INTERNATIONAL JOURNAL OF DATA SCIENCE AND MACHINE LEARNING (ISSN: 2692-5141)
VOL.06 ISSUE 02 (2026)

*Research Article*

# Event-Sourced Governance Of Data Quality Rules For Real-Time, Auditable Healthcare Data Platforms

**Ritulkumar Patel**
Independent Researcher, Boston, MA, USA

check for updates
Received: 20 October 2026
Revised: 30 October 2026
Accepted: 06 November 2026
Published: 14 November2026

Doi: 10.55640/ijdsml-06-02-15
Page No: 241-260

**Copyright:** © 2026 Authors retain the copyright of their manuscripts, and all Open Access articles are disseminated under the terms of the Creative Commons Attribution License 4.0 (CC-BY), which licenses unrestricted use, distribution, and reproduction in any medium, provided that the original work is appropriately cited.

**Abstract**

Rule-based checks are central to data quality (DQ) assessment on healthcare data platforms, and their parameters change often. In regulated settings each change must be approved, promoted through staged environments, and later explained to auditors, yet DQ tools rarely govern the rule set itself. This study asks whether event sourcing can make such changes both fast and verifiable. We propose an architecture in which every rule-lifecycle action is an immutable, hash-chained event in a log partitioned by rule; separation of duties and risk-tiered approval are enforced before an event is written; idempotent projections drive gated promotion; rule parameters are bound at runtime; and the log head is anchored outside the platform. We evaluated the design in a single-process simulation on synthetic patient data against three baselines: a dbt rebuild pipeline, a mutable parameter table with an audit trigger, and a ledger-style table with a hash-chained audit. The simulation reproduces the relevant Kafka log semantics but is not a production deployment. Approved changes reached production in a median of 0.092 s, against tens of seconds to minutes for the rebuild pipeline; the two table baselines were faster still, so the speed comes from runtime parameter binding rather than from event sourcing. With every record anchored, the hash-chained designs detected all seven tested attack types, whereas the unchained audit table missed every edit of attribution or value. Historical states were reconstructed exactly, an independent verifier found no constraint violations in normal runs and flagged every injected violation, and ablations identified the components that correctness depends on. The main limitations are synthetic data, a simulated log, and modeled rather than measured approval delays

**Keywords:** Data quality; Data governance; Event sourcing; Audit trail; Tamper-evident logging; Electronic health records; Regulatory compliance; Change management

## 1. INTRODUCTION

### 1.1 Background and motivation

Secondary use of electronic health record (EHR) data depends on systematic data quality (DQ) assessment [1,2], and much of that assessment is implemented as rules. Wang *et al.* compressed 63,397 rules into 28 parameterized templates whose parameters live in knowledge tables [3]; Mohamed *et al.* ran 8,199 rules derived from these templates across 13 sites of a clinical research network hosted on Snowflake [4]; and the Observational Health Data Sciences and Informatics (OHDSI) Data Quality Dashboard executes more than 3,300 checks, each paired with a configurable threshold [5]. In each case a rule is a piece of logic plus a few parameters: a pass threshold, a plausible range, an expected unit, or a required diagnosis code.

Those parameters change, and the changes matter. Wang *et al.* report that rule knowledge evolves, that users ask for local exceptions, and that such configurations need governance over their lifecycle [3]; Mohamed *et al.* list rule refinement and automated, repeated execution as open needs [4]. A common case is an incoming data file that fails a completeness rule at 75%

against a required 80%. Processing the file requires relaxing the threshold, and in a regulated setting the relaxation must first be confirmed with the data owner, approved by someone other than its author, promoted through test environments, and recorded so that an auditor can later establish who changed what, why, when, and where it was in force. The HIPAA Security Rule requires mechanisms that record and examine activity in systems containing electronic protected health information [6] and requires specified documentation to be retained for six years from its creation or the date it was last in effect, whichever is later [7]; the GDPR makes controllers responsible for demonstrating compliance (Art. 5(2)) [8].

### 1.2 Related work

*Rule-based DQ assessment in healthcare.* Kahn *et al.* harmonized DQ terminology into conformance, completeness, and plausibility [1], and Weiskopf and Weng reviewed assessment methods and dimensions [2]. Wang *et al.* built a rule system from logic templates and knowledge tables, applied it to a warehouse of 1.46 million patients, and separated rules for physically impossible values from rules for possible but implausible values, which need validation before use [3]. Mohamed *et al.* adapted those templates to the PCORnet common data model and found laboratory-unit and medication problems that structural curation checks missed [4]. AIDAVA validates data continuously across integration stages and shows that completeness should be checked before consistency [9]. These systems define and execute rules; none models who may change a rule, how a change is approved and promoted, or how the rule set in force at a past time is reconstructed.

*Validation as code, rule deployment, and model registries.* Deequ expresses constraints as unit tests for data and computes metrics incrementally [10]; TensorFlow Extended (TFX) data validation checks data against a schema that its owners curate [11]. Both treat validation logic as an artifact that evolves with the data, but neither addresses how changes to it are approved, promoted, or audited. Kumar *et al.* stream air-quality data through Kafka into a complex event processing engine whose rules are hard-coded [12]. Polimeno *et al.* enforce access-control policies on service pipelines while maximizing data quality [13]; their policies govern data flows rather than changes to quality rules, a role that general policy engines such as Open Policy Agent could also play at the command layer [14]. Machine learning platforms solve a related problem for models: MLflow tracks model artifacts [15], and model registries move versions through stages such as Staging and Production with approvable transition requests and an activity history [16]. Our lifecycle borrows this staged promotion and adds parameter-level events, risk tiers, and tamper evidence. Comuzzi *et al.* formalize imperfection patterns of event logs [17], a reminder that a governance log must itself be well formed, and language models that generate data-refinement programs [18] point to a growing volume of machine-proposed rules that will need the same approvals as human ones.

*Event sourcing, ledgers, and tamper evidence.* Event sourcing stores every state change as an immutable event and derives the current state through replay; command-query responsibility segregation separates the write path from read-optimized projections [19]. Overeem *et al.* studied 19 event-sourced systems and reported schema evolution, projection rebuilding, and data privacy as recurring difficulties [20]. Kafka offers partitioned, offset-addressed logs with at-least-once delivery [21]. Schneier and Kelsey introduced hash-chained audit logs [22]; Crosby and Wallach showed that detecting insider tampering requires auditors to hold commitments to the log [23], an idea rooted in publishing digests to an independent party [24] and used at Internet scale by Certificate Transparency [25]. Ledger databases bring these ideas inside the database: SQL Ledger chains the history of updatable tables in SHA-256 blocks and stores database digests in immutable storage [26], and LedgerDB provides verifiable audit for a centralized ledger [27]. SQL:2011 system-versioned tables [28] and lakehouse table formats [29,30] keep history without such guarantees. Politou *et al.* analyze the tension between immutable records and erasure rights [31], and Fitzgerald *et al.* describe continuous compliance in regulated software development [32].

### 1.3 Where existing approaches fall short

We state the requirements that follow from the scenario in Section 1.1 and check, for each family of approaches reviewed above, which requirements it supports and what evidence shows it. A governed rule platform should (R1) activate an approved parameter change within seconds without rebuilding code; (R2) enforce approval, separation of duties, and a recorded justification for risky changes; (R3) promote changes through ordered, gated environments; (R4) reconstruct the rule set in force at any past time; (R5) make tampering with the change history detectable, including tampering by insiders with storage access; and (R6) let an auditor verify the recorded decisions without trusting the system that made them. Table 1 summarizes the assessment. Our own measurements refer to the event-sourced prototype (ES) and three baselines described in Section 2.4: B1, a dbt rebuild pipeline; B2, a mutable parameter table with an audit trigger; and B3, a ledger-style table with a hash-chained audit and external anchors.

Rebuild pipelines treat rules as code and deploy them with tools such as dbt (data build tool) [33]. Code review provides some control (R2), but approvals and justifications live outside the deployed artifact, and every parameter change pays for a build; our measurements put approval-to-production latency at 25.5 s for on-demand builds and 241–1,000 s for scheduled builds

(Section 3.1). DQ frameworks separate parameters from logic (knowledge tables [3], check thresholds [5], declarative constraints [10], curated schemas [11]) but do not govern changes to those parameters; Wang *et al.* explicitly call for such governance [3]. Mutable configuration tables with audit triggers, or SQL:2011 system-versioned tables [28], give fast changes and a history (R1, R4), but the history is only as trustworthy as the database administrators: in our experiment an unchained audit table detected none of the edits to who made a change or to a recorded value (Section 3.3). Ledger tables chain row history with cryptographic hashes and publish digests outside the database [26,27], which provides R4–R5, but approval workflows, separation of duties, and environment promotion remain the application's responsibility. Table formats with snapshot time travel provide R4 for data [29,30], and machine learning model registries provide stage transitions with approval requests and an activity log for models (R2–R3) [15,16]; neither targets tamper evidence. Rule engines fed by streams activate rules quickly but hard-code them without versioning or approval [12]. No single approach covers R1–R6 for DQ rule parameters.

**Table 1. Requirements met by existing approaches and the evidence for each assessment.**

| Approach (examples) | R1 | R2 | R3 | R4 | R5 | R6 | Evidence |
|---|---|---|---|---|---|---|---|
| Rebuild pipelines (dbt) [33] | – | ○ | ● | ○ | – | – | Measured B1 latency 25.5–1,000 s (Section 3.1) |
| DQ frameworks [3,5,10,11] | ● | – | – | – | – | – | Lifecycle governance named as a need [3] |
| Mutable table + audit trigger; temporal tables [28] | ● | ○ | ○ | ● | – | – | Measured B2: attribution and value edits undetected (Section 3.3) |
| Ledger tables [26,27] | ● | ○ | ○ | ● | ● | ○ | Hash chain + external digests; measured B3 (Section 3.3) |
| Time travel tables [29,30] | ○ | – | – | ● | – | – | Snapshot history of data, not of rule decisions |
| Model registries [15,16] | – | ● | ● | ○ | – | – | Stage transitions with approval requests and activity log |
| Streaming rule engines [12] | ● | – | – | – | – | – | Rules hard-coded, no versioning or approval |
| This work | ● | ● | ● | ● | ● | ● | Sections 2–3 |

R1: activation within seconds without rebuild; R2: enforced approval, separation of duties, justification; R3: ordered, gated promotion; R4: point-in-time reconstruction; R5: tamper evidence against insiders; R6: independent verification of recorded decisions. ● supported; ○ partly, or left to the application; – not supported.

### 1.4 Contributions and organization

This paper treats the rule set as governed state and asks whether event sourcing [19,20] can give rule changes both low latency

and verifiable accountability. Its contributions are:

(1) A formulation of DQ rule governance as ten constraints (C1–C10), with an objective over approval-to-production latency.

(2) An event-sourced architecture that keeps rule templates as code and records parameters as events, enforces policy before any event is written, chains events with SHA-256 and external anchors, and drives gated promotion through idempotent projections, with an explicit classification of which components are needed for correctness and which only improve performance or convenience.

(3) A simulation study on Synthea data [34] against three baselines: a dbt rebuild pipeline, a mutable parameter table with an audit trigger, and a ledger-style table with a hash-chained audit and anchors. It covers latency, capacity, integrity, policy enforcement, gating, component ablations, verification cost, and a lead-time model that includes human approval.

(4) Evidence that separates two effects: sub-second propagation comes from binding parameters at runtime, whereas tamper evidence comes from a hash chain anchored outside the platform. Event sourcing adds per-rule ordering, replayable projections across environments, and decision-level verification, and we identify when a ledger table is the simpler choice.

Section 2 formulates the problem, presents the architecture, and describes the experiments. Section 3 reports the results, Section 4 discusses them, and Section 5 concludes.

## 2. Materials and methods

### 2.1 Problem formulation

A *template* is a parameterized SQL predicate maintained as code. A rule $r$ has a fixed template $\tau(r)$ and a sequence of versions $v = 1, 2, \ldots$ with parameter vectors $\theta_{r,v}$; retiring a rule is a version with $\theta_{r,v} = \perp$. Environments form an ordered list $E = (e_1, \ldots, e_4)$ = (dev, uat, prod_test, prod): development, user acceptance testing (uat), pre-production testing (prod_test), and production (prod). Each actor holds one role $\rho \in$ {author, approver, system}. A command $c = (a, op, r, v, \hat{v}, \theta, j)$ carries an actor, an operation $op \in$ {create, update, retire, approve, promote, rollback}, a target version $v$, the version $\hat{v}$ the actor last saw, new parameters $\theta$, and an optional justification reference $j$. For approve and promote, $v$ names the version being approved or promoted, which must be the latest version of the rule; for create, update, and retire, the handler checks $\hat{v}$ and assigns $v = \text{latest}(r) + 1$; for rollback, it checks $\hat{v}$ and sets $v$ to the most recent earlier production version. We write promote($r$, $v$, $e_k$) for the system command that activates version $v$ of rule $r$ in environment $e_k$. Accepted commands become a history $H$ of events, and each environment $e$ has a state $S_e(t)$ that maps each rule to its active version or $\perp$; rule evaluation binds the parameters in $S_e(t)$ to the templates at runtime. The constraints are:

**C1** *Append-only history.* Events are only appended; none is modified or removed, and each is recorded once.

**C2** *Tamper evidence.* Modification, deletion, reordering, or insertion of a recorded event, and truncation of the history, are detectable by a verifier that holds externally published anchors, for every record at or before the newest published anchor.

**C3** *Separation of duties.* Each operation is permitted only to its role (authors create, update, retire, and roll back; approvers approve; only the system promotes), and no version is approved by its author.

**C4** *Risk-tiered approval.* A high-risk version, per (1), is approved only with a justification reference that points to the recorded confirmation of the affected data owner.

**C5** *Ordered, gated promotion.* A version becomes active in $e_{k+1}$ only after it was approved, became active in $e_k$, and passed the gate there; a rollback restores only a version previously active in production.

**C6** *No lost updates.* A change based on a stale version is rejected, versions of a rule are contiguous, and a superseded version is never approved or promoted.

**C7** *Idempotent application.* Under at-least-once delivery, each event changes the projections exactly once.

**C8** *Data minimization.* Governance events carry rule metadata only, never patient-level values.

**C9** *Retention.* The history is retained for a configured period. In the evaluated setting we assume six years, mirroring the HIPAA retention period for required documentation [7]; this is a design assumption for governance history, not a claim

that every healthcare event log must be kept for six years.

**C10** *Reconstructability.* The state at any time $t$ equals the replay of the history up to $t$, and the live projections equal the replay of the whole history.

A change is high risk if it makes a check less strict. With $\pi$ the pass threshold, $[lo, hi]$ the accepted value range, and $\alpha$ the minimum age of an age-dependent rule, and $\theta$ and $\theta'$ the active and proposed parameters,

$$\text{risk}(\theta, \theta') = \text{high if} \quad \theta' = \bot \vee \pi' < \pi \vee lo' < lo \vee hi' > hi \vee \alpha' < \alpha, \tag{1}$$

and is classified as low risk otherwise. The propagation latency of version $v$ of rule $r$ is

$$\ell(r, v) = t_{\text{prod}}(r, v) - t_{\text{appr}}(r, v), \tag{2}$$

where $t_{\text{appr}}$ is the approval time and $t_{\text{prod}}$ the time the version becomes active in production. The objective is to minimize the median and tail of $\ell$ subject to C1–C10 at the offered rate of approved changes. Human approval time is excluded from $\ell$ because no design controls it; Section 3.6 models its effect on end-to-end lead time. We assume templates are tested in continuous integration (CI), actors are authenticated, and delivery is at least once. The adversary is an insider with write access to governance storage who may modify, delete, reorder, insert, or truncate records and recompute hashes, but cannot alter anchors already published to a separate trust domain.

### 2.2 Proposed architecture

Figure 1 shows the design. Commands enter a command handler that validates them and, if accepted, appends one event to a log partitioned by rule identifier. The head of each partition is periodically published to an anchor store in a separate trust domain. A projector consumes each partition, records applied events, and maintains the active parameters of every rule in every environment. After a version becomes active in $e_k$, a gate evaluates it on $e_k$'s data and, if it passes, the projector issues a system promotion command for $e_{k+1}$, which again passes through the handler. Rule evaluation reads parameters from the projections and templates from the repository; a verifier reads the log, the anchors, and the projections.

The central design choice extends the template and knowledge-table split of Wang *et al.* [3]. A template, such as "value of code $c$ outside $[lo, hi]$", is SQL in the repository and is compiled and tested in CI. Its parameters are recorded as events and bound when the rule runs, so a threshold change needs no build, only an approved event; template changes still follow the code path, which suits their lower frequency.

**Definition 1** *(Rule event).* An event is a tuple $\varepsilon$ = (id, $r$, $v$, type, actor, role, env, $j$, payload, $p$, $o$, ts, $h_{\text{prev}}$, $h$). Its type is one of BOOTSTRAPPED, CREATED, UPDATED, RETIRE_REQUESTED, APPROVED, PROMOTED, GATE_FAILED, or ROLLED_BACK. The payload holds parameters and the computed risk tier; $p$ and $o$ are the partition and offset.

**Definition 2** *(Partitioning and hash chain).* Rule $r$ is assigned to partition $p(r)$ = MD5($r$) mod $P$, so all events of a rule are totally ordered, as in a Kafka topic keyed by rule [21]. Within a partition,

$$h_o = \text{SHA-256}(h_{o-1} \| \text{canon}(\varepsilon_o)), \qquad h_{-1} = 0^{256}, \tag{3}$$

where canon(·) is the event serialized as JSON with sorted keys and without its own hash and $0^{256}$ is the 256-bit all-zero value (64 zero hexadecimal digits), so changing any recorded event changes every later hash [22].

**Definition 3** *(Anchor and projection).* Every $k$-th head $(p, o, h_o)$ is written to an anchor store that the log's administrators cannot rewrite, such as write-once object storage or a public time-stamping service [23,24]. A history verifies if every chain recomputes and every anchor matches. The state of environment $e$ at time $t$ is a fold over the history:

$$S_e(t) = \text{fold}(\text{apply}, S_0, \{\varepsilon \in H : \text{ts}(\varepsilon) \le t\}). \tag{4}$$

Projections are disposable caches; the log is the record. Figure 2 shows the lifecycle of a version.

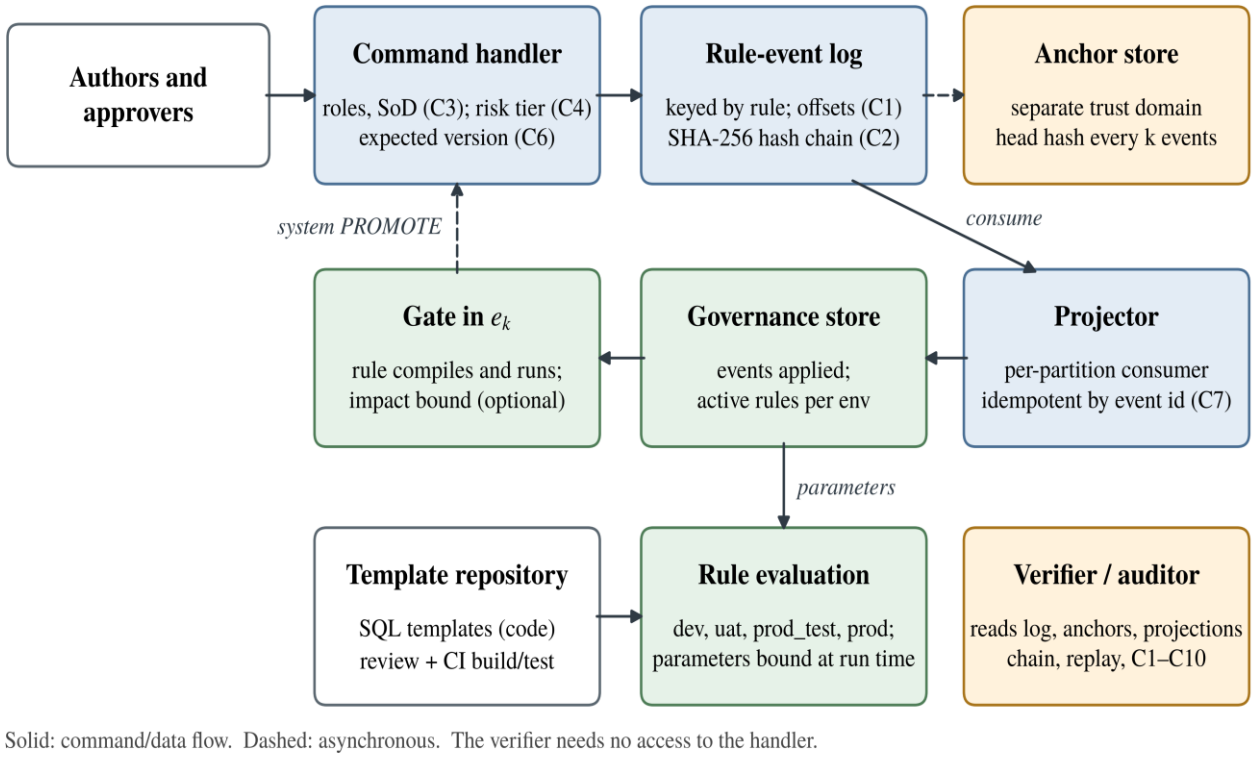



**Figure 1. Architecture. The handler is the only writer of the rule-event log; partition heads are copied to an anchor store outside the platform's administrative domain. Gate evidence in the governance store triggers a system promotion command for the next environment. SoD: separation of duties.**

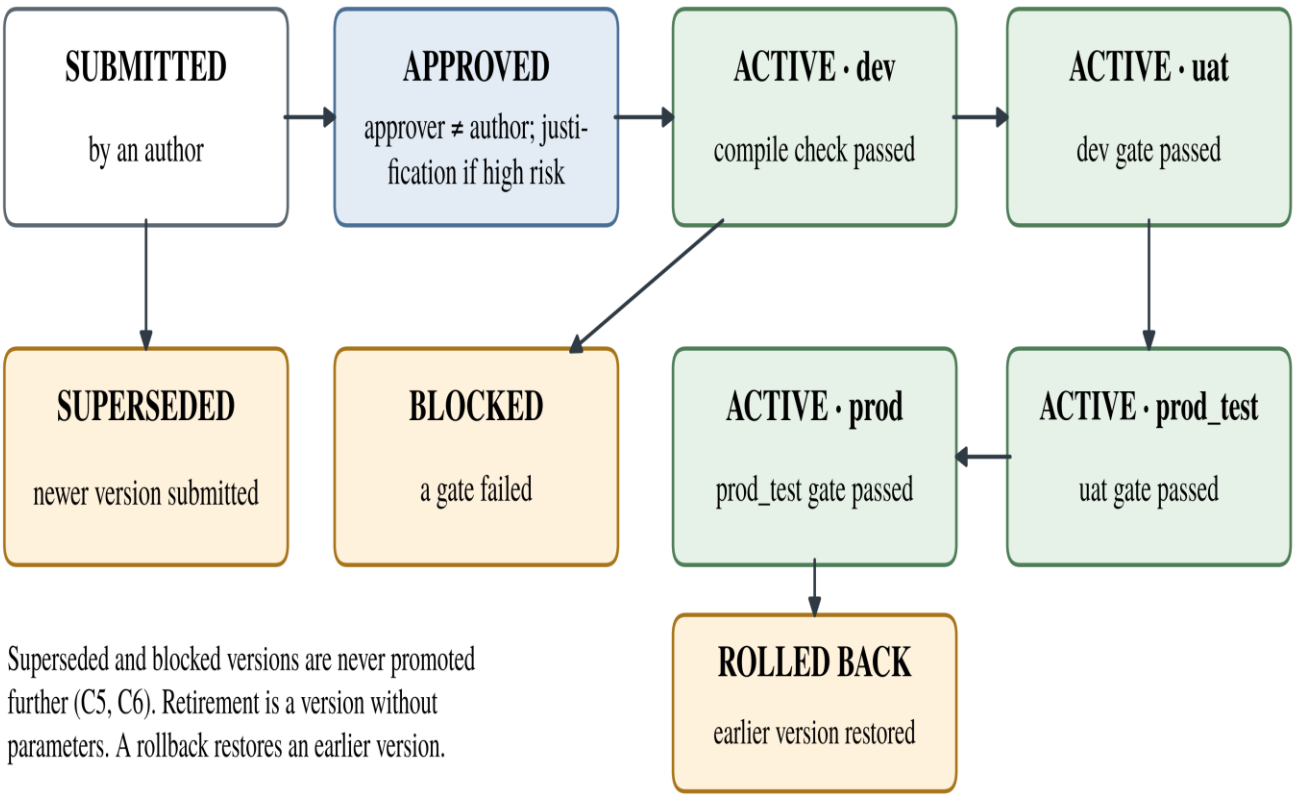



**Figure 2. Lifecycle of a rule version. Promotion beyond dev requires a passed gate in the preceding environment; a**

**failed impact gate blocks the version. A rollback re-activates an earlier production version in all environments and marks newer pending versions as rolled back.**

Algorithm 1 gives the handler. Each rule is an aggregate guarded by a lock, so commands on different rules proceed in parallel. Policy checks run before state checks, so a self-approval or an unjustified high-risk approval is rejected as a policy violation even when the version has meanwhile been superseded. Optimistic concurrency rejects any change based on a stale version; the client re-reads and retries. Only the system role may promote: the projector issues promote($r$, $v$, $e_k$) only after the gate in $e_{k-1}$ has passed (Algorithm 2, line 9), or after the compile check for dev (line 5), and a failed gate is recorded as GATE_FAILED, which blocks the version so that the handler's status check rejects any later promotion. The handler re-checks role, status, and order; the gate itself runs in the projector.

**Algorithm 1.** Command handling

**Input:** command $c = (a, op, r, v, \hat{v}, \theta, j)$, with target version $v$ and expected version $\hat{v}$ (Section 2.1)

1: **if** $op \notin$ perms($\rho(a)$) **then reject** ROLE ▷ C3 (only the system may promote)

2: **if** $\theta \neq \perp$ **and not** valid($\tau(r)$, $\theta$) **then reject** SCHEMA

3: lock aggregate $r$

4: **if** $op$ = approve **then**

5: **if** $a$ = author($r$, $v$) **then reject** SOD ▷ C3

6: **if** risk($r$, $v$) = high **and** $j = \emptyset$ **then reject** C4

7: **if** status($r$, $v$) ≠ submitted **or** $v \neq$ latest($r$) **then reject** STALE

8: **else if** $op$ = promote($r$, $v$, $e_k$) **then** ▷ sent after gate($r$, $v$, $e_{k-1}$) passed, or the compile check if $k = 1$

9: **if** status($r$, $v$) ≠ approved **or** $v \neq$ latest($r$) **then reject** STALE ▷ C5, C6; blocked after GATE_FAILED

10: **if** $k > 1$ **and** active($r$, $e_{k-1}$) ≠ $v$ **then reject** ORDER ▷ C5

11: **else** ▷ create, update, retire, rollback

12: **if** $\hat{v} \neq$ latest($r$) **then reject** CONFLICT ▷ C6

13: $v \leftarrow$ latest($r$) + 1, or the most recent earlier production version for rollback

14: $\varepsilon \leftarrow$ event($c$, $v$, risk tier); append $\varepsilon$ to partition $p(r)$ ▷ C1, C2

15: update aggregate $r$; unlock; **return** $\varepsilon$

Algorithm 2 gives the projector. Because delivery is at least once, it records each event identifier and skips repeats (C7); in production the identifier, the projection update, and the consumer offset would be committed in one transaction. Promotion is strictly sequential (C5). The *execution gate* requires the rule to compile and run on the environment's data and records its violation and scope counts. The optional *impact gate* also compares the rule's flagged-record count with that of the version currently active in the next environment, on the same data, and blocks the version (event GATE_FAILED) if it newly flags more than max(5, 10%) of the in-scope records.

**Algorithm 2.** Projector for partition $p$

**loop**
  **for each** $\varepsilon$ **in** fetch($p$, offset$_p$) **do**
    **if** $\varepsilon$.id ∈ applied **then continue** ▷ C7
    insert $\varepsilon$.id into applied
    **if** $\varepsilon$.type = APPROVED **and** the compile check passes **then** promote($r$, $v$, $e_1$)
    **if** $\varepsilon$ = PROMOTED($r$, $v$, $e_k$) **then** ▷ $S_k$ = state of $e_k$
      $S_k[r] \leftarrow (v, \theta_{r,v})$, removed if $\theta = \perp$
      **if** $k < 4$ **then**
        **if** gate($r$, $v$, $e_k$) passes **then** promote($r$, $v$, $e_{k+1}$) **else** append GATE_FAILED($r$, $v$, $e_k$) ▷ C5
    **if** $\varepsilon$.type = ROLLED_BACK($v'$) **then** $S_e[r] \leftarrow v'$ for all $e$
    offset$_p$ ← offset$_p$ + 1

Algorithm 3 is what an auditor runs. It needs only the log, the anchors, and the projections, and it re-derives the decisions recorded in the log instead of trusting the handler that made them.

**Algorithm 3.** Verification of a history

**Input:** partitions $H_{1..P}$, anchors $A$, projections $S$

1: **for each** partition $p$: recompute (3) starting from $h_{-1}$ = 0x00…00 (the 256-bit zero value); check offsets 0, 1, … ▷ C1, C2
2: **for each** $(p, o, h) \in A$: check $H_p[o].h = h$ ▷ C2
3: replay each rule in offset order; check role, author ≠ approver, recomputed risk and justification, version contiguity, staleness, promotion order, gate outcomes, and rollback targets ▷ C3–C6
4: check every event applied exactly once ▷ C7
5: scan payloads for patient identifiers and non-schema fields ▷ C8
6: check replay($H$) = $S$ ▷ C10

Events reference rules, clinical codes, and thresholds, never patient records (C8). Actor identifiers can be pseudonyms whose mapping to people is stored separately; deleting a mapping makes the corresponding events harder to attribute to a person without rewriting the hash chain [20,31]. This reduces identifiability but does not by itself show that erasure obligations are met: pseudonymized data that can still be attributed with additional information remain personal data under the GDPR (Recital 26) [8], and copies in backups, exports, and derived records must also be handled, so each deployment needs its own legal assessment. Retention (C9) is a deployment setting, such as a Kafka topic with unlimited retention or tiered storage plus archived anchors. In the intended deployment the log is a Kafka topic keyed by rule with an idempotent producer, dbt compiles and tests templates, Snowflake tables hold the projections, and anchors go to storage under a different administrative account; this mapping was not evaluated.

### 2.3 Component roles: correctness versus performance

Not every component serves the same purpose. Table 2 classifies each component by whether the governance guarantees depend on it (correctness), whether it lets an organization *demonstrate* those guarantees to a third party (assurance), or whether it only improves speed, cost, or convenience. Three components were removed experimentally to test the classification (Section 3.5).

**Table 2. Role of each component, the constraints that depend on it, and the evidence.**

| Component | Class | Constraints | Evidence if removed or varied |
|---|---|---|---|
| Policy checks in the handler | Correctness | C3, C4 | Ablation: injected violations accepted and recorded; verifier flags C3/C4 |
| Expected-version check | Correctness | C6 | Contention: 0.314 conflicts per command at 20 authors, all retried |
| Partitioning by rule | Correctness | C1, C4–C6, C10 | Ablation: 13,936 violating events; 66% reconstruction; wrong final state |
| Deduplication by event id | Correctness | C7 | Ablation: events re-applied (C7 in 3 of 3 runs); 412 duplicate promotions |
| Hash chain | Correctness (C2) | C2 | B2: edits of attribution or value undetected |
| External anchors | Correctness (C2) | C2 | Without anchors: 0% of suffix rewrites and truncations detected |
| Execution gate / impact gate | Correctness / DQ safety | C5 | Impact gate stopped 96.2% of faulty changes |
| Projections (governance store) | Performance | — | Rebuildable by replay at 1.0 µs per event (C10) |
| Polling interval, partition count | Performance | — | Median 0.044–0.547 s for 0.01–1.0 s polls |
| Independent verifier | Assurance | C1–C8, C10 | 80 µs per event; not on the critical path |
| Template repository and CI | Correctness of templates | — | Outside the scope of this evaluation |

Correctness: a governance guarantee fails without the component. Assurance: guarantees still hold, but cannot be demonstrated independently. Performance: affects latency, throughput, or cost only.

### 2.4 Research questions and experimental design

The study is a single-process simulation. It reproduces the log properties the design relies on (partitions keyed by rule, offsets, consumer-managed positions, at-least-once delivery) in one Python process, uses DuckDB [35] in place of Snowflake, and runs on one virtual machine; it does not measure a production Kafka or Snowflake deployment. It answers six research questions: **RQ1** latency of approved changes versus the baselines; **RQ2** capacity and behavior under concurrent changes; **RQ3** reconstruction, tamper detection, verification cost, and independent verification; **RQ4** policy enforcement, delivery semantics, and the promotion gate; **RQ5** which components correctness depends on; and **RQ6** sensitivity to the polling interval, the mix of change types, and human approval time.

*Data, rules, and environments.* Synthea [34] generated a Massachusetts population of 10,000 living patients (seed 42) with three years of history, which together with deceased patients gave 11,508 patient records, 1,937,028 numeric observations, 207,809 condition records, and 238,260 medication records. Following the templates of Wang *et al.* [3], 240 rules were derived from the clean data in six templates: physically impossible values (T1, 49 rules), implausible values outside the clean 0.1st–99.9th percentiles (T2, 49), sex-specific diagnoses (T3, 15), age-dependent diagnoses (T4, 40), laboratory units (T5, 47), and medication–diagnosis consistency (T6, 40), each with pass threshold 0.80. Labeled errors were injected at 1%, 5%, and 10% of in-scope records or patients (16,272, 81,361, and 162,722 errors). Production used the 5% file; dev, uat, and prod_test were samples of it with 235, 608, and 1,180 patients.

*Workload.* Author and approver threads issue commands concurrently: 40% parameter updates (a range moved by 1–5% of its width or a threshold raised by 0.01), 30% relaxations (a threshold lowered by 0.05), 15% creations, 10% rollbacks, and 5% retirements, with one approver per five authors. Approvers justify every high-risk approval; to test enforcement they attempt a self-approval before 2% of approvals and an unjustified approval before 2% of high-risk approvals, and authors attempt a direct promotion after 1% of commands. The projector redelivers 5% of events. Light-load runs use 20 authors, 300 commands, about four commands per second, and *idle* targeting (rules with no change in flight); contention runs submit 2,000-command bursts on Zipf-distributed ($s$ = 1.1) rules. Gate runs add 5% *faulty* changes: schema-valid but harmful parameter edits that the risk policy rates low risk (a range collapsed to its middle 20%, a wrong unit, a flipped sex, a minimum age raised by 40 years, or a nonexistent required diagnosis).

*Systems.* **ES** is the prototype (Python 3.13, eight partitions, 0.1 s polling, anchors every 50 events). **B1** deploys each rule as a dbt model (dbt-core 1.12.6); build times were measured by running dbt five times per environment, and scheduling was simulated with them over 10 × 2,000 Poisson approvals (jobs every 5, 15, or 30 min at 50 approvals/h, or on-demand selective builds at 10 and 50 approvals/h). **B2** keeps runtime parameters and the same handler policy code, but stores state in mutable SQLite 3.45.1 tables whose triggers write an audit table, and promotes synchronously. **B3** is B2 with a ledger-style audit: each audit row stores SHA-256(previous hash || row), computed by a database trigger, and the head is anchored every 50 rows, mirroring SQL Ledger [26]. B2 and B3 ran the light-load workload with the same seeds as ES, so all three received identical command sequences. Table 3 summarizes the setup.

**Table 3. Experimental setup.**

| Item | Configuration |
|---|---|
| Data | Synthea (seed 42): 11,508 patients; 1,937,028 observations; 207,809 conditions; 238,260 medications |
| Rules and errors | 240 rules in 6 templates; labeled errors 1% / 5% / 10%: 16,272 / 81,361 / 162,722 |
| Environments | dev 235, uat 608, prod_test 1,180 patients; prod = full 5% file |
| Light load | 20 authors, 4 approvers, 300 commands per run, idle targeting; ES 10 runs + 10 replication runs; B2 10; B3 10 |
| Contention, load | 1–20 authors with 2,000-command bursts (10 runs each); offered load 1–40 changes/s (6 levels × 3 runs) |
| Gate, ablations | Impact gate with 5% faulty changes (10 runs, plus 3 diagnostic re-runs logging every gate decision); no deduplication, no policy checks, random partitions (3 runs each) |
| Sensitivity | Polling 0.01, 0.05, 0.5, and 1.0 s (4 levels × 3 runs) plus the 0.1 s default (the 10 light-load runs); 4 alternative command mixes × 3 runs; approval-time model |
| Machine | One virtual machine, 2 vCPU, 7 GB RAM, Linux; Python 3.13, DuckDB 1.5.6, SQLite 3.45.1, dbt-core 1.12.6 |

*Metrics and statistics.* M1 is propagation latency, (2). M2 is throughput and conflict rate. M3 is reconstruction accuracy: at 200

random instants per run, the state replayed from the history is compared for every (rule, environment) pair with an oracle written by the handler at decision time. Because oracle and log come from the same handler, M3 shows that the log records the handler's decisions, not that the decisions were correct; M7 closes that gap. M4 is tamper detection over 200 trials per attack. M5 counts blocked violations and absorbed duplicates. M6 is rollback latency. M7 is the outcome of Algorithm 3, implemented separately from the handler, projector, and oracle, with a mutation test that injects histories violating one constraint each. A control check compares the violation sets of all 240 rules under dbt and under runtime binding. Because changes within a run share a machine state, they are not independent; designs are therefore compared on per-run medians ($n$ = 10 runs per design) with the Mann–Whitney U test [36] and Cliff's $\delta$ [37]. Pooled medians are reported with 95% confidence intervals from a cluster bootstrap that resamples whole runs (10,000 resamples), which preserves the dependence of changes within a run [38]; with 10 runs per design these intervals are approximate.

## 3. Results

### 3.1 RQ1: Propagation latency

Table 4 and Figure 3 report M1. Under light load, ES activated approved changes in production in a median of 0.092 s (95% confidence interval 0.090–0.094 s; p99 0.167 s); the replication runs gave 0.085 s. The fastest rebuild baseline, an on-demand selective build at 10 approvals/h, took 25.5 s, 277 times longer, because a dbt invocation costs about 6.3 s per environment even for one model; scheduled rebuilds took 241, 542, and 1,000 s for 5-, 15-, and 30-minute schedules. On per-run medians, every B1 setting was slower than ES in all 10 of 10 runs (Cliff's $\delta$ = 1.00, $p$ < 0.001).

B2 was faster still at 0.0091 s, and B3 at 0.0114 s, about 75 ms and 73 ms below the ES replication on identical command sequences ($p$ < 0.001 on per-run medians). B2 and B3 apply a change synchronously inside the approving request, whereas in ES the approval waits for a polling consumer (up to 0.1 s here) before the same gates run. The sub-second result is therefore a property of binding parameters at runtime, and hashing every audit row cost B3 only 2.3 ms per change over B2.

**Table 4. Approval-to-production latency (s).**

| Design | Median (95% conf. interval) | p95 | p99 | Run medians, mean ± SD | Ratio |
|---|---|---|---|---|---|
| ES, light load | 0.092 (0.090–0.094) | 0.143 | 0.167 | 0.092 ± 0.003 | 1 |
| ES, light load (replication) | 0.085 (0.082–0.087) | 0.132 | 0.148 | 0.084 ± 0.004 | 0.92 |
| B2: mutable table + audit trigger | 0.0091 (0.0090–0.0093) | 0.016 | 0.020 | 0.0091 ± 0.0002 | 0.10 |
| B3: table + hash-chained audit | 0.0114 (0.0111–0.0117) | 0.020 | 0.025 | 0.0114 ± 0.0005 | 0.12 |
| B1: dbt on-demand, 10/h | 25.5 | 33.3 | 47.5 | 25.5 ± 0.0 | 277 |
| B1: dbt on-demand, 50/h | 25.8 | 59.7 | 81.2 | 25.8 ± 0.0 | 281 |
| B1: dbt every 5 min | 241 | 375 | 387 | 240 ± 4 | 2,622 |
| B1: dbt every 15 min | 542 | 944 | 981 | 542 ± 9 | 5,902 |
| B1: dbt every 30 min | 1,000 | 1,801 | 1,871 | 1,002 ± 28 | 10,887 |

Ratio: pooled median relative to ES, light load. Approved changes: 2,699 (ES), 2,692 (replication), 2,696 (B2), 2,695 (B3), and 20,000 simulated per B1 setting. B1 latencies come from simulated scheduling over measured dbt build times (four environments: full build 90.0 s, selective build 25.5 s). Run medians: 10 runs per ES, B2, and B3 configuration; 10 simulation replications per B1 setting. Confidence intervals: cluster bootstrap over runs (Section 2.4).

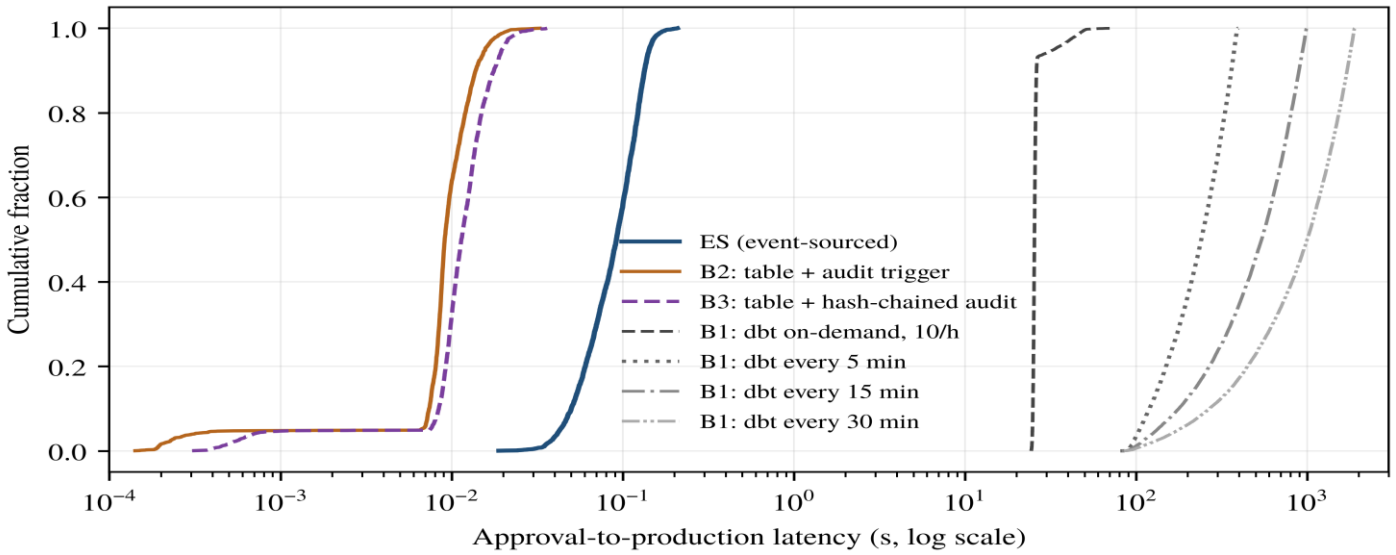



**Figure 3. Cumulative distribution of approval-to-production latency (log scale). ES, B2, and B3 all stay below one second; the B1 rebuild pipelines are two to four orders of magnitude slower. The lowest 5% of B2 and B3 values are retirements, which run no gate.**

### 3.2 RQ2: Capacity and contention

Figure 4 shows ES latency as offered load grows. The median stayed within 0.089–0.108 s from 0.56 to 17.3 approved changes per second and rose at 36.7 changes/s (median 0.240 s, p95 1.7 s), the knee of the curve on this two-core machine. The on-demand rebuild baseline can deploy at most one change per 25.5 s (about 141 per hour), so ES sustained roughly 440 times its capacity before degrading. Under 2,000-command bursts the handler accepted about 3,070–3,360 commands per second; conflicts grew with the number of authors editing the same popular rules (0.043, 0.127, 0.224, and 0.314 per command for 1, 5, 10, and 20 authors), and every conflict was resolved by a retry. Because the bursts exceed steady-state capacity, the projector worked through a backlog and median latency rose to 11.9–13.9 s. With one author changing hot rules in rapid succession, 9,716 of 14,493 approved versions were superseded before reaching production and were never promoted (C6).

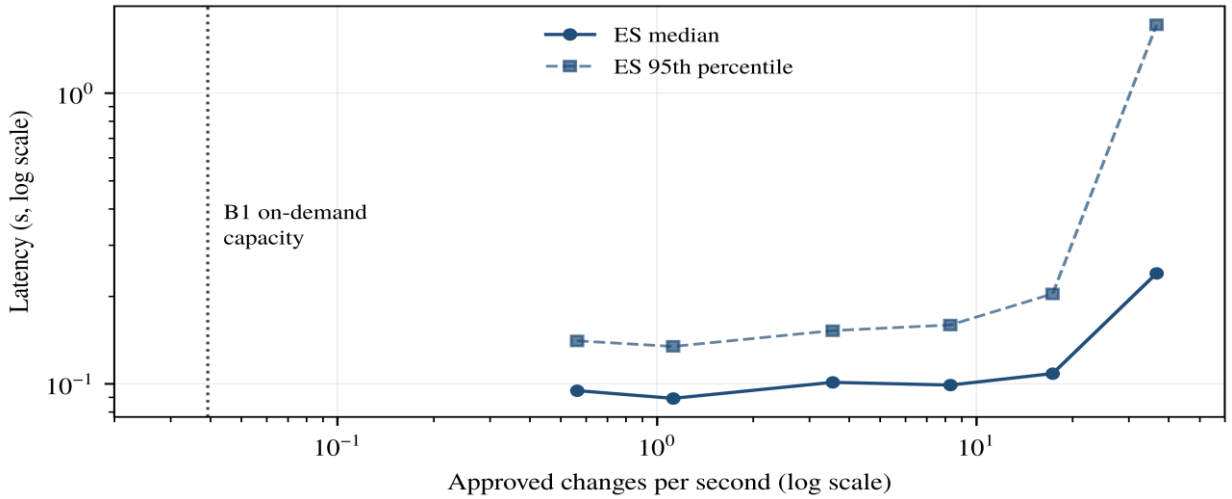



**Figure 4. ES propagation latency versus offered load (log–log). The dotted line marks the maximum rate of the on-demand rebuild baseline.**

### 3.3 RQ3: Reconstruction, tamper evidence, verification cost, and independent verification

*Reconstruction.* Across the 80 original runs, all 20,100,829 sampled (rule, environment, instant) states replayed from the log matched the oracle, the final projection equaled the oracle in every run, and no rule showed a version gap or duplicate. In the first contention block, 4 of 13.2 million checks exposed an oracle bug; after the oracle alone was fixed, the re-run matched all 11,083,883 checks. The replication runs matched 2,108,097 of 2,108,097 checks. The audit tables of B2 (2,129,261 of 2,129,261) and B3 (2,130,149 of 2,130,149) also reconstructed every sampled state: an intact history answers point-in-time questions in all three designs.

*Tamper evidence.* Table 5 reports M4. A hash chain, whether over events (ES) or audit rows (B3), detected every edit, deletion, reordering, and insertion that left the rest of the history untouched. It did not detect an insider who rewrote a record and recomputed all later hashes, or who truncated the newest records, because the result is again a valid chain; external anchors closed both gaps: with an anchor for every record, ES and B3 detected all seven tested attack types. The experiments took anchors from the untampered log, which models anchors published synchronously with each append; Section 4.3 states the conditions under which this carries over to a deployment. B2's structural checks detected deletions and back-dated insertions and 51% of reorderings, but none of the changes to who made a change or to a recorded value, and no consistent rewrite or truncation. Tamper evidence therefore comes from the chain and its anchors, not from event sourcing as such.

**Table 5. Tamper detection rate (%), 200 trials per attack type**

| | **ES: anchor every *k* events** | | | | **B3: anchor every *k* rows** | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| **Attack** | **none** | **50** | **1** | **B2** | **none** | **50** | **1** |
| Edit a field (attribution) | 100 | 100 | 100 | 0 | 100 | 100 | 100 |
| Edit and recompute own hash | 100 | 99.5 | 100 | 0 | 100 | 100 | 100 |
| Delete a record | 100 | 99 | 100 | 100 | 100 | 100 | 100 |
| Swap adjacent records | 100 | 100 | 100 | 51 | 100 | 100 | 100 |
| Insert a forged record | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Rewrite suffix (insider) | 0 | 95.5 | 100 | 0 | 0 | 99.5 | 100 |
| Truncate tail (insider) | 0 | 67 | 100 | 0 | 0 | 94.5 | 100 |

ES: hash chain over events; none = chain only. B2: contiguous keys, key counter, non-decreasing timestamps. B3: hash chain over audit rows. Attacks use the same procedures in all designs; a B2 row has no hash, so the second attack edits a recorded value instead. Misses at sparse anchoring involve the newest records after the last anchor.

*Verification cost.* Recomputing the ES hash chain and checking anchors took 12 µs per event, replaying the log into projections 1.0 µs per event, and the full independent verifier 80 µs per event over 67,261 events in 32 logs; computing the hash on append took 53 µs per event. Verifying a B3 chain took 9.0 µs per row. At these rates, a year of 5,000 events per day would take about 22 s to check against its anchors and 2.4 min to verify fully, so routine verification is affordable.

*Independent verification.* The verifier ran on all 22 retained ES logs from the replication and workload-mix runs (46,255 events) and on the 10 gate runs, and found 0 violations of C1–C8 and C10. In the mutation test it flagged the targeted constraint in 480 of 480 injected histories, covering a duplicated event (C1), an edit without re-hashing (C2), a self-approval and a promotion by an author (C3), an unjustified high-risk approval and a relaxation labeled low risk (C4), promotions that skip environments or lack approval (C5), a version gap (C6), an event never applied (C7), a patient identifier in a payload (C8), and a projection that differs from the log (C10). C9 cannot be tested in a run of minutes.

### 3.4 RQ4: Policy enforcement, delivery semantics, and the promotion gate

Table 6 summarizes M5. All injected violations were blocked: 2,075 self-approvals, 785 unjustified high-risk approvals, and 1,069 direct promotions by authors across all ES runs, and every violation injected into B2 and B3, which reuse the same handler code; enforcement at the command layer does not depend on how history is stored. All 20,959 duplicate deliveries were skipped. Rollbacks restored the previous production version in all four environments in a median of 0.051 s (p95 0.098 s).

*Promotion gate.* With the impact gate enabled, 131 faulty changes were approved; the gate blocked 126 of them (96.2%: 108 in

dev, 12 in uat, 6 in prod_test), and 5 reached production. The gate also blocked 38 of 2,566 ordinary changes (1.5%) and raised the median latency of ordinary changes to 0.094 s, against 0.085 s without it. Re-running three seeds with every gate decision logged explained both outcomes. The faulty changes that reached production were sex-specific diagnosis rules with at most 15 in-scope records in any test environment, too few to exceed the gate's five-record minimum. Of the 10 ordinary changes blocked, five were new rules, which have no baseline; the other five narrowed a range at a bound where most observations lie and flagged 258–601 instead of 11–54 records, a change that merits review. A gate can catch only what the test environments exercise.

**Table 6. Policy enforcement, delivery, gate, and verification outcomes.**

| Check | Result |
|---|---|
| *Command-layer enforcement (all ES runs)* | |
| Self-approval blocked (C3) | 2,075 / 2,075 |
| High-risk approval without justification blocked (C4) | 785 / 785 |
| Direct promotion by an author blocked (C3, C5) | 1,069 / 1,069 |
| Duplicate deliveries absorbed (C7) | 20,959 / 20,959 |
| Injected violations blocked in B2 and B3 | 207 / 207 |
| *Impact gate, 5% faulty changes (10 runs)* | |
| Faulty changes blocked before production | 126 / 131 (96.2%) |
| Faulty changes that reached production | 5 |
| Ordinary changes blocked for review | 38 / 2,566 (1.5%) |
| *Independent verifier* | |
| Violations in 32 retained logs (C1–C8, C10) | 0 |
| Mutated histories flagged (12 types × 40) | 480 / 480 |

### 3.5 RQ5: Which components correctness depends on

Table 7 reports the three ablations that test the classification in Table 2. Without command-layer policy, all 15 injected self-approvals and all 13 unjustified high-risk approvals were accepted and written to the log, and the verifier reported exactly these as C3 and C4 violations: removing enforcement does not hide violations from an auditor, but it no longer prevents them. (The 6 injected author promotions targeted a superseded version and were stopped by the state checks.) Without deduplication, redelivered events were applied to the projections a second time, which the verifier flagged as C7 in 3 of 3 runs, and re-processed promotions issued 412 duplicate promotion events. The final state was still correct in 3 of 3 runs, because setting a rule's active version is idempotent, but the history contained duplicate promotions that an auditor would have to explain. Without rule-keyed partitioning, events of one rule were spread across partitions and applied out of order: the verifier reported 13,936 violating events, point-in-time reconstruction fell to 66% accuracy, the projector hit 30 write conflicts, and the final state was wrong in 3 of 3 runs. Per-rule ordering is therefore a correctness requirement, whereas the number of partitions and the polling interval only affect performance.

**Table 7. Component ablations (3 light-load runs each): violations reported by the independent verifier.**

| Ablation | C1 | C3 | C4 | C5 | C6 | C7 runs | C10 runs | Recon. (%) | Final state correct |
|---|---|---|---|---|---|---|---|---|---|
| No command-layer policy | 0 | 15 | 13 | 0 | 0 | 0 of 3 | 0 of 3 | 100 | 3 of 3 |
| No deduplication | 0 | 0 | 0 | 0 | 0 | 3 of 3 | 0 of 3 | 100 | 3 of 3 |
| Random partitions (not keyed by rule) | 4,974 | 0 | 307 | 4,517 | 4,138 | 0 of 3 | 3 of 3 | 66 | 0 of 3 |

C1–C6: violating events summed over the three runs. C7, C10: runs in which the check failed (each is checked once per run). Recon.: share of sampled (rule, environment, instant) states reconstructed correctly from the log. C2, C8, and C9 were unaffected and are omitted. With all components present, the same workload produced no violations (Section 3.3).

### 3.6 RQ6: Sensitivity to polling, workload mix, and approval time

*Polling interval.* Four polling levels were run three times each, and the 0.1 s default comes from the 10 light-load runs, giving five settings in all. The ES median fell from 0.547 s at a 1.0 s poll to 0.315, 0.092, 0.065, and 0.044 s at 0.5, 0.1 (default), 0.05, and 0.01 s. Most of the gap to B2 is polling delay, which a push-based consumer would remove; the remaining 35 ms at the 0.01 s poll is the cost of the asynchronous projection path on this machine.

*Workload mix.* Table 8 varies the share of change types. The median stayed within 0.083–0.085 s across five mixes, even when 70% of commands were relaxations requiring justification or 35% were rollbacks; reconstruction stayed exact and the verifier found no violation in any mix.

**Table 8. Sensitivity to the command mix (ES, light load).**

| Mix | U/R/C/RB/T (%) | *n* | Median (s) | p95 (s) | High risk | Violations |
|---|---|---|---|---|---|---|
| Default | 40/30/15/10/5 | 2,692 | 0.085 | 0.132 | 45.2% | 0 |
| Update-heavy | 70/10/10/5/5 | 850 | 0.084 | 0.135 | 27.3% | 0 |
| Relaxation-heavy | 10/70/10/5/5 | 852 | 0.085 | 0.132 | 78.8% | 0 |
| Rollback-heavy | 30/20/10/35/5 | 565 | 0.084 | 0.137 | 44.6% | 0 |
| Creation-heavy | 20/15/55/5/5 | 860 | 0.083 | 0.131 | 22.1% | 0 |

U: parameter update; R: relaxation; C: creation; RB: rollback; T: retirement. *n*: approved changes that reached production. High risk: share of approvals that required a justification. Default: replication runs (10); other mixes: 3 runs each. Reconstruction accuracy was 100% in every mix.

*Approval time.* Human approval, which in practice waits on the data owner, was excluded from (2). Table 9 adds a log-normal approval wait (median 15 min, 2 h, or 1 day; $\sigma = 1$) to the measured propagation latencies. Once approval takes hours, runtime binding (ES or B3) makes propagation a negligible share of lead time, whereas a 30-minute rebuild schedule still adds a visible delay when approvals are fast. The speed-up that matters to users is therefore largest for routine, quickly approved changes.

**Table 9. Modeled end-to-end lead time (approval wait plus measured propagation): median in minutes and share of**

**mean lead time spent in propagation.**

| Design | Approval median 15 min | Approval median 2 h | Approval median 1 day |
|---|---|---|---|
| ES (event-sourced) | 15.0 / <0.1% | 120 / <0.1% | 1,441 / <0.1% |
| B3: hash-chained table | 15.0 / <0.1% | 120 / <0.1% | 1,441 / <0.1% |
| B1: dbt on-demand | 15.4 / 1.7% | 120 / 0.2% | 1,441 / <0.1% |
| B1: dbt every 5 min | 19.1 / 14.0% | 124 / 2.0% | 1,444 / 0.2% |
| B1: dbt every 30 min | 33.8 / 40.1% | 137 / 7.7% | 1,457 / 0.7% |

Approval wait is log-normal with $\sigma = 1$ (an assumption, not a measurement); propagation is sampled from the measured distributions. Cells: median lead time (min) / propagation share of mean lead time.

### 3.7 Control: detection is unchanged

For all 240 rules, the violation sets produced by dbt-materialized models (B1) and by runtime binding (ES) were identical (0 of 240 differed). Against the injected ground truth in the 5% file, recall was 0.998 or higher for every template; precision was 1.00 for T1 and T6, 0.98 for T3, 0.99 for T4, 0.61 for T5, and 0.48 for T2. The low precision of T2 and T5 reflects rule design, not governance: they flag the rare tails of the clean distribution and legitimate minority units, the "possible but implausible" category that Wang *et al.* advise validating before use [3].

## 4. Discussion

### 4.1 Three design principles

The results separate three effects. *Speed* comes from binding parameters at runtime: ES, B2, and B3 all activated approved changes in well under a second, while rebuild pipelines took tens of seconds to tens of minutes, mostly in fixed per-invocation overhead. *Tamper evidence* comes from a hash chain anchored outside the platform: ES and B3 detected all seven tested attack types when every record was anchored, and the unchained audit table of B2 detected none of the attribution or value edits. *Enforcement* comes from the command layer: every design that ran the same policy code blocked every injected violation. Event sourcing is one way to combine the three, at a cost of about 75 ms per change at a 0.1 s polling interval.

### 4.2 Event sourcing or a ledger table?

B3 matched ES on reconstruction and tamper detection and was about 73 ms faster per change, so the choice between them rests on other properties. Event sourcing is preferable when rule state must reach several environments or stores, because each consumer derives its projection from one ordered log and can rebuild it by replay; when the record must capture decisions rather than row images, such as approvals, justifications, gate outcomes, and rollbacks, which is what Algorithm 3 checks; and when other systems need to subscribe to rule changes. A ledger table is the simpler choice when all rule state lives in one database that already offers verifiable history, provided that the approval workflow, risk tiers, and promotion gates are implemented on top of it. In both designs the same command-layer policy and the same anchoring discipline are what make the record trustworthy.

### 4.3 Anchoring as a design parameter

A hash chain stored next to the data it protects is not enough, because anyone who can rewrite the log can rewrite the chain. A rewrite or truncation escapes detection only if it touches nothing at or before the newest anchor, so anchoring every $k$ records leaves up to $k - 1$ records per partition exposed, and anchoring every record closes the window only if each anchor is published to, and acknowledged by, a store that the adversary cannot rewrite before the record is acknowledged. With asynchronous or batched publication, records written after the newest acknowledged anchor stay exposed for the publication delay, so the window depends on both $k$ and that delay. Since anchors are small and rule changes infrequent, anchoring every event, or on a

short timer, is affordable.

### 4.4 Implications for practice

Keep templates in code review and CI, and move parameters to governed runtime state; this alone removes the build from the critical path. Enforce separation of duties and risk tiers in the command layer, and record the justification reference in the event itself. Store the history as a hash chain with anchors in a separate trust domain and run the verifier routinely. Do not rely on risk tiers alone: in our gate runs every faulty change was rated low risk, and only an impact check on representative test data stopped most of them.

### 4.5 Threats to validity

*Internal.* The log is an in-process stand-in for Kafka and DuckDB stands in for Snowflake, so absolute latencies reflect one machine and omit network hops, broker replication, and warehouse queueing; the relative ordering of designs is more transferable than the absolute numbers. The oracle is written by the handler under test, which is why the independent verifier, its mutation test, and the ablations were added; the verifier was written by the same author from the same specification, so a shared misunderstanding of the specification would not be caught. *Construct.* Approval time is excluded from the measured latency and only modeled; the impact gate's threshold was set a priori, not tuned. *External.* Data are synthetic, rules come from six templates, the workload mix is assumed (although Table 8 shows stable results), B1 scheduling is simulated from measured build times, and B2 and B3 use SQLite rather than a server database.

### 4.6 Limitations and future work

The prototype does not implement event schema evolution [20] or crypto-shredding of actor identities. Projections remain mutable caches, so a short-lived edit to a projection that is reverted before the next comparison with the log would go unnoticed; stamping each evaluation result with the rule version and a hash of the parameters it used would close this gap. Future work will deploy the design on Kafka, dbt, and Snowflake, measure lead time including approval in a real organization, evaluate gates on test environments selected for rule coverage, and route machine-proposed rules [18] through the same approvals.

## 5. Conclusion

This paper modeled the DQ rule lifecycle as an event-sourced, hash-chained history with command-layer policy and gated promotion and evaluated it in a simulation against three baselines. Binding parameters at runtime reduced approval-to-production latency from tens of seconds or minutes to 0.092 s; a hash chain with external anchors made the tested insider attacks detectable, provided anchors are published before records are acknowledged; and command-layer policy blocked every injected violation. Event sourcing added replayable projections across environments and decision-level verification at a cost of about 75 ms per change.

**Contribution.** The paper contributes three design principles for governing DQ rules (runtime parameter binding, an anchored hash chain, and policy enforced before history is written), an architecture that realizes them, a classification of which components correctness depends on, and a reusable evaluation framework with baselines, attacks, ablations, and an independent verifier. **Conditions of usefulness.** The design pays off where rule parameters change often, changes must be approved and later proven to auditors, and rule state must reach several environments or stores; where all rule state lives in one database with verifiable history, a ledger table with the same command-layer policy is simpler. **Validation before production adoption.** Adopters should repeat the evaluation on their own Kafka (or equivalent log), warehouse, and network; tune promotion gates on test environments that cover each rule's codes; implement schema evolution, transactional offset commits, and pseudonymized actor identities; place anchors under a separate administrative authority; and measure end-to-end lead time, including human approval, in a pilot before relying on the design for regulated workloads.

### Acknowledgements

The author used Claude (Anthropic), a generative AI assistant, to summarize the cited literature and draft text (Section 1); to draft text, algorithms, and figures and write the simulation, baseline, and verifier code (Section 2); to run the experiments and produce the analyses, tables, and figures (Section 3); and to draft and edit the abstract and Sections 4 and 5. The author defined the problem and scenario, made the design and evaluation decisions, reviewed all AI-assisted output, checked every reported number and reference, and takes full responsibility for the content.

**Funding information**

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

**Author contributions**

The author: conceptualization, problem definition, methodology, validation, and writing (review and editing). Software, formal analysis, visualization, and drafting were carried out with AI assistance under the author's direction, as described in the Acknowledgements.

**Conflict of interest**

The author declares no conflict of interest.

**Data and code availability**

All data are synthetic and were generated with Synthea (seed 42). The code, baselines, verifier, dependency versions, raw per-run results, and figure scripts are provided to reviewers as an anonymized archive (Supplementary_Code_and_Data_v4.zip); the public repository link is withheld for anonymity. No patient data or employer data were used.

**References**

**1.** M. G. Kahn et al., "A harmonized data quality assessment terminology and framework for the secondary use of electronic health record data," eGEMs, vol. 4, no. 1, Art. no. 1244, 2016, doi: 10.13063/2327-9214.1244.

**2.** N. G. Weiskopf and C. Weng, "Methods and dimensions of electronic health record data quality assessment: Enabling reuse for clinical research," J. Am. Med. Inform. Assoc., vol. 20, no. 1, pp. 144–151, 2013, doi: 10.1136/amiajnl-2011-000681.

**3.** Z. Wang, J. R. Talburt, N. Wu, S. Dagtas, and M. N. Zozus, "A rule-based data quality assessment system for electronic health record data," Appl. Clin. Inform., vol. 11, no. 4, pp. 622–634, 2020, doi: 10.1055/s-0040-1715567.

**4.** Y. Mohamed et al., "Electronic health record data quality variability across a multistate clinical research network," J. Clin. Transl. Sci., vol. 7, no. 1, Art. no. e130, 2023, doi: 10.1017/cts.2023.548.

**5.** C. Blacketer, F. J. Defalco, P. B. Ryan, and P. R. Rijnbeek, "Increasing trust in real-world evidence through evaluation of observational data quality," J. Am. Med. Inform. Assoc., vol. 28, no. 10, pp. 2251–2257, 2021, doi: 10.1093/jamia/ocab132.

**6.** Security Standards: Technical Safeguards—Audit Controls, 45 C.F.R. § 164.312(b). [Online]. Available: https://www.ecfr.gov/current/title-45/subtitle-A/subchapter-C/part-164/subpart-C/section-164.312

**7.** Policies and Procedures and Documentation Requirements—Time Limit, 45 C.F.R. § 164.316(b)(2)(i). [Online]. Available: https://www.ecfr.gov/current/title-45/subtitle-A/subchapter-C/part-164/subpart-C/section-164.316

**8.** Regulation (EU) 2016/679 of the European Parliament and of the Council (General Data Protection Regulation), Off. J. Eur. Union, vol. L 119, pp. 1–88, May 2016. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj

**9.** J. Declerck et al., "Assessing data quality in heterogeneous health care integration: Simulation study of the AIDAVA framework," JMIR Med. Inform., vol. 13, Art. no. e75275, 2025, doi: 10.2196/75275.

**10.** S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger, "Automating large-scale data quality verification," Proc. VLDB Endow., vol. 11, no. 12, pp. 1781–1794, 2018, doi: 10.14778/3229863.3229867.

**11.** E. Breck, N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data validation for machine learning," in Proc. Conf. Mach. Learn. Syst. (SysML), Stanford, CA, USA, 2019. [Online]. Available: https://research.google/pubs/data-validation-for-machine-learning/

**12.** S. S. Kumar, R. Chandra, and S. Agarwal, "Rule based complex event processing for an air quality monitoring system in smart city," 2024, arXiv:2403.14701, doi: 10.48550/arXiv.2403.14701.

**13.** A. Polimeno, C. Braghin, M. Anisetti, and C. A. Ardagna, "Maximizing data quality while ensuring data protection in service-based data pipelines," J. Big Data, vol. 12, Art. no. 62, 2025, doi: 10.1186/s40537-025-01118-5.

**14.** Open Policy Agent, "OPA documentation," openpolicyagent.org. [Online]. Available: https://openpolicyagent.org/docs (accessed Oct. 22, 2026).

**15.** M. Zaharia et al., "Accelerating the machine learning lifecycle with MLflow," IEEE Data Eng. Bull., vol. 41, no. 4, pp. 39–45, 2018. [Online]. Available: http://sites.computer.org/debull/A18dec/p39.pdf

**16.** Microsoft, "Manage model lifecycle using the Workspace Model Registry (legacy)," Azure Databricks documentation. [Online]. Available: https://docs.azure.cn/en-us/databricks/mlflow/model-registry (accessed Oct. 22, 2026).

**17.** M. Comuzzi, J. Ko, and F. Maggi, "A language to model and simulate data quality issues in process mining," ACM J. Data Inf. Qual., vol. 17, no. 2, pp. 1–36, 2025, doi: 10.1145/3743144.

**18.** F. Zhou, Z. Wang, Q. Liu, J. Li, and P. Liu, "Programming every example: Lifting pre-training data quality like experts at scale," in Proc. 42nd Int. Conf. Mach. Learn. (ICML), PMLR, vol. 267, 2025, pp. 79281–79327. [Online]. Available: https://proceedings.mlr.press/v267/zhou25aa.html

**19.** M. Kleppmann, Designing Data-Intensive Applications. Sebastopol, CA, USA: O'Reilly Media, 2017, ISBN: 978-1-449-37332-0.

**20.** M. Overeem, M. Spoor, S. Jansen, and S. Brinkkemper, "An empirical characterization of event sourced systems and their schema evolution—Lessons from industry," J. Syst. Softw., vol. 178, Art. no. 110970, 2021, doi: 10.1016/j.jss.2021.110970.

**21.** J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in Proc. 6th Int. Workshop Networking Meets Databases (NetDB), Athens, Greece, 2011. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf

**22.** B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," ACM Trans. Inf. Syst. Secur., vol. 2, no. 2, pp. 159–176, 1999, doi: 10.1145/317087.317089.

**23.** S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in Proc. 18th USENIX Security Symp., Montreal, QC, Canada, 2009, pp. 317–334. [Online]. Available: https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/efficient-data-structures-tamper-evident

**24.** S. Haber and W. S. Stornetta, "How to time-stamp a digital document," J. Cryptol., vol. 3, no. 2, pp. 99–111, 1991, doi: 10.1007/BF00196791.

**25.** B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," IETF, RFC 6962, Jun. 2013, doi: 10.17487/RFC6962.

**26.** P. Antonopoulos, R. Kaushik, H. Kodavalla, S. Rosales Aceves, R. Wong, J. Anderson, and J. Szymaszek, "SQL Ledger: Cryptographically verifiable data in Azure SQL Database," in Proc. Int. Conf. Manage. Data (SIGMOD), 2021, doi: 10.1145/3448016.3457558.

**27.** X. Yang, Y. Zhang, S. Wang, B. Yu, F. Li, Y. Li, and W. Yan, "LedgerDB: A centralized ledger database for universal audit and verification," Proc. VLDB Endow., vol. 13, no. 12, pp. 3138–3151, 2020, doi: 10.14778/3415478.3415540.

**28.** K. Kulkarni and J.-E. Michels, "Temporal features in SQL:2011," ACM SIGMOD Rec., vol. 41, no. 3, pp. 34–43, 2012, doi: 10.1145/2380776.2380786.

**29.** M. Armbrust et al., "Delta Lake: High-performance ACID table storage over cloud object stores," Proc. VLDB Endow., vol. 13, no. 12, pp. 3411–3424, 2020, doi: 10.14778/3415478.3415560.

**30.** Apache Software Foundation, "Iceberg table spec," Apache Iceberg. [Online]. Available: https://iceberg.apache.org/spec/ (accessed Oct. 22, 2026).

**31.** E. Politou, E. Alepis, and C. Patsakis, "Forgetting personal data and revoking consent under the GDPR: Challenges and proposed solutions," J. Cybersecurity, vol. 4, no. 1, Art. no. tyy001, 2018, doi: 10.1093/cybsec/tyy001.

**32.** B. Fitzgerald, K.-J. Stol, R. O'Sullivan, and D. O'Brien, "Scaling agile methods to regulated environments: An industry case study," in Proc. 35th Int. Conf. Softw. Eng. (ICSE), San Francisco, CA, USA, 2013, pp. 863–872, doi: 10.1109/ICSE.2013.6606635.

**33.** dbt Labs, "dbt Core," GitHub repository. [Online]. Available: https://github.com/dbt-labs/dbt-core (accessed Oct. 22, 2026).

**34.** J. Walonoski et al., "Synthea: An approach, method, and software mechanism for generating synthetic patients and the synthetic electronic health care record," J. Am. Med. Inform. Assoc., vol. 25, no. 3, pp. 230–238, 2018, doi: 10.1093/jamia/ocx079.

**35.** M. Raasveldt and H. Mühleisen, "DuckDB: An embeddable analytical database," in Proc. Int. Conf. Manage. Data (SIGMOD), Amsterdam, The Netherlands, 2019, pp. 1981–1984, doi: 10.1145/3299869.3320212.

**36.** H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," Ann. Math. Statist., vol. 18, no. 1, pp. 50–60, 1947, doi: 10.1214/aoms/1177730491.

**37.** N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," Psychol. Bull., vol. 114, no. 3, pp. 494–509, 1993, doi: 10.1037/0033-2909.114.3.494.

**38.** B. Efron and R. J. Tibshirani, An Introduction to the Bootstrap. New York, NY, USA: Chapman & Hall, 1993, doi: 10.1201/9780429246593.