performance is an issue. However, given hardware advancements and that this is not on the critical path, we have flexibility.

Having covered the design and how it handles many edge cases, we proceed to outline the **implementation deliverables**: the code modules, database schema, CLI tools, tests, runbooks, and CI integration that will make this design a reality.

Reference Implementation and Deliverables

This section provides a blueprint of the implementation, including module breakdown, schema definitions, CLI utilities, testing approach, operational runbooks, and CI gates. The code will be written in TypeScript (Node.js) within a package named `/packages/ledger/`. We will use Prisma for database access and migrations, and the solution will run on Postgres. All components are aligned with the simplicity and interface stability requirements – meaning we expose clean functions and CLI commands that integrate naturally with existing code, and we don't break existing database models outside adding new tables and relations.

1. Ledger Modules (TypeScript)

`hash.ts` **– Hashing & Canonicalization:** This module defines how we compute cryptographic hashes for ledger entries. It includes functions to canonicalize a change-set object to a byte string and to compute the SHA-256 hash chain. We use Node's `crypto` library for SHA-256. For canonicalization, we can use a stable JSON stringify (ensuring keys sorted, etc.). Example code snippet:

```
import { createHash } from 'crypto';

// Deterministically serialize the essential parts of a change-set for hashing
function canonicalizeChange(change: ChangeSetRecord): string {
  // pick and sort keys to avoid variability
  const obj: any = {
    id: change.id,
    entity: `${change.entityType}:${change.entityId}`,
    // Include fields that should be immutable in content
    before: change.before,
    after: change.after,
    // ... other relevant fields
    timestamp: change.appliedAt?.toISOString(),
    prevHash: change.prevHash || ""
  };
  // Sort object keys recursively if needed (assuming small depth)
  const sortedJSON = JSON.stringify(sortObjectKeys(obj));
```

```
  return sortedJSON;
}

export function computeHash(change: ChangeSetRecord): string {
  const canon = canonicalizeChange(change);
  return createHash('sha256').update(canon).digest('hex');
}

// Example usage:
// When preparing a new applied change record:
record.prevHash = lastRecord.hash;
record.hash = computeHash(record);
```

The `hash.ts` also exports helpers to validate hash values (e.g., checking length, format) and perhaps a function to chain multiple entries (used in verification). It ensures that for a given `ChangeSetApplied` entry, everyone computing the hash will get the same result. By encapsulating this in one module, any future change to the hashing scheme (like adding a field) can be done in one place (though such changes should be avoided or done in backward-compatible ways to not break old links).

`append.ts` **– Idempotent Append of Proposed Changes:** This module handles the creation of new change-set drafts. It exposes a function like `appendChange(draft: DraftChangeInput): Promise<ChangeSet>` which will insert a row in `change_set` table if not already existing. The logic: - Generate a new unique `id` for the change (Prisma will do this via autoincrement). - Populate fields from `draft` input (entityType, entityId, proposed data diff, createdBy, etc.), set status = 'draft', createdAt = now. - Assign an `idempotencyKey` (from input; if none provided by caller, we can generate one or require it – likely the caller provides a UUID). - Attempt to insert via Prisma. If a UniqueConstraintViolation error on `idempotencyKey` occurs, do a lookup: return the existing change (and do not create a duplicate). - If insertion succeeds, also insert any evidence or lineage records that were provided along with the draft. For evidence, likely we first call `evidence.pack()` to store blobs and get URIs, then save records referencing the new change ID. - Return the created ChangeSet object (possibly with evidence sub-objects attached if needed).

We ensure this runs in a **transaction** if multiple tables are involved (Prisma allows transactions via `$transaction`). Since a draft creation might involve writing evidence and lineage, those inserts should all-or-none as well. But if evidence upload to S3 fails mid-way, we can safely abort before DB commit and propagate error, so that's fine (and maybe try again).

Pseudocode for append (in Prisma style):

```
async function appendChange(input: DraftChangeInput) {
  const changeData = {
    entityType: input.entityType,
    entityId: input.entityId,
    data: input.data, // JSON diff or content
    createdBy: input.author,
    idempotencyKey: input.idempotencyKey,
```

```
    status: 'draft'
  };
  try {
    const result = await prisma.$transaction(async (tx) => {
      const newChange = await tx.changeSet.create({ data: changeData });
      // Insert evidence records if any
      if (input.evidence && input.evidence.length > 0) {
        for (const ev of input.evidence) {
          // Suppose evidence already contains URI (upload done prior) and type
          await tx.evidence.create({ data: {
            changeSetId: newChange.id,
            type: ev.type, uri: ev.uri, contentHash: ev.hash, encryptedKey:
ev.encryptedKey
          }});
        }
      }
      // Insert lineage edges if provided
      if (input.parents) {
        for (const parent of input.parents) {
          await tx.lineageEdges.create({ data: {
            parentType: parent.type, parentId: parent.id,
            childType: 'ChangeSet', childId: newChange.id
          }});
        }
      }
      return newChange;
    });
    return result;
  } catch (err: any) {
    if (isUniqueConstraintError(err, 'idempotencyKey')) {
      // fetch existing
      return prisma.changeSet.findUnique({ where: { idempotencyKey:
input.idempotencyKey } });
    }
    throw err;
  }
}
```

This demonstrates ensuring idempotency (the unique index triggers an error we catch). We also link evidence and lineage in one go.

`apply.ts` – **Apply Changes, Inverse Ops and Snapshots:** This module manages transitioning a change-set from draft to applied. Key tasks: - **Precondition/Version Check:** If the `ChangeSet` has an associated expected version or condition, verify it against current DB state (outside of ledger). For example, if `change_set.data` includes an expected old value, confirm the DB's actual value matches. If not, throw an error (or mark the change as needing rework). - **Execute Update:** Apply the intended modification to the

domain data. This could be a simple SQL update/insert/delete on the target table, or a call to some domain service. Because we prefer to do it in the same transaction as ledger insertion, we likely perform direct SQL (through Prisma) for the data change as part of the transaction. We make sure to capture the *before* values for inverse. - **Inverse Operation:** Based on the type of change, compute its inverse and store it in the `change_set_applied` record or associated structure. We can store the `before` and `after` states in JSON columns. For a deletion, `before` contains full record, `after` is null; for insertion, `before` null, `after` is full record; for update, both are partial (only fields changed ideally, though could store full before/after snapshots of the object). - **Insert into Applied Ledger:** Create a `ChangeSetApplied` entry with a new `seq` number (auto increment), linking to the original change via `changeSetId`. Set `prevHash` to the last applied entry's hash (which we get by querying the max seq or last record – careful to do this in a way that locks the last row to avoid two concurrent apply both thinking they have the same prevHash). Compute the `hash` for this new entry using `hash.ts` logic. Also set `appliedAt = now` and `appliedBy` (if this is executed by an agent or system user, identify it; could be the approver or a system service account). - **Update ChangeSet status:** Mark the original `change_set` record as applied (e.g., set status 'applied', or we can rely on the existence in applied table to infer it's applied). We might still update it for convenience (and fill an `appliedAt` in it too). - **Post-Apply Hooks:** Possibly trigger any side-effects (like sending notifications) outside the DB transaction (see partial failure handling). Could also generate a snapshot if this is a snapshotting checkpoint (though snapshots likely via separate job). - **Return result:** The function returns the `ChangeSetApplied` record or some summary (including the new chain hash, etc.).

We ensure this entire sequence for each change is one DB transaction (except external calls). By doing the select last hash + insert new within a transaction serialized, we prevent race conditions on computing `prevHash`. If two apply transactions run concurrently, one will naturally commit first, the second will see the updated last hash. In practice, to ensure sequential prevHash safely, we might use a locking read: e.g., `SELECT hash FROM change_set_applied ORDER BY seq DESC LIMIT 1 FOR UPDATE` as part of the transaction (which will lock that row until commit). This will serialize concurrent inserts — essentially we get a behavior akin to a singly-linked list being updated by one transaction at a time. The performance impact of that minimal locking is negligible given single-row contention only when truly concurrent applies happen. Another approach is to designate the DB sequence of `seq` as the order and trust it – one can simply use the *previous sequence* by `newSeq = nextval(); prev = newSeq-1`. But that doesn't guarantee the previous seq was committed; better to actually fetch the last committed hash.

Pseudo-code combining things:

```
async function applyChange(changeId: bigint, user: string):
Promise<ChangeSetApplied> {
  return await prisma.$transaction(async (tx) => {
    const draft = await tx.changeSet.findUnique({ where: { id: changeId },
include: { evidence: true } });
    if (!draft) throw new Error("Change not found");
    if (draft.status !== 'draft') throw new Error("Change already applied or
invalid status");
    // Precondition check if any (assuming draft.data.precondition stores
expected version or value)
    if (draft.data.precondition) {
      const ok = await checkPrecondition(draft);
```

```
      if (!ok) {
        await tx.changeSet.update({ where: {id: changeId}, data: { status:
'error', error: 'Precondition failed' }});
        throw new Error("Precondition failed, not applying change");
      }
    }
    // Execute domain update and capture inverse
    const { inverseOp } = await executeDomainUpdate(tx, draft);
    // Lock last ledger entry (to get prevHash)
    const lastApplied = await tx.changeSetApplied.findFirst({ orderBy: { seq:
'desc' }, lock: { mode: 'for update' } });
    const prevHash = lastApplied ? lastApplied.hash : GENESIS_HASH;
    // Insert into applied ledger
    const applied = await tx.changeSetApplied.create({
      data: {
        changeSetId: draft.id,
        prevHash: prevHash,
        hash: '',  // placeholder
        appliedAt: new Date(),
        appliedBy: user
      }
    });
    // Compute hash after insertion (need seq and everything)
    const computedHash = computeHash({ ...draft, ...inverseOp, prevHash,
appliedAt: applied.appliedAt, id: applied.seq });
    await tx.changeSetApplied.update({ where: { seq: applied.seq }, data: {
hash: computedHash } });
    // Mark draft as applied
    await tx.changeSet.update({ where: { id: draft.id }, data: { status:
'applied', appliedAt: applied.appliedAt } });
    return { ...applied, hash: computedHash };
  });
}
```

The above is conceptual; in practice, we might compute the hash fully before inserting if we know all data including seq. But seq is auto increment; we could also get the next sequence via `nextval` function without inserting. Alternatively, insert with dummy hash, then update as shown (two queries). The slight time when dummy hash is in DB is within same transaction (no other sees it), so that's fine.

The `executeDomainUpdate` would contain logic specific to the entity type. Possibly the change record has enough info (like table name, record id, changes) to apply generically. But more likely, we'd have a switch or hooks for each entity type. E.g., if change.entityType == "User", call userService.updateUser(...). Since this is a reference implementation, we might assume simple direct DB operations: if change.data indicates an update of certain fields on a certain table, we perform that. For capturing `inverseOp`, we gather the "before" state from the DB (maybe the SELECT ... FOR UPDATE on the record to update as well). That could be done prior to modifying it.

`verify.ts` **– Verification of Ledger Integrity:** This module implements the verification logic for both full and partial checks. It might have functions like: - `verifyAll(): VerificationResult` – recompute the entire chain from the start and verify hashes. - `verifyRange(fromSeq, toSeq): VerificationResult` – verify a subset, possibly requiring a starting prevHash (if fromSeq is not 1, we can supply what the prev_hash at fromSeq should be, e.g., from a checkpoint). - `verifyAgainstCheckpoint(day): boolean` – verify all up to that day's last seq matches the stored daily checkpoint hash.

Under the hood, verify functions will typically: - Query batched results from `change_set_applied` in order (maybe stream to avoid loading huge data at once). - For each consecutive pair, check that `record.prev_hash` equals the previous record's hash (consistency of chain links). - Recompute `hash` of each record via `hash.ts` (taking care to reconstruct the same canonical input that was originally hashed). We may need the original change data to do that – which is accessible via the relation to `change_set` and possibly evidence. But in our design, `change_set_applied` doesn't store the full data, it references `change_set`. So verification will likely join or sequentially fetch the draft data for each applied event to recompute. We could store some digest of essential data in the applied record to avoid heavy join, but to be safe, we might join because the change_set might be huge JSON. Alternatively, a pragmatic approach: trust that if the prev_hash chain is intact and the last hash matches checkpoint, then the data is intact. Strictly speaking, one should hash the data too, since an attacker with DB access could change both the data in change_set and adjust the hashes accordingly to cover it up. But because the chain links through the hash that presumably included the data content, if they change data, they'd have to change the hash of that entry and then all subsequent ones which would conflict with checkpoint. Unless they collude with updating checkpoint table too. But external anchor stops that. So verifying just chain link consistency plus anchor might suffice without re-hashing full content. But to be thorough, we will re-hash content as well. - Check final hash vs known anchor if doing full verification.

We'll implement it straightforwardly in TypeScript for clarity, but potentially one could push it into the database (some DBs support doing cumulative hash via window function, etc.). Simpler: just retrieve all `change_set_applied` sorted by seq, loop in JS computing rolling hash and compare to stored. 1M entries is feasible in Node if careful (1e6 SHA256 ops – might take maybe a few seconds in optimized C, but Node's crypto might handle it pretty fast since it's implemented in C++ under the hood).

Pseudo-code:

```
async function verifyAll(): Promise<{ ok: boolean, error?: string }> {
  const stream = prisma.changeSetApplied.findMany({ orderBy: { seq: 'asc' },
include: { change: true } });
  let prevHash = GENESIS_HASH;
  let count = 0;
  for await (const entry of stream) {
    count++;
    if (entry.prevHash !== prevHash) {
      return { ok: false, error: `Chain link broken at seq ${entry.seq}:
expected prevHash ${prevHash} but found ${entry.prevHash}` };
    }
    // Recompute hash from data
```

```
    const fullChangeRecord = { ...entry.change, appliedAt: entry.appliedAt,
seq: entry.seq, prevHash: entry.prevHash };
    const expectedHash = computeHash(fullChangeRecord);
    if (entry.hash !== expectedHash) {
      return { ok: false, error: `Data hash mismatch at seq ${entry.seq}:
expected ${expectedHash} got ${entry.hash}` };
    }
    prevHash = entry.hash;
  }
  // Optionally compare last hash with the latest daily checkpoint or anchor
  const lastAnchor = await getLatestCheckpoint();
  if (lastAnchor && prevHash !== lastAnchor.hash) {
    return { ok: false, error: `Latest hash ${prevHash} != recorded checkpoint $
{lastAnchor.hash}` };
  }
  return { ok: true };
}
```

(Using `.findMany()` for 1M might need pagination or streaming to not consume huge memory; Prisma can stream or we could do it in chunks.)

`evidence.ts` – **Evidence Pack Assembly:** This module offers utilities to manage evidence packs. Its responsibilities: - When creating a draft change, take any provided evidence input (raw files, data) and **save them** to our evidence storage: - It may handle uploading files to S3 or writing to disk, and compute their hashes. - It should generate an `encryptedKey` if we encrypt the file. Possibly using AWS KMS: generate a data key, encrypt file with it, then store the data key encrypted by KMS in `encrypted_key`. - Provide back the URI (like `s3://evidence-bucket/…`) and content hash to store in DB. - Provide a function `packEvidence(changeId)` that queries all evidence entries for that change and produces a single packaged object (like a zip or a JSON manifest plus binary data). This might be used by the CLI or export features. - Conversely, `unpackEvidence(manifest or changeId)` to retrieve and verify evidence. For example, for the evidence viewer: given a changeId, fetch evidence records, for each: - download or access the blob from URI, - if encrypted, decrypt using the `encryptedKey` (which likely requires calling KMS or using a stored key). - verify the blob's content hash matches what we stored (guarding against accidental corruption in storage). - return the blobs or a structured set of files ready to serve to front-end (maybe in memory or a temporary URL).

While some of this goes beyond a simple reference implementation (setting up S3 etc.), we outline the structure. Perhaps the actual code uses an abstraction `StorageProvider` so we can plug in local disk for dev and S3 for prod.

We will also enforce evidence **schema validation** here: e.g., if we expect evidence type "screenshot" to be an image file under 5MB, we can check that. Or at least ensure the `type` field is one of allowed values. This plays into our CI tests as well.

Example snippet for evidence upload:

```
import { readFileSync } from 'fs';
import { encryptData } from './crypto'; // custom or AWS SDK call
import { uploadToS3 } from './storage';

interface EvidenceItemInput { type: string; filePath: string; }

async function saveEvidenceItems(changeId: bigint, items: EvidenceItemInput[]):
Promise<void> {
  for (const item of items) {
    const data = readFileSync(item.filePath);
    const { encryptedData, encryptedKey } = await encryptData(data);
    const hash = createHash('sha256').update(data).digest('hex');
    const uri = await uploadToS3(`evidence/${changeId}/$
{path.basename(item.filePath)}`, encryptedData);
    await prisma.evidence.create({ data: {
      changeSetId: changeId, type: item.type, uri, contentHash: hash,
encryptedKey
    }});
  }
}
```

(This is simplified; in practice we'd stream files etc.)

`replay.ts` – **Replaying Changes to Rebuild Projections:** This module provides the logic for the `ledger-replay` tool, allowing us to rebuild state. It might not directly manipulate our production DB (unless we are doing a real recovery). Instead, it could output SQL or apply to a target database/connection provided.

Key tasks: - Connect to the source ledger (likely the same database or a backup of it). - Optionally connect to a target database (for rebuilding state) or create in-memory objects for a dry run. - Read all `change_set_applied` events in order (or up to a given point). - For each event, apply its changes to the target. This requires logic similar to `applyChange` but in a simplified form (no need to re-check approvals or idempotency, just trust the event and do it). - If the event is an insertion, insert the record (ensuring to use the same primary key if needed or regenerate? Ideally, events include the primary keys). - If update, set the field to the value in event.after. - If delete, remove the record. - We should consider foreign keys and constraints on target: since original operations presumably were valid, replaying in order will maintain consistency too (assuming no out-of-order). - If replaying to a different schema (like if we changed schema since, might have to adapt events – out of scope for now). - Provide options: e.g., `includeRejected` false by default so we skip non-applied events. Possibly allow filtering by entity type (to rebuild only one service's data).

We'll incorporate deterministic ordering by using the seq. If multiple events for different entities can be replayed in parallel, one could conceive a performance improvement by parallelizing. But to keep it simple and deterministic, we do serial for now (which is fine for correctness; performance can be improved if needed by parallel on independent entity streams, but that requires careful partition detection).

A pseudo-code outline:

```
async function replayAll(targetDB: PrismaClient) {
  const events = await prisma.changeSetApplied.findMany({ include: { change:
true }, orderBy: { seq: 'asc' } });
  for (const ev of events) {
    const change = ev.change;
    // We stored before/after in change.data for example.
    switch (change.type) {
      case 'update':
        await targetDB[change.entityType].update({
          where: { id: change.entityId },
          data: change.after  // assuming after is an object of fields
        });
        break;
      case 'insert':
        await targetDB[change.entityType].create({ data: change.after });
        break;
      case 'delete':
        await targetDB[change.entityType].delete({ where: { id:
change.entityId } });
        break;
    }
  }
}
```

This implies that `change.data` holds something indicating type and the diff (before/after). We might have structured it differently in actual implementation (maybe an operation field). But the idea stands. If some events were already applied in target (like if target is not empty), we might need to handle conflicts or start from empty always (preferable).

We'll also ensure the replay function can stop at a given seq or date if needed (for point-in-time recovery). And potentially verify the final state's hash if we have a method to hash entire DB state (not usually, but one could hash each table rows – too heavy typically; instead we rely on our ledger hash being correct as indicator state is correct given initial conditions).

2. Database Schema (SQL/Prisma Models)

We define the database tables needed. Using a Prisma schema for clarity:

```
// Prisma schema representation
model ChangeSet {
  id            BigInt    @id @default(autoincrement())
  entityType    String
  entityId      String     // or BigInt/UUID depending on domain
```

```
  status        String     // 'draft', 'applied', 'rejected', 'error'
  createdBy     String
  createdAt     DateTime   @default(now())
  appliedAt     DateTime?  // set when applied
  idempotencyKey String    @unique
  data          Json       // JSON blob of change details (diff, type, etc.)
  evidence      Evidence[]
  approvals     Approval[]
  appliedEntry  ChangeSetApplied? @relation(fields: [id], references:
[changeSetId])
  // Note: one-to-one relation: appliedEntry will exist if status = applied.
}

model ChangeSetApplied {
  seq         BigInt    @id @default(autoincrement()) // ledger sequence
  changeSetId BigInt    @unique // 1-to-1 with ChangeSet (applied changes)
  appliedAt   DateTime
  appliedBy   String
  prevHash    String    // store as hex string or binary
  hash        String
  change      ChangeSet @relation(fields: [changeSetId], references: [id])
}

model Approval {
  id          BigInt   @id @default(autoincrement())
  changeSetId BigInt
  approver    String
  approvedAt  DateTime  @default(now())
  signature   String?   // optional digital signature or just an approver note
  decision    String    // 'approved' or 'rejected' (in case we track individual
votes)
  ChangeSet   ChangeSet @relation(fields: [changeSetId], references: [id])
}

model Evidence {
  id          BigInt   @id @default(autoincrement())
  changeSetId BigInt
  type        String
  uri         String    // location of evidence blob (could be long, e.g. S3
URL)
  contentHash String
  encryptedKey String?  // e.g., base64 of encrypted symmetric key, if used
  metadata    Json?     // any extra metadata (filename, size, etc.)
  ChangeSet   ChangeSet @relation(fields: [changeSetId], references: [id])
}

model LineageEdge {
  id         BigInt  @id @default(autoincrement())
```

```
  parentType String
  parentId   BigInt
  childType  String
  childId    BigInt
  // We might not have formal foreign keys, since parent could be in different
table
  // Alternatively, parentType/childType could be enums 'ChangeSet'/'Evidence',
etc.
}

model DailyCheckpoint {
  id          BigInt  @id @default(autoincrement())
  day         Date    @unique
  hash        String
  anchoredAt  DateTime?
  anchorRef   String?  // reference to external anchor (tx id or file)
}
```

Some notes on this schema: - We use `BigInt` for IDs and sequence to allow a large number of entries without overflow. - `ChangeSet.data` is a JSON where we'll store details of the change operation (e.g., type of operation, perhaps the diff or new values, and possibly the "before" state for quick reference if we want). We could break those out into fields, but JSON gives flexibility if different entity types have different shape of changes. - Inverse operations could either be stored in `data` as well (like `before` and `after` sub-objects) or one could add explicit columns or a separate table for before/after. To keep things in one place, we might put them in JSON under `ChangeSet.data` at creation time for proposals (though for a proposal, "before" might not be known or might be checked at apply). - `ChangeSetApplied` references `ChangeSet` in a one-to-one (with unique constraint on changeSetId). This is how we separate draft vs applied record while linking them. Alternatively, we could have merged them into one table with status, but then the hash chain might include entries that aren't applied. By having applied separate, our chain is exactly the `ChangeSetApplied` table's sequence of hashes. - `LineageEdge` does not have explicit foreign keys because parent/child can point to different tables. We can enforce via app logic that parentType must be 'ChangeSet' or 'Evidence' and then use parentId accordingly. This flexible approach means we can link an evidence to a change or change to change as needed. - `DailyCheckpoint.day` unique ensures one per day. `hash` is likely stored as hex string (64 hex chars for SHA-256) or could be base64. Storing as binary (BYTEA) is possible too; we chose string for readability in e.g. admin queries. - We ensure to index or use primary keys on things we query often: e.g., `ChangeSetApplied.seq` is primary key (clustered index, iterating in order is efficient). We might want an index on `ChangeSet.entityId` if we query by entity, and on `LineageEdge.parentId` / `childId` to traverse lineage quickly.

This schema will be added via a Prisma migration. None of the existing tables are altered except perhaps we might add a field to domain tables if we want to store a version number for optimistic locking (e.g., each entity table might get a `lastChangeId` or `version` field). That wasn't explicitly requested, but could be considered. If we do, we'd add migrations for those as well (with triggers or app logic to maintain them).

3. CLI Tools

We will provide command-line tools to manage and inspect the ledger system, implemented as small Node scripts or as part of a larger CLI (maybe integrated into our existing ops CLI if any). These tools help with verification, anchoring, and replay:

- `ledger-verify`: This tool runs an integrity check on the ledger. It can be invoked to verify the full chain or a specific range. Usage examples:
- `ledger-verify --full`: Verify the entire ledger from the beginning to the latest entry, and report "OK" or the point of failure.
- `ledger-verify --since 2025-09-01`: Verify from the first entry on or after Sept 1, 2025 to the end (this would use the last checkpoint before that date as a trusted base, if available).
- `ledger-verify --from-seq 10000 --to-seq 11000`: Verify a specific sequence range, maybe used when focusing on a suspected segment.
- Internally, this command uses the `verify.ts` module. It will output the results to stdout, e.g., *"Ledger verification PASSED. All 12345 entries intact (last hash = abcdef...)."* or an error like *"Verification FAILED at seq 9876: prev-hash mismatch"* or *"hash mismatch (data tampering suspected) at seq X"*.
- The tool can also cross-verify daily checkpoints. Perhaps `ledger-verify --daily` will loop through each daily checkpoint and ensure the chain segments match each day's hash, printing any discrepancies for specific days (this is essentially doing full verification but aligned with days).

- We intend to integrate this into CI or at least as a regular admin operation. It's lightweight enough to run on a schedule (depending on ledger size – verifying tens of thousands of hashes is quick; if it's millions, might take longer, but we could still run nightly or weekly with no issue, or rely on checkpoints for intermediate trust).

- `ledger-anchor`: This tool computes the current daily checkpoint and emits/stores it. Typically run as a scheduled job at 00:00 UTC daily (or after the last change of the day). It can also be run ad-hoc to anchor the latest state.

- Function: Query the latest `ChangeSetApplied` entry, get its `hash` and `seq`. Compute (if needed) a Merkle root or something, but in our case, just use that hash as the day's digest (since it inherently represents all prior changes).
- Insert a new row into `daily_checkpoint` with `day = yesterday` (if anchoring for the day that just ended) or `day = today` if we anchor at end of day. We store the hash.
- Connect to external anchor service to publish this hash. The specifics depend on which service:
    - For blockchain: call a script or API to send the transaction containing the hash. This might return a transaction ID.
    - For an external log: maybe call a REST endpoint of an audit service.
    - For a simple approach, we could output the hash with some signature to a file or console and rely on an operator to post it publicly.
- Update the `daily_checkpoint` record with `anchoredAt = now()` and `anchorRef = <reference from external>`.
- Example usage: `ledger-anchor` (with perhaps env vars configured for blockchain keys etc.). Could also allow manual invocation with an explicit date: `ledger-anchor --date 2025-09-27` if we needed to re-anchor a specific day.

- On success, the tool prints *"Checkpoint for 2025-09-27 recorded. Hash: XYZ, anchor ref: ABC123"*. If using blockchain, the ref might be a tx hash which can be shared.
- Security: The anchor tool should ideally run on a secure machine, especially if it uses private keys to sign a transaction. It will likely be part of our infrastructure (a CI job or a backend process, not something users run). We ensure it has access to needed keys (maybe via environment variables or a KMS).

- This tool ensures *regular anchoring*. If it fails one day, we have a runbook entry to handle that (see runbooks).

- `ledger-replay`: This tool allows replaying the ledger events to reconstruct state or exports. Possible usage:

- `ledger-replay --target newdb_url`: Connect to a blank database at `newdb_url` (connection string), create schema if needed (or assume same schema), then apply all changes in order. This effectively clones the current state into the new DB. Useful for setting up staging environments or recovering production on a new server.
- `ledger-replay --to-sql dump.sql`: Output SQL statements that would reproduce the state. For example, it could go through each change and output an `INSERT` or `UPDATE` statement for each. Or more efficiently, it could directly output the final state: since applying all changes yields the final state, the tool might just as well output the final content of each table. But that requires building full state in memory or using intermediate DB. It's simpler to do the sequential updates approach. However, for large data sets, that SQL might be huge. Still, it's an option for portability. We might skip text dump and stick to direct apply.
- `ledger-replay --since 2025-01-01 --target partial_db_url`: Replays only events after a certain date to bring an out-of-date system up to sync, etc. Or `--from-seq X` to start at a certain event.
- We also support a dry-run: `ledger-replay --verify-only --to-seq N` which would simulate applying events to see if any constraint violations or unexpected differences occur. But since events were applied in production, unless the schema diverged, it should be fine.
- The implementation uses `replay.ts` logic. It should use the same business rules as original applies, but since we stored final outcomes, it's mostly straightforward. If some events depend on external services (which they shouldn't, as all state changes are internal), we ignore those aspects.
- After replay, the tool can optionally compute a checksum of important data and compare with something. For example, it could compute the hash chain of the new DB and compare with the ledger (which would match by design if all events applied, but maybe to double-check).
- Output example: *"Replayed 12345 events to target DB. Final state matches ledger up to seq 12345."* Or if any errors encountered (like target schema missing a column), it logs those.

The CLI tools will be integrated into our monorepo, likely with entries in `package.json` scripts or a dispatch from a central CLI. They can be tested in CI by running them on a test database.

4. Testing and Benchmarking

We will develop a comprehensive test suite for the ledger package. Key testing strategies:

- **Unit Tests for Hashing and Idempotency (Property Tests):** We'll write tests to ensure the hash computation is stable and correct. For example, given a fake change object, ensure that hashing it twice yields the same hash (property: deterministic hashing), and altering any field changes the hash. We can include known test vectors (maybe hash a known JSON and compare to expected hex). For idempotency, simulate calling `appendChange` twice with the same idempotency key and confirm we get one record and no duplicates. Also test that two different idempotency keys produce two distinct records.

- **Simulated Race Condition Tests:** We can simulate concurrent operations using either multi-threading (in JS we might use `Promise.all` to fire parallel requests) or by interleaving operations manually. For instance:

- Launch two `appendChange` calls in parallel with the same key; verify only one succeeds and the other returns the same result or an error that we handle.
- Launch two `applyChange` calls for the *same* draft (this shouldn't normally happen, but if it does, one should fail due to status or DB constraints).
- Launch two `applyChange` for different drafts at nearly the same time and ensure the final chain has both with correct prevHash linking (one should have prevHash of the other, no duplicate prevHashes). This tests our locking on `prevHash`. We can also inspect that the `seq` numbers are consecutive without a gap (which they will be by autoincrement).

- We might also simulate an agent hitting the system with the same idempotency key twice, expecting the second to retrieve the first result.

- **Integrity and Corruption Detection Tests:** Create a small ledger of, say, 10 changes, then:

- Tamper one of the hashes in the DB (simulate a corruption or malicious change: e.g., flip a bit in one `hash` field or in one of the data fields and adjust nothing else).
- Run `ledger-verify` and assert that it reports a failure at the tampered record [30]. For example, if we changed a hash, verification should catch a prevHash mismatch at the next record. If we changed data but left hashes, recomputing hash for that record should fail to match stored hash.
- Possibly test detection of a missing record: remove one and see that the chain breaks at that point (prevHash of following record doesn't match).

- Also test that if no tampering happens, verify passes and outputs "OK".

- **Performance Benchmark (1M-row verification):** We will include a performance test that generates a ledger of 1,000,000 dummy change entries and measures the time to run `verifyAll()` on them. This might be too slow to run in normal unit tests, so it could be a separate benchmark script. We can optimize verification logic if needed (for example, using Node's streaming APIs or writing parts in Rust via Neon if extreme). But likely, verifying 1e6 SHA256 hashes is doable within tens of seconds in Node, which might be acceptable for an offline tool. We aim for at least on the order of

seconds to a minute. If it's too slow, we consider verifying in chunks using checkpoint leaps (like verify each day and sum up).

- We can also test memory usage by streaming from DB so we don't load all 1M into RAM at once.
- The benchmark test might not run in CI (due to time) but we will conduct it manually and document the result in the report (e.g., "Verifying 1 million entries took 8.5 seconds on a 4-core machine", or whatever result).
- **Apply/Rollback Logic Tests:** Ensure that for a given applied change, the inverse operation truly reverses it. We can test on a simple entity stub:
- e.g., have an entity with value X, apply a change "add 5" (so X becomes X+5), then apply the inverse (X becomes X again). Use the ledger to store both and verify final state equals original. This test of inverses being correct will build confidence that our recorded `before` state is accurate.
- We can incorporate more complex scenario like sequence: apply A then B, then apply inverse of B then inverse of A, state should go back.
- This is essentially testing our ability to do compensating events properly (though in practice we wouldn't normally revert out of order without addressing dependent events, but test in isolation).
- **Multi-Agent Flow Test:** Simulate a realistic sequence:
- Agent proposes change1 (draft in ledger).
- Another agent proposes change2 concurrently for different entity.
- Approve both (simulate human approval by directly calling apply).
- Ensure both got applied with correct chain and no interference.
- Check that evidence and lineage are stored and can be retrieved.
- Run verify and ensure it passes.
- Try to replay those changes to an empty state and ensure the resulting state matches the one after apply (i.e., verifying replay consistency).
- **Concurrent Append & Apply Test:** Simulate an edge case: an agent posts a draft and almost immediately an automated process applies it (if some changes are auto-approved perhaps). If `appendChange` and `applyChange` overlap, ensure no issues. This might require careful ordering but we can mimic by making `appendChange` slow (like adding a delay after insert but before commit) and in that window calling `applyChange`, which likely will not find the draft or will error. Such a scenario might not happen because normally apply is only triggered after approval, but it's good to consider.

All tests will be run via our CI (Jest or similar). The **coverage goal is ≥ 85%**, which we will enforce with a coverage threshold. This ensures we've tested most branches (except maybe some external integration like actual blockchain calls might be stubbed).

5. Runbooks (Operational Procedures)

We prepare documentation/runbooks for various operational scenarios. These are step-by-step guides for engineers or operators to follow when performing maintenance or handling incidents. Key runbooks:

- **Replay & Re-Mint Procedure:** This is used if we want to rebuild the database (re-mint the state) or create a new environment from the ledger.
- **Scenario:** Suppose a bug in the code caused some derived data to be wrong. We fix the code and now want to recompute the correct state from scratch.
- **Procedure:**
    1. Provision a new database or clear existing one (depending if we want side-by-side or in-place).
    2. Initialize schema (run migrations).
    3. Run `ledger-replay --target=<newdb>` to apply all events to newdb.
    4. Wait for completion and verify it finished without errors. Compare counts of records or some key checks between old and new if possible.
    5. Optionally run `ledger-verify` on the new database's ledger tables just to double-check nothing weird happened.
    6. Swap the application to use the new database (or if in-place, ensure no other writes happened during replay by having had a freeze).
    7. Alternatively, if doing in-place "rewind and reapply":
    8. Dump current ledger events (which are up to date).
    9. Restore a backup of the database from before the bug (or an empty with just base schema).
    10. Use `ledger-replay` to apply events up to present.
    11. This variant is more complex, easier to do side-by-side then cut over.
    12. Once done, monitor system to ensure everything is correct.
- Also mention using this procedure for migrating to new infrastructure or populating staging: just run replay accordingly.

- Emphasize that since the ledger is source of truth, this is a reliable process that *"turns rollbacks into a one-command replay"* [31] [32] .

- **Ledger Corruption Response:** What to do if the ledger verification fails or we suspect tampering/corruption.

- **Detection:** If `ledger-verify` finds an error or if daily checkpoint doesn't match, treat it as a serious incident.

- **Steps:**

    1. Immediately pause all agent-driven changes (to not continue on potentially bad state). Possibly switch system to a read-only or limited mode.
    2. Identify the last known good checkpoint. For example, if yesterday's anchor was fine but today's hash is off, the issue occurred sometime after the last checkpoint.
    3. Use binary search on the ledger range if needed: verify halfway between known good and head to narrow down the exact event or time where things went wrong.
    4. Once the problematic entry or range is found, inspect it:
    5. Was a hash altered? Was data altered? Or is an entry missing?

6. Check database logs, audit logs, to see if there was any unauthorized access or errors.
7. If it's a simple corruption (bit flip, disk error) and we have redundancy, consider restoring that part from a replica or backup, then re-running verify.
8. If it's malicious (someone edited an entry), gather evidence (which fields changed).
9. Decide on remedy:
10. If a single entry's data got corrupted but we trust the rest, one could fix that entry by using evidence to restore its correct values and recompute its hash and all subsequent hashes. However, that means rewriting history for all subsequent entries (their hash field). That's not trivial and would break anchors (since anchor would differ). Thus, likely not allowed to alter historical chain (we would prefer to append correction events).
11. A better approach: Create a *fork* from the last good point. That is, spin up a new ledger from that checkpoint and replay all events after that using evidence (assuming evidence has everything). Essentially, treat everything after the corruption as potentially invalid and rebuild those events either manually or by automated means, then redeploy state.
12. This is similar to what a blockchain does after a attack – you might abandon a fork after a point.
13. In practice, easiest: restore DB to last known good daily checkpoint backup (assuming we keep a backup per checkpoint), then reapply events from that day forward from evidence (maybe by manual approval).
14. If that's impossible, at least mark in the ledger that from that point on, integrity is suspect.
15. If we recovered by replaying into a fresh ledger, archive the old ledger for analysis (don't throw it away). Publish a statement if needed (for compliance) that "on X date, an integrity issue was detected and the ledger was reconstructed from checkpoint Y; all operations after Y were re-executed" – this informs auditors that there's a discontinuity (though evidence ensures nothing lost).
16. Hardening: If the cause was malicious, rotate credentials, patch security, etc. If it was system error, consider additional integrity checks (maybe enabling page checksums in Postgres, etc.).

- This is a complicated procedure, and we highlight that **anchoring prevented silent tampering** – the fact we caught it is by design [1] [2]. Emphasize to always run verify when suspicious and use anchors as source of truth for where to cut off.

- **Evidence Retrieval SOP:** How to retrieve and inspect evidence for a particular change or set of changes, for audit or debugging.

- Provide steps for a few scenarios:
  - *Single change investigation:* Suppose an auditor wants to see everything about change #12345.
  - Using our admin UI or CLI: find the change by ID or filter (maybe `ledger-cli get-change 12345` which prints the JSON of change and evidence references).
  - Retrieve evidence: either via the UI (open the evidence drawer for that change, which will show the manifest and allow downloading attachments) or via CLI: `ledger-cli get-evidence --change 12345 --out /tmp/evidence12345/`. Our tooling would then fetch all evidence items for that change, save them to a folder (decrypting if needed), and produce a manifest file describing them.
  - Verify evidence integrity: ensure that the content hashes match (our tool does it automatically). If any mismatch, that's a red flag (could be storage bit rot or tampering).

    - Hand over to auditor: The evidence folder can be shared, or if sensitive, bring the auditor to a secure environment to view it.
    - Use lineage: If they need context, use `ledger-cli trace-lineage 12345` to get a list of preceding changes and their IDs, then retrieve those as needed.
    - *Bulk evidence for audit:* If preparing for a compliance audit covering many changes (e.g., all changes in Q4 2025):
    - Possibly have a script to gather all evidence manifests in that period. We might provide `ledger-cli export-evidence --from 2025-10-01 --to 2025-12-31 --out audit_Q4_2025.zip`. That would iterate changes in that range, collect evidence similarly, and zip them with a top-level index.
    - The runbook says to be mindful of data volume and to coordinate with IT if output is huge. Also maintain chain-of-custody: when giving evidence to external auditors, record what was given and ensure it's read-only (they can verify hashes too).
    - Emphasize that evidence often is sensitive (could contain raw user data or model outputs), so access is restricted. Only certain roles should run these, and possibly evidence data is encrypted such that even DB admins can't see content without access to keys. So retrieving evidence might involve contacting the security team to get a decryption key unsealed (depending on design).
- Also mention **performance**: retrieving an old evidence from cold storage may take time (if we tier older evidence to glacier, etc.). Plan accordingly with auditors.

- If any evidence is missing or corrupt (should not happen if process is correct), escalate (it means gap in our forensics). Typically, because we attach evidence at change time and never delete it, we should have it. If it was stored externally and that external store lost it, that's a problem to address (with backups or redundant storage).

- **Daily Checkpoint Triage:** Process to handle issues around daily checkpointing:

- If the `ledger-anchor` job fails on a given day (e.g., due to network issue or API failure):
    1. The next morning, Ops gets an alert (we should set up an alert if anchor job didn't log success by certain time).
    2. Run the anchor manually as soon as possible for the missed day. The tool might allow anchoring for a previous date by specifying `--day`. If the day has passed, we can still anchor the exact hash that was end-of-day (since our ledger still has it).
    3. Mark in our records why it was late (just for audit). If anchoring to blockchain, a late anchor is not ideal but acceptable as long as it's before any tampering (which presumably it is).
    4. If multiple days were missed (say the job was down a week), catch up sequentially for each day, or at least anchor the last day of the bunch with a Merkle root covering the range. But better to do each to maintain daily granularity.
- If a daily checkpoint hash mismatch is found (meaning the internal daily_checkpoint doesn't match recomputed chain – which implies tampering or DB inconsistency):
    - That is essentially the corruption scenario. Use the corruption runbook. The checkpoint triage here is just an early detection of that.
- If an anchor posting itself fails (e.g., transaction didn't confirm on blockchain):
    - Possibly double-post or use another chain as backup. Or if minor, just note and try next block. It's rare but possible e.g., Ethereum tx dropped due to low gas, so resend with higher fee. This is more of an anchoring system issue.

- If someone queries a past anchor (e.g., auditor wants proof of day X's hash):
    - Use our stored `daily_checkpoint.hash` and the external reference to retrieve the publicly anchored value, ensure they match, and provide the evidence (like showing the transaction on a blockchain explorer).
    - This is more an audit activity than triage, but our runbook would instruct how to do it quickly (like we might have a script to fetch anchor from Ethereum given txid and compare to local).
- Summarize: ensure that daily anchors are treated like a heartbeat – any irregularity triggers investigation.

We will maintain these runbooks in our docs (`/docs/research/DR-0032-change-set-ledger/runbooks.md` maybe). They ensure operational readiness.

6. CI/CD Gates and Quality Measures

To maintain the ledger's integrity and reliability, we integrate several checks into our CI pipeline and development process:

- **Test Coverage ≥ 85%:** We configure our test runner to enforce that at least 85% of lines/branches of the ledger package are covered by tests. This gate prevents merging code that significantly lacks tests. The current test suite (unit tests for hashing, idempotency, etc., and integration tests for the overall flow) will easily meet this as we plan thorough tests. If coverage drops (say a new feature added without tests), CI fails and developers must add tests before merging. This maintains high confidence in the ledger logic.

- **Automated Verification on Pull Requests:** For any proposed code change to the ledger (or possibly any DB schema change affecting it), we incorporate an automated sanity test. For example, as part of CI, we spin up a test Postgres, run a mini-sequence:

- Insert a few changes via the API,
- Approve/apply them,
- Run `ledger-verify` on the result and expect OK.
- Maybe also run `ledger-replay` on them and ensure resulting state matches original. This acts as a regression test to catch if any code change inadvertently breaks the chain or causes inconsistent results. It's like a mini end-to-end test focusing on ledger functionality for each PR.

Additionally, for UI or domain changes that might not touch ledger code, we could still have a small representative ledger verification test just to ensure nothing weird (though those likely won't affect ledger, it's more for ledger changes).

- **Schema Validation for Evidence Packs:** In CI, whenever changes are made to evidence structure or formats, we run a validation:
- We might maintain a JSON Schema for the evidence manifest format. We can generate some sample manifests from tests and validate against the schema using a library. If the code's output doesn't conform, that's a bug.
- Also, if a developer changes the Evidence model or usage (like renaming a field), tests should catch it. We can ensure any such change still supports reading old evidence (backward compatibility). For example, if we initially stored `contentHash` and someone thinks to rename it, a test should fail

(since schema or integration test expecting old name fails). This prevents accidental breaking of the evidence contract.
- Also ensure all evidence items have required fields (type, uri, hash) not null unless intended.

- We can add a CI check that any Prisma migration affecting these tables must be accompanied by a verification step. Possibly an automated migration test: run migration, then run `ledger-verify` on an existing sample ledger to confirm no data lost or changed. (Though migration ideally should not alter existing ledger data at all except adding new tables/columns).

- **Migration Integrity Hook:** When database schema changes (especially for `change_set` or related tables), we institute a hook to verify that the ledger integrity is preserved across the migration.

- For example, if a migration adds a new column or changes a data type, we should ensure it doesn't modify existing data in a way that would break hashes. Our approach is to avoid destructive updates to ledger data. But if a migration were to do something like trim whitespace in a column that is part of hashed data, that would break things. So we need a process:
    - Code reviewer should flag any migration touching ledger data fields as dangerous.
    - We can enforce via tests: simulate the migration on a test DB with some ledger entries, then run verification after migration to ensure all hashes still match [33].
    - Alternatively, disallow by policy any migration that updates `change_set` or `change_set_applied` data for existing rows. If we need to change something, it might have to be a compensating event in ledger, not a direct alter.
- We can write a small test that runs all migrations up and down on a sample and checks ledger integrity at each step.

- In CI, possibly run `prisma migrate deploy` on a test container and then run `ledger-verify` to ensure it's still good.

- **Continuous Integration of Verification Tools:** We plan to include the `ledger-verify` command in some regular pipeline – e.g., a nightly job on a staging environment could automatically run verification and perhaps anchor a hash to an internal log. This isn't strictly a CI gate, but a maintenance job to catch issues early (especially hardware issues causing silent corruption, which is rare but possible; Postgres has checksums option to catch that at IO level too).

- **Code Quality and Simplicity:** Not a gate per se, but we will do code reviews focusing on clarity (since ledger code is security-critical, it should be easy to reason about). We avoid overly complex logic that could hide bugs. Our stable TypeScript interfaces ensure that other parts of the system (like the multi-agent orchestrator or the approval service) call these functions without needing to change their behavior. For instance, perhaps previously, after an approval, the code directly updated some record; now it will call `applyChange()`. We ensure that `applyChange()` interface is straightforward (taking a change ID or object and performing apply). We maintain backward compatibility where possible to avoid breaking existing flows.

- **Performance Budget:** While not a formal gate, we monitor that key operations (append, apply, verify) remain within acceptable time:

- Append and apply should be sub-second operations under normal load, which they will be since mostly one DB transaction and some hashing.
- Verify of a day's worth of events (maybe thousands) should be seconds, which is fine.
- We could have a performance test in CI if needed, but likely not necessary. Instead, we rely on the benchmark done offline.

By establishing these CI gates and practices, we aim to ensure the ledger system remains **correct, secure, and maintainable**. Each code change is vetted to not compromise the tamper-evident properties or idempotency guarantees. The ledger is a critical piece of infrastructure for trust, so we treat it with the rigor of security-sensitive code: heavy testing, careful review, and continuous verification.

Conclusion: The proposed Change-Set Ledger system provides a robust upgrade to our draft→approve→apply workflow. It brings blockchain-inspired integrity (hash chains and external anchors) [1] without the complexity of a public blockchain, ensuring that any internal collusion or admin tampering would be evident [33] [2]. It enforces idempotent, exactly-once change application [4], preventing duplicate writes and race conditions even in a concurrent multi-agent environment. It also greatly enhances forensic readiness: every change is accompanied by a rich evidence trail and is linked into a lineage graph, so we can reconstruct not just the state but the story behind each change. The integration with our existing systems (Prisma ORM, multi-agent orchestration, approval UI, observability stack) has been carefully designed to minimize friction – agents and users will follow almost the same steps as before, with the ledger working transparently in the background to provide safety and auditability. With comprehensive tests, documentation, and runbooks in place, we are set to implement this ledger with high confidence in its correctness and to maintain it over the long term. This system will significantly strengthen our compliance posture (we have immutable audit logs for all AI agent actions) [7], and protect our data integrity against both accidental errors and malicious attempts.

---

[1] [2] [3] [16] [17] [18] [19] [20] [21] [30] [33] Integrity Guarantees of Blockchains In Case of Single Owner Or Colluding Owners - Bozho's tech blog
https://techblog.bozho.net/integrity-guarantees-of-blockchains-in-case-of-single-owner-or-colluding-owners/

[4] [13] [14] An In-Depth Introduction To Idempotency | Luca Palmieri
https://lpalmieri.com/posts/idempotency/

[5] [6] Event Sourcing pattern - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing

[7] [8] [9] [12] What Payments & Fintech Architecture Keeps Forgetting | by Yuri Geronimus | Aug, 2025 | Medium
https://yurigeronimus.medium.com/what-payments-fintech-architecture-keeps-forgetting-f6d4d7034da6

[10] [11] [15] Modern Treasury Journal - Designing the Ledgers API with Concurrency Control
https://www.moderntreasury.com/journal/designing-ledgers-with-optimistic-locking

[22] [PDF] Digital Evidence Preservation - NIST Technical Series Publications
https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8387.pdf

[23] [24] [25] [26] [27] [28] [29] [31] [32] I don't feel that auditability is the most interesting part of Event Sourcing. : r/softwarearchitecture
https://www.reddit.com/r/softwarearchitecture/comments/1kpmsf8/i_dont_feel_that_auditability_is_the_most/