Reliable Context

RELIABLE CONTEXT / RESEARCH & PRACTICAL GUIDANCE

# Event Sourcing vs Audit Tables vs Change Data Capture

> Reliable Context is a kernel of truth: a system of record for organizations that fuse private and public information and must defend every value they act on. One canonical record per real thing, a cited source and an observation date behind every value, unknowns declared rather than left blank, and any past state of the record reproducible.

Published by Reliable Context · 2026-10-29 · event sourcing · event sourcing vs audit log · change data capture vs event sourcing · event sourcing audit trail · event sourcing explained

Original article: https://reliablecontext.com/articles/event-sourcing-vs-audit-table-vs-change-data-capture



### In this article

1. The question history must answer
2. One change, three records
3. Event sourcing: events as the record
4. Audit tables and triggers: what they miss
5. Change data capture: log-based, row-level, no intent
6. Retention, compaction and deletion
7. Side by side
8. Decision table
9. Checklist

## The question history must answer

There are three common ways to keep the history of a database: make the history the record (event sourcing), copy each changed row into a side table from a trigger (an audit table), or read the database's transaction log and publish each row change (change data capture, CDC). One question separates them: **can you reproduce any past state and show who changed it and why?**

- **Event sourcing** can answer all three parts, but only if each event is designed to carry the actor, the reason and the effective time. The pattern does not supply those fields.
- **An audit table** reproduces past rows if it stored full row images from the first day. It records a database user, often a shared application role, and never a reason.
- **CDC** reproduces the ordered sequence of committed row changes with transaction identifiers. It records neither a person nor a reason, and a compacted topic is not history at all.

The general method of rebuilding a record as it stood on a given date is described in reproducing a past state.

## One change, three records

The example is fictional. **Halden Therapeutics AG is a fictional company**, and the people, addresses, identifiers and systems below are invented for illustration.

Halden moved its registered office within Basel, effective 30 September 2026; the commercial register published the change on 2 October. On 5 October 2026 at 09:12 UTC, Lena Brunner, a corporate records officer, entered the new address in the entity master system, citing the register publication. The system writes to a PostgreSQL table `company` through a connection pool that logs in as the role `halden_app`.

### 1. As a domain event

In an event-sourced system the change is a named event appended to the company's stream. The application, not the database, decides what goes in it.

```
{
  "eventType": "RegisteredAddressChanged",
  "schemaVersion": 2,
  "streamId": "company/HAL-0001",
  "streamVersion": 17,
  "effectiveFrom": "2026-09-30",
  "recordedAt": "2026-10-05T09:12:44Z",
  "data": {
    "previous": { "street": "Rheinweg 12", "postcode": "4058", "city": "Basel" },
    "new":      { "street": "Elisabethenanlage 7", "postcode": "4051", "city": "Basel" }
  },
  "reason": {
    "code": "REGISTERED_OFFICE_RELOCATION",
    "text": "Board resolution of 14 Sept 2026; register publication of 2 Oct 2026",
    "evidenceRef": "doc/register-extract-2026-10-02.pdf"
  },
  "actor": { "userId": "u-4471", "name": "Lena Brunner", "role": "corporate-records-officer" }
}
```

The fields that answer "who" and "why" (`actor`, `reason`, `effectiveFrom`) exist because someone designed them in. A generic `CompanyUpdated` event carrying only the new address is also event sourcing, and it answers neither question.

### 2. As an audit-table row from a trigger

The PostgreSQL manual's audit example records a row in a side table for each insert, update or delete, stamping "the current time and user name" and the operation; on update it inserts `'U', now(), current_user, NEW.*` (PostgreSQL 18, Example 41.4). Applied to `company`:

| operation | stamp | userid | company_id | legal_name | street | postcode | city |
|---|---|---|---|---|---|---|---|
| U | 2026-10-05 09:12:44.208 | halden_app | HAL-0001 | Halden Therapeutics AG | Elisabethenanlage 7 | 4051 | Basel |

Three things are missing. The old address is not in this row, because the example stores only the new image; the previous value is whatever the preceding audit row says, if one exists. `userid` is `current_user`, the database role, which here is the pool's shared login, not Lena Brunner. And there is no reason, because the database never received one.

### 3. As a Debezium CDC change event

Debezium's PostgreSQL connector reads the write-ahead log through logical decoding and emits change events whose value has an envelope of `before`, `after`, `source`, `op` and timestamps (Debezium PostgreSQL connector, change event values). With the table set to `REPLICA IDENTITY FULL`:

```
{
  "before": { "company_id": "HAL-0001", "legal_name": "Halden Therapeutics AG",
```

```
              "street": "Rheinweg 12", "postcode": "4058", "city": "Basel" },
  "after":  { "company_id": "HAL-0001", "legal_name": "Halden Therapeutics AG",
              "street": "Elisabethenanlage 7", "postcode": "4051", "city": "Basel" },
  "source": { "version": "3.7.0.Final", "connector": "postgresql", "name": "halden",
              "ts_ms": 1791191564208, "snapshot": false, "db": "entity_master",
              "schema": "public", "table": "company", "txId": 918377,
              "lsn": 5381274632, "xmin": null },
  "op": "u",
  "ts_ms": 1791191564391
}
```

Per the update events documentation, `op` is one of `c`, `u`, `d`, `r` (read, snapshots only), `t` (truncate) or `m` (message). Inside `source`, `ts_ms` is "the time that the change was made in the database"; the top-level `ts_ms` is when the connector processed the event. Under the default replica identity, `before` would hold only the primary key; full previous values require `REPLICA IDENTITY FULL` (replica identity).

The event is a precise, ordered, transaction-stamped account of the row before and after. It contains no person and no reason. Neither does the log it was read from.

## Event sourcing: events as the record

Martin Fowler's definition is one line: "Capture all changes to an application state as a sequence of events" (Fowler, Event Sourcing). Two of the facilities he lists are the reproducibility test itself. **Complete rebuild**: "discard the application state completely and rebuild it by re-running the events from the event log on an empty application". **Temporal query**: determine the state at any point in time "by starting with a blank state and rerunning the events up to a particular time or event". He also notes a weaker variant in which current state lives in a database and "the event logs may only be there for audit"; replay proves something only when the log is the official record.

Microsoft's Azure Architecture Center describes events recorded "in an append-only store", which "acts as the system of record", with current state derived by replaying an entity's stream (Microsoft, Event Sourcing pattern). Corrections are new events, never edits: "The only way to update an entity or undo a change is to add a compensating event". The glossary entry on event sourcing gives the short definition.

### Intent and time are design choices

The Microsoft guidance says the "why" has to be designed in: "Design events to capture the business intent behind each change in addition to the resulting state." It contrasts *two seats were reserved* with *remaining seats changed to 42*, and warns that "State-focused events reduce the event store to a change log that has no business meaning" (Problems and considerations). The first reason it gives for using the pattern is "You want to capture intent, purpose, or reason in the data" (When to use this pattern).

Time needs the same care. Fowler: "if events need to support bitemporal history, they need to be bitemporal themselves. This means each event needs an actual date (or time) for when the event occurred in the world, and a record date (or time) for when we learned about it" (Fowler, Bitemporal History). In the example, `effectiveFrom` and `recordedAt` are those two dates. Other timestamps a record may need are covered in valid, transaction, observation and publication time.

### Replay, snapshots and external inputs

Long streams are expensive to replay, so implementations take snapshots. Microsoft: "Snapshots are an optimization, not a replacement for the eventstream. The eventstream remains the source of truth." Read models built under CQRS (Command Query Responsibility Segregation, separating the write model from read models) are likewise projections, not records (Fowler, CQRS).

Replay has a trap the other two patterns avoid, because they never re-run logic. If handling an event depended on an external answer such as an exchange rate, "the data that they return has an effect on the results on handling an event"; Fowler's remedy is a gateway that remembers responses so replay reuses them (Fowler, Event Sourcing). A store that does not capture its external inputs reproduces the events, not necessarily the state.

### Schema evolution and cost

Stored events are permanent, so their schemas accumulate. Microsoft lists tolerant deserialization, a version identifier per event, upcasting old events on read, and in-place migration, which "breaks immutability and should be a last resort because it undermines the audit trail". The same page opens with a warning: "For most systems and most parts of a system, traditional data management is sufficient." Fowler says of CQRS that "for most systems CQRS adds risky complexity".

## Audit tables and triggers: what they miss

An audit trigger runs inside the transaction that changes the row, so it cannot miss an insert, update or delete it covers. Its gaps are structural:

- **Who.** `current_user` is the database role. Behind a connection pool, every person's change carries the same role. A common workaround is an application-set session variable that the trigger reads, which the database does not verify.
- **Why.** The database never sees a reason unless the application writes one into the base table.
- **Coverage.** PostgreSQL triggers "may be defined to fire for `TRUNCATE`, though only `FOR EACH STATEMENT`" (PostgreSQL 18, CREATE TRIGGER), so the row-level example trigger records nothing on truncation. Changes made with triggers disabled, or by a role that can write the audit table, are outside its guarantee.
- **Reproducibility.** A past state is the latest audit row per key stamped at or before the target time. That works only if full images were stored from the first insert; a trigger added in year three cannot tell you year one.
- **Schema evolution.** `INSERT ... SELECT ..., NEW.*` depends on matching columns in matching order, so adding a base-table column breaks the trigger until the audit table is altered too.

The warehouse equivalent is Kimball's Slowly Changing Dimension Type 2: changes "add a new row in the dimension with the updated attribute values", with a row effective date, a row expiration date and a current row indicator (Kimball Group, Type 2). Like an audit table, it versions state, not the actor or the reason.

## Change data capture: log-based, row-level, no intent

Log-based CDC reads what the database has already committed. That adds no write to the user's transaction, orders changes by log position (`lsn`) and groups them by transaction: with transaction metadata enabled, Debezium emits `BEGIN` and `END` events whose `id` combines transaction ID and LSN (transaction metadata). The limits follow from the source:

- **No who, no why.** The envelope has no user or reason field, because the logged row change carries neither.
- **History starts at deployment.** Transaction metadata is available "only for transactions that occur after you deploy the connector". Earlier rows arrive as snapshot reads (`op: r`), which record state at snapshot time, not how it got there.
- **Before-images depend on configuration**, as shown above.
- **Key changes split.** An update to a primary key is emitted as a delete for the old key, a tombstone and a create for the new key (primary key updates).

There is one way to put intent into the same log. Debezium captures "a generic logical decoding message that is written directly to the WAL by using the `pg_logical_emit_message` function" as an `op: m` event, through the pgoutput plugin on PostgreSQL 14 and later (message events). An application can emit a transactional message carrying actor and reason alongside the update, and consumers can join the two on the transaction ID. At that point the application is designing events, and the remaining question is why those events are not the record.

Microsoft also warns against treating the transport as the store: message brokers such as Apache Kafka "aren't a substitute for an event store" (Problems and considerations).

## Retention, compaction and deletion

Whether a CDC topic or an event stream is history depends on retention. Kafka log compaction "ensures that Kafka will always retain at least the last known value for each message key within the log of data for a single topic partition", by removing "records where we have a more recent update with the same primary key" (Apache Kafka 4.4, Log Compaction). A record with a key and a null payload is a tombstone; tombstones are themselves removed after `delete.retention.ms` ("the default is 24 hours"), so a consumer lagging longer can miss deletes (What guarantees does log compaction provide?). Debezium's PostgreSQL events "are designed to work with Kafka log compaction", and each delete is followed by a tombstone for that purpose (tombstone events).

A compacted topic is therefore a current-state snapshot keyed by row. After compaction, the Rheinweg address is gone. That is intended, and it means compacted CDC, or an event stream kept in a compacted topic, cannot answer a temporal query. A reproducible history needs unbounded retention or archiving before compaction runs; the append-only log entry describes the property compaction gives up.

Deleting on purpose is the opposite problem. Microsoft notes that an immutable event store "conflicts with data protection regulations that require deletion of personal data", and describes two approaches: keep personal data outside the store and reference it by identifier, or encrypt it per subject and delete the key (crypto-

shredding). Audit tables and CDC archives hold the same personal data in more copies. What a particular law requires is outside this article, which is not legal advice.

## Side by side

| Criterion | Event sourcing | Audit table (trigger) | Log-based CDC |
|---|---|---|---|
| Reproduce any past state | Yes, by replay, if the stream is the record and external inputs were captured | Yes, if full row images were stored from the first insert | Only from connector deployment, with full before-images, and never from a compacted topic |
| Intent (why) | Yes, if events carry a reason | No, unless written into the base table | No; only via logical messages emitted alongside |
| Who | Yes, if events carry the actor | Database role; a person only via session context | No user field |
| Effective vs recorded time | Yes, if events carry both dates | Recorded time only | Commit time and capture time |
| Schema evolution | Old events kept forever; versioning and upcasting | Audit table altered in step | Events follow table schema; consumers handle each version |
| Deletion and erasure | Hard by design; external storage or crypto-shredding | Deleting rows breaks reproducibility for that subject | Tombstones remove keys; archives keep copies |
| Cost | High: new storage model, projections, eventual consistency | Low; one extra write per change | Moderate: connector, broker, replication slot |

## Decision table

| If your situation is... | Prefer | Because |
|---|---|---|
| Changes are business decisions that must be justified (registry data, approvals, ledgers) | Event sourcing with actor, reason and two dates in every event | Only pattern where why is part of the record |
| An existing CRUD system needs change history soon | Audit trigger storing old and new images, plus application-set actor and reason | Cheapest route to who and what; why depends on application discipline |
| Feeding search indexes, caches or a warehouse | CDC | Built for propagation and ordering, not evidence |
| A warehouse must report "as was" | SCD Type 2 loaded from CDC or audit rows | Versions state for analysis |
| History must survive scrutiny years later | Any of the above, with unbounded retention and actor and reason captured at the source | Retention and intent decide defensibility, not the capture mechanism |

## Checklist

1. Each change records a person or named system, not a pooled database role.
2. Each change records a reason, with a pointer to the evidence.
3. Each change records when it took effect and when it was recorded.
4. Full before and after values exist for every change since the first.
5. History retention is unbounded, or history is archived before compaction.
6. Reconstruction is tested: state as of a past date matches what was reported then.
7. Personal data has a deletion path that does not rewrite history.

The test used here is one of Reliable Context's published guarantees: any past state reproducible.

## Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.


1. https://www.postgresql.org/docs/current/plpgsql-trigger.html
2. https://debezium.io/documentation/reference/stable/connectors/postgresql.html
3. https://martinfowler.com/eaaDev/EventSourcing.html
4. https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing
5. https://martinfowler.com/articles/bitemporal-history.html
6. https://martinfowler.com/bliki/CQRS.html
7. https://www.postgresql.org/docs/current/sql-createtrigger.html
8. https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/kimball-techniques/dimensional-modeling-techniques/type-2
9. https://kafka.apache.org/44/design/design

THE PUBLISHER

# About Reliable Context

Reliable Context is a system of record for organizations that fuse private and public information and must defend every value they act on.

### The guarantees

One canonical record per real thing: duplicates resolve by rule, ambiguity is shown rather than hidden, and related organizations relate without ever being merged. A verbatim, cited source and an observation date sit behind every value. Unknowns are declared as not found, blocked or not applicable with a reason, never left as an empty cell. Any past state of the record is reproducible, together with what changed since.

### Where it runs

It runs inside the customer's perimeter, one database per organization. Permitted sources, model routing, reader clearance and retention are set as the customer's own policies, and every value is exportable with its chain of custody for an auditor or regulator.

### What it is not

It is not a vector database or a RAG framework, not a data catalog, MDM or data-quality suite, not a CRM or data-enrichment vendor, and not a chatbot or AI assistant. Where a reader needs one of those, the honest answer is to say so and point them there.

### Visit

See Reliable Context.

Reliable Context is in production use in two sectors; more than six thousand of its identity decisions were made by rule and none has been reversed. Articles published here explain general practice, cite their primary and academic sources, and name the jurisdiction of any regulatory rule they describe. They are not legal, regulatory or compliance advice for a specific organization, and requirements change.

**Website:** https://reliablecontext.com

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.