NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 4 Issue 3, Pg. No. 60-86, AIJCST-V4I3P106, 2022
https://doi.org/10.63282/3117-5481/AIJCST-V4I3P106

Original Article

Unified State Machine Normalization across Heterogeneous Clinical Systems: An Architectural Pattern for Enterprise Status Federation

***Neeraj Gaddam**
Independent Researcher, Leander, Texas, USA.

Abstract:

Hospitals run separate applications for registration, insurance verification, laboratory, pharmacy and billing, and each one describes the status of a workflow in its own words. Staff who need to know where a patient's process stands make telephone calls, search several screens, or guess. This paper specifies an architectural pattern for enterprise status federation that leaves the source systems unchanged. Adapters extract committed changes as events, a versioned mapping registry translates each native status into one of eight canonical states and keeps the original value, a transition guard checks every encounter against a table of legal moves, and a read-only store serves one dashboard. We give a formal model with three propositions: pairwise translation needs n(n − 1) translators where a hub needs n adapters, replay under recorded mapping versions rebuilds the store, and a lenient guard flags a subset of what a strict guard flags. A synthetic study with illustrative parameters examines four design questions. Timestamp polling missed 2.0% of the status changes in a simulated day, while log-based capture missed none and delivered the median change in 0.5 s against 30.0 s. In an in-process benchmark, a pre-normalized store answered aggregate queries 3.2 to 5.0 times faster than query-time translation, and in a fan-out model a twelve-source query had a 99th-percentile latency of 903 ms against 100 ms for the store. A strict guard exposed 41% of the encounters hit by faults that map a code to a neighboring state, but it raised false alarms on 9.9% of encounters when sources skipped statuses. No real data were used, the results depend on assumed parameters, and sub-second dashboard response remains a design target.

Keywords:

Enterprise Status Federation, State Machine Normalization, Healthcare Interoperability, Change Data Capture, Schema-Agnostic Mapping, Transition Validation, Clinical Workflow, Event-Driven Architecture.

Article History:

Received: 09.04.2022

Revised: 23.04.2022

Accepted: 02.05.2022

Published: 19.05.2022

1. Introduction

A hospital or health system rarely runs one application. Registration, clinical documentation, insurance verification, laboratory, pharmacy and billing are usually separate products, bought at different times and kept up by different teams. Each does its job, and each keeps its own interfaces, tables and status labels. No single screen then shows where a patient-related process stands. Hospital information systems often fall short of what staff need for their decisions [1], advanced use of electronic records is spread unevenly across hospitals [2], and the support a record gives to teamwork shapes how care teams coordinate [3]. Where a system does not fit the work, people improvise, and Friedman et al. catalogued the workarounds that primary care staff invent [4]. Fragmented status

Copyright by the Author(s). This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International (CC BY-NC-ND 4.0) License (https://creativecommons.org/licenses/by-sa/4.0/)

information has the same effect. A coordinator telephones another department, opens a second application, or asks a colleague, and the check is repeated for every patient and every process.

Interoperability standards address part of the problem. HL7 version 2 messages carry orders and results between systems [5], and FHIR defines resources and a REST interface for the same purpose [6], [7]. A standard fixes the form of the content, but two systems that both send valid messages can still use status values differently. One system may call a request "awaiting approval" where another calls the same stage "verification pending", and a status that reads "closed" can mean finished in one system and cancelled in another. An integration that translates such values wrongly shows staff a false state, and that kind of error is harder to spot than a gap.

The usual integration answers are point-to-point interfaces, which need a translator for every pair of systems, and a shared canonical model with one adapter per system [8]. A data warehouse with nightly loads gives a consistent view that is a day old. A federated query layer gives current data but ties the dashboard to the speed and availability of every source [9], [10]. This paper takes the canonical-model route and adds what a status view needs in particular: capture of changes as they commit, translation of status values that depends on context and keeps the source value, a check of the resulting progression, and a read store that the dashboard can query without touching the sources.

We call the pattern Unified State Machine Normalization. Extractors read committed changes from each source and emit events with a common envelope. A normalization engine translates the native status of each event into one of eight canonical states, using a versioned mapping registry. A state service applies the event to the state of the encounter if the transition is legal under a transition table, and quarantines the event otherwise. A read-only status store keeps the current state, the history and the provenance of every state. A dashboard queries the store and nothing else. Throughout, the source systems stay as they are: none is asked to adopt the canonical vocabulary, and no component of the federation writes to one.

The ingredients are known: log-based change capture [11], [12], canonical data models [8], explicit state machines [13], and read models that are kept apart from write models [14], [15]. Our contribution is the way they are combined for the semantics of status and the way the combination is examined. We make four contributions.

- We specify ten requirements, an event envelope, a mapping registry with qualifiers and versions, a transition table with a strict and a lenient mode, and rules for duplicate, late, missing and illegal events (Section III and Section IV).
- We define the objects of the pattern formally, give an algorithm that applies one event, and prove three propositions about artifact counts, replay and guard comparison (Section V).
- A synthetic study in four parts examines the number of mapping artifacts, strategies for extracting changes, the query path, and the detection of mapping faults by the transition guard, with code and seeds so that every number can be reproduced (Section VI and Section VII).
- A table of claims keeps evidence apart from expectation by recording whether each claim follows by construction, was observed in the synthetic study, or is a hypothesis for later measurement (Section VIII).

Related work is reviewed in Section II. Section III then states the setting and the requirements, Section IV the architecture and Section V the formal model. Section VI and Section VII describe the study and its results, Section VIII discusses them, and Section IX, Section X and Section XI give the limitations, an evaluation agenda and the conclusions.

2. Related Work

2.1. Use of Health Information Systems

Studies of hospital information systems describe uneven adoption and use [2], [16], limited fit with the decisions that staff must make [1], and the workarounds that follow when a system fits the work badly [3], [4]. Pickering et al. found that a clinician-designed viewer of record data was feasible and well received in an intensive care unit [17], which supports building a consolidated view around the user's task rather than around the screens of the source systems. Dashboards of patient data have been reviewed [18] and designed and tested with patients [19]. These studies establish the need for a coherent view but offer no model of status that spans systems.

2.2. Architecture and Integration in Health Information Systems

Reference architectures and models for health information systems include RICHE [20] and 3LGM^2 [21]. They help to structure complex environments, but they do not say how events from many applications become one operational state model. Experience re-

ports describe the building and deployment of medical information systems [22], [23] and the integration of systems in palliative care [24]. Reviews of the obstacles of data distribution service middleware [25] and of on-premise enterprise resource planning systems [26] bear on the connection of long-lived systems, although they do not address clinical status. FHIR and its application platform SMART define resources and interfaces for exchange [6], [27], and the adoption of health data standards has been slow and uneven [28]. Standards reduce the translation work for content, but the status values of local workflows still differ.

2.3. Data Integration and Schema Mapping

The theory of data integration separates mappings from a mediated schema to the sources into global-as-view and local-as-view forms [10], [29], and federated database systems were defined for autonomous and heterogeneous sources [9]. Schema matching surveys cover the automatic discovery of correspondences [30]. Extract–transform–load processes move data into a warehouse in batches [31], and data virtualization exposes sources through one query layer [32]. Enterprise integration patterns describe the canonical data model and the message translator [8], and Evans names the anticorruption layer that keeps the model of one context from being distorted by another [33]. The adapters in this paper are anticorruption layers for status. Two things set status apart: the mapping is a small table of meanings that people govern, rather than a schema correspondence to be discovered, and its errors have operational consequences. The mapping registry therefore has owners, versions and tests, and the transition guard serves as a runtime check on it.

2.4. Change Capture and Event Streaming

Log-based change data capture reads the transaction log of a database and publishes the changes in commit order. Databus at LinkedIn is an early description [11], Kafka provides the durable log from which consumers replay [12], and Kleppmann treats capture, event logs and derived views in book form [15]. Streams and tables are two views of the same data [34], which is why a status store can be rebuilt from a log. The outbox pattern writes an event to a table in the same transaction as the business change and relays it afterwards [14]. Exactly-once processing is available inside Kafka Streams [35], but at the boundary with other systems the practical guarantee is at-least-once delivery, and consumers must tolerate duplicates, a property that Helland calls idempotence [36]. Stream processing models keep event time apart from processing time [37], and complex event processing detects patterns over streams of events [38]. Studies of event-sourced systems and of data management in microservices report difficulties with schema evolution and with consistent derived views [39], [40].

2.5. State Models for Workflows

Statecharts are a visual formalism for hierarchical state machines [13]. Workflow research models processes with Petri nets [41] and catalogs patterns of control flow [42]. Process mining discovers such models from event logs [43] and has been applied in healthcare [44], and texts on business process management cover modeling and analysis [45]. These works describe how to model or discover the progression of one process. We take the transition table as given by the organization, and we use it to check the statuses that mapped systems report, which is a different use.

2.6. Consistency and Distribution

Asynchronous propagation trades immediate consistency for availability and decoupling [46]–[48], and transactions that span systems are avoided in large systems for the same reason [49], [50]. The order of events across machines rests on logical time rather than on clock readings [51]. A query that fans out to many servers waits for the slowest of them, which Dean and Barroso call the tail at scale [52]. We rely on these results for the design of the read path and quantify the effect under the assumptions of Section VI.

2.7. Gap

The works above cover adoption, workflow, architecture modeling, integration theory, change capture and dashboards. None of them, taken alone, combines asynchronous capture of source changes, context-dependent normalization of status values with provenance, and a transition guard that also detects mapping faults, and none quantifies the cost of the alternatives under one set of assumptions. This paper provides such a treatment on synthetic data and marks what a deployment must still show.

3. Status Vocabularies and Requirements

3.1. Heterogeneous Status Vocabularies

A process such as the verification of insurance for an admission, or the preparation of a prescription, passes through steps that are carried out in different departments, each with its own application. The applications record progress as a status value, usually a code with an optional second field, such as a reason or a payer class, that changes what the code means. We call the sequence of status-

es that a process records for one encounter a *status stream*. Table I shows invented examples from three hypothetical systems. They come from no real product and serve only to show the kinds of difference that a mapping must resolve.

Table 1. Invented Example Vocabularies of Three Source Systems and the Canonical State to Which Each Native Value Maps. A Reason (Qualifier) Changes the Mapping of Two of the Values

Canonical State	System A, Registration	System B, Insurance	System C, Pharmacy
Initiated	NEW	REQ	RECEIVED
Pending	WAIT	AWAIT (reason: documents)	QUEUED
Under Review	VERIF	VERIFYING; AWAIT (reason: payer)	PHARMACIST_REVIEW
Approved	CLEARED	APPROVED	RELEASED
Rejected	DENIED	DENIED	REJECTED
In Progress	ADMITTING	none	FILLING
Completed	CLOSED	CLOSED	DISPENSED
Exception	ERR; CLOSED (reason: cancelled)	ERR	ON_HOLD

The table contains four kinds of difference. Systems use different words for the same stage, as with VERIF, VERIFYING and PHARMACIST_REVIEW. The same word can also name different stages: AWAIT in the insurance system means a request that waits for documents or a request that waits for a payer, depending on the reason. Granularity differs, since the insurance system has no value that corresponds to In Progress. Finally, terminal values are overloaded: CLOSED is a completed process unless the reason says that it was cancelled. A mapping that looks only at the code gets two of the four wrong.

3.2. Requirements

The pattern is designed to meet the ten requirements in Table II. They are stated as properties of the whole federation, so that the architecture and the evaluation can both refer to them.

Table 2. Requirements of the Pattern and the Mechanism That Meets Each

ID	Requirement	Mechanism
R1	Source autonomy: sources keep their schemas, labels and release cycles, and no source is changed	Read-only extractors; no write path to a source
R2	Asynchronous propagation: a read never waits for a source	Durable event log; separate status store
R3	Complete capture: every committed status change is captured, or its loss is detected	Extraction technique chosen per source; heartbeats; gap rule E4
R4	Context-dependent normalization: the mapping may depend on source, code, qualifiers and time, and an unmapped value is never guessed	Versioned mapping registry; quaran-tine of unmapped events (E6)
R5	Provenance: every canonical state links to its source event, its native val-ue and its mapping version	Envelope keeps the native value; the store records the version
R6	Progression validation: transitions are checked, and illegal ones are visi-ble	Transition table with strict and lenient modes; quarantine (E5)
R7	Idempotent, order-tolerant processing: duplicates, late events and gaps are handled by rule	Rules E1 to E4; sequence numbers per source
R8	Read isolation: dashboard queries do not touch source systems	Pre-normalized, read-only status store
R9	Extensibility: a new source needs one adapter and one set of rules	Adapter and registry per source (Prop-osition 1)
R10	Security and auditability: identifiers are protected and every change is traceable	Role-based access, encryption, audit log

The requirements correspond to dimensions that data-quality frameworks for health records name, namely completeness, plausibility and currency [53], [54]. R3 concerns completeness of capture, R6 is a plausibility check on the stream of statuses, and the freshness display of Section IV makes currency visible, since the federation is asynchronous and never as current as a source.

3.3. Principles

Five principles guide the design, and Table III relates each to the requirements it serves and to the mechanism that realizes it. Under *system autonomy*, sources keep their schemas and business rules. *Asynchronous propagation* means that no component waits on a source in order to answer a read. *Schema agnosticism* requires the federation to interpret status through a mapping instead of through assumptions about source tables. *State consistency* asks that every status stream have one canonical state at any time and that illegal progressions be surfaced. *Presentation independence*, finally, means that the dashboard reads a store that the federation owns.

Table 3. The Five Design Principles, the Requirements they Serve and the Mechanisms that Realize them

Principle	Requirements	Realized by
System autonomy	R1, R9	Read-only extractors, one adapter per source
Asynchronous propagation	R2, R3	Event log between capture and use
Schema agnosticism	R4, R5	Mapping registry with qualifiers and versions
State consistency	R6, R7	Transition table, rules for duplicate and late events
Presentation independence	R8	A store that the federation owns

3.4. Scope

The federation is a read-only information layer. It never writes to a source, and it neither replaces transactional systems or clinical documentation nor infers clinical meaning. Each status stream is keyed by encounter and process. We assume that every source can supply an encounter identifier, or a patient identifier that an enterprise index resolves to one. Record linkage is a separate problem with its own literature [55], and a linkage error would show as a status attached to the wrong encounter. The study in this paper does not examine it.

4. Architecture

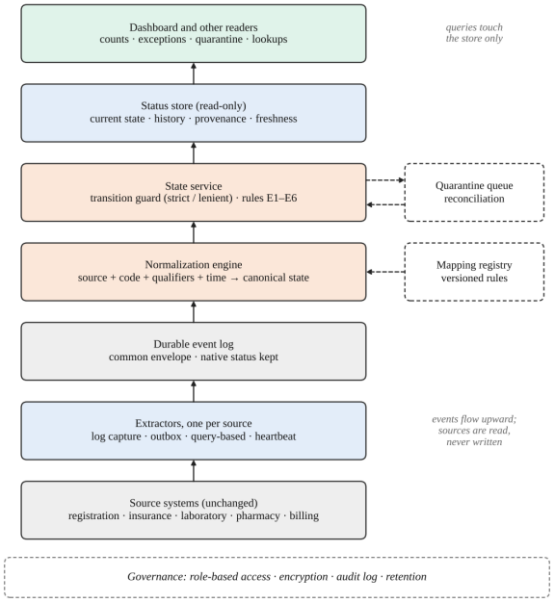


Figure 1. Layers of the Federation. Solid Arrows Carry Events, and Dashed Arrows Carry Configuration and Reconciliation. Sources are Read but Never Written

4.1. Overview

Fig. 1 shows the layers. Source systems keep their own databases. One extractor per source reads committed changes and wraps each in an event. The events go to a durable log, which absorbs bursts and lets consumers fall behind and catch up. The normalization engine reads the log, translates native statuses with the mapping registry, and passes canonical events to the state service. The state service applies each event to the status stream it belongs to, under the transition table, and writes the result to the status store. The dashboard and every other reader query the store. Governance, meaning access control, audit and retention, applies to all layers. Two flows are off the main path: the registry supplies mappings to the engine, and a quarantine queue returns events to the state service after a person has reconciled them.

4.2. Extraction from Sources

The extractor is the only component that touches a source, and the technique it uses decides which changes can be lost. There are three families of technique, and one deployment may use different ones for different sources (Table IV).

Table 4. Extraction Techniques, What Each Needs from the Source, and What Each Can Lose

Technique	Needs from the source	Can lose	Latency
Timestamp watermark	Read access; a timestamp column	Late commits, unchanged timestamps, deletions, overwritten versions	Poll interval
Watermark with overlap	The same	Unchanged timestamps, deletions, overwritten versions	Poll interval
Snapshot comparison	Read access	Versions overwritten within one interval	Poll interval
Log-based capture	Log access and privileges	Nothing committed, if the log is retained	Short
Outbox, log or polled	A change in the source	Writes that bypass the outbox	As the reader

Log-based capture reads the transaction log of the source database. It needs no change to the application or the schema, but it needs privileges on the log, support from the vendor, and a log that is retained long enough for the extractor to resume after an outage. It delivers every committed change in commit order, including changes that do not touch a timestamp column, and the log position serves as a sequence number [11]. Outbox capture adds a table that the application, or a trigger, writes in the same transaction as the business change, and a relay then publishes its rows [14]. This needs a change in the source, which the owner of the source may not accept, and it misses every write that bypasses the outbox, such as a batch job or an administrative script. Query-based capture needs only read access. A timestamp watermark asks for the rows whose change time is later than the last row seen. This loses a change that commits late with an earlier timestamp, because the watermark has passed it, and it cannot see an update that leaves the timestamp unchanged or a row that was deleted. An overlap window repeats the read of everything within a margin of the watermark and discards duplicates by key and version, which recovers late commits up to the size of the margin. Snapshot comparison reads the whole table and compares it with the previous read. It sees updates that leave the timestamp unchanged and deletions, but it collapses several changes within one interval into one, and its read cost grows with the size of the table rather than with the number of changes.

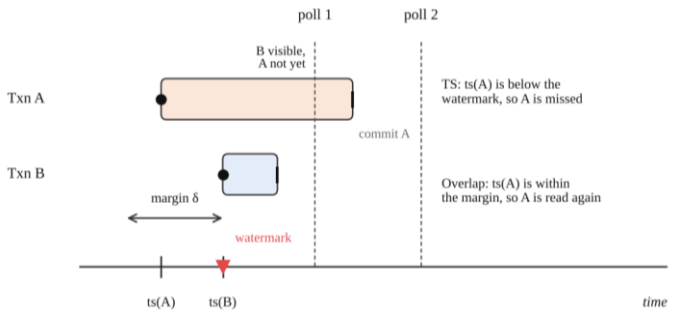


Figure 2. Shows How a Watermark Loses a Change That Commits Late

Fig. 2. A watermark poll loses a late commit. Transaction A starts first and commits after the poll that has already read transaction B, so the watermark passes the timestamp of A. An overlap window re-reads rows within a margin of the watermark and recovers A.

Every extractor also emits a heartbeat with its position, even when no change occurs. The federation can then tell a quiet source from a broken extractor, and the dashboard can show the age of the latest news from each source.

4.3. Event Envelope

Every extractor emits the same envelope (Table V). The envelope carries the native status exactly as the source stored it, with its qualifiers, and no later stage replaces it. Time stamps serve display and the measurement of lag. Ordering within one source uses the source sequence number, and no comparison of clocks across sources is assumed.

Table 5. Fields of the Event Envelope.

Field	Content
event_id	Unique identifier, formed from the source and the log position
source_id	The source system and its extractor
stream key	Encounter identifier and process
source_seq	Sequence number or log position, increasing within the source
commit_time	Time of the commit by the clock of the source
capture_time	Time of capture by the clock of the extractor
operation	Insert, update or delete
native_code	The status exactly as the source stored it
qualifiers	Named fields that can change the meaning of the code
txn_id	Transaction identifier of the source, where available
schema_version	Version of the layout of the source row
raw_ref	Pointer to the retained raw row, subject to retention policy

4.4. Mapping Registry and Normalization

The normalization engine computes the canonical state of an event from the source, the native code, the qualifiers and the commit time. The registry holds one record per rule (Table VI). A rule names a source and a native code and may add a predicate on qualifiers and an interval of validity. Of all rules that are valid at the commit time of the event and match it, the one with the most specific qualifier predicate wins. Two matching rules of equal specificity are a registry error that is rejected when the rule is approved, so that the result is deterministic. An event that matches no rule is never guessed; it is quarantined with the reason *unmapped*, and the number of unmapped events per source is a monitored quantity.

Table 6. Fields of a Mapping Rule in the Registry

Field	Content
rule_id, version	Identifier and version of the rule
source_id, native_code	The native status that the rule matches
qualifier predicate	Optional condition on qualifier fields, for example reason = cancelled
canonical_state	The state to which the rule maps
valid_from, valid_to	Interval of commit times in which the rule applies
steward	The person who owns the meaning of the status
approval	Proposer, approver and approval time; the two persons differ
test evidence	Reference to the replay test that supported the rule
note	Free text, for example the source documentation consulted

Rules follow a short life cycle. A steward or an integration analyst proposes a rule. A test replays recorded events of the source through the proposed rule and shows the canonical sequences that result and the guard outcome. The owner of the source process approves the rule, and the proposer cannot be the approver. The rule becomes active from an effective time, and monitoring of unmapped events and of guard flags per native code follows. When a rule turns out to be wrong, the corrected rule gets a new version with the effective time of the original error, and the engine normalizes the retained events of the affected codes again. The store records the mapping version with every state, so a state computed under the old version is distinguishable, and the audit trail shows both. Fig. 3 illustrates the resolution of one qualified status.

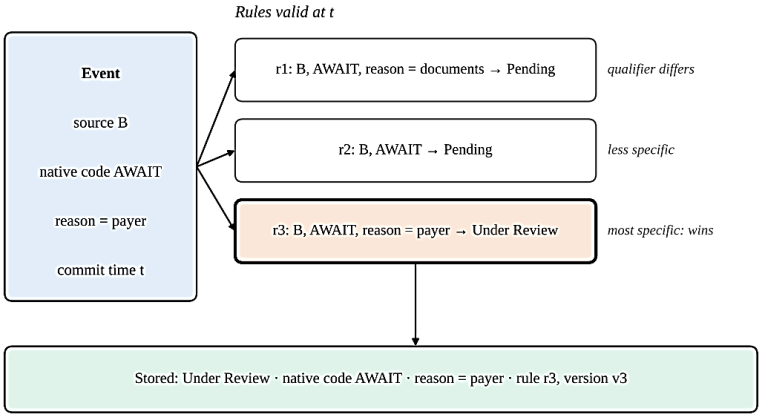


Figure 3. Resolution of a Native Status. The Most Specific Matching Rule That Is Valid at the Commit Time of the Event Wins, and the Event Keeps Its Native Code and the Version of the Rule

4.5. Unified State Machine

The eight canonical states are listed in Table VII. They form an operational vocabulary and not a clinical one. Where an export is needed they can be projected onto the status codes of the FHIR Task resource [7], with a loss of detail.

Table 7. Canonical States

State	Code	Meaning
Initiated	INI	A request exists and no one has acted on it
Pending	PEN	The process waits for an action or input from outside, such as documents, a payer or a patient
Under Review	REV	A person or a system is evaluating the request
Approved	APP	A decision to proceed has been made
Rejected	REJ	A decision not to proceed has been made; the request can be resubmitted
In Progress	PRG	Work toward completion is under way
Completed	CMP	The process has ended; no transition leaves this state
Exception	EXC	The process cannot continue normally and needs attention: an error, a hold or an escalation

The transition table lists the moves that a status stream may make (Table VIII and Fig. 4). Completed is terminal. Exception can be entered from any non-terminal state and leaves toward the stages at which work can resume. The guard has two modes. The strict mode passes only the listed transitions. The lenient mode also passes forward moves along the main path, from Initiated through Pending, Under Review, Approved and In Progress to Completed, so that skipped statuses are tolerated, and it passes any move from a non-terminal state into Rejected or Exception and any move out of those two states except a return to Initiated. A source that reports every step of its process suits the strict mode, and a source whose extraction can collapse steps suits the lenient mode. The choice is made per source, and Section VII measures what it costs in each direction.

Table 8. Moves Accepted By The Strict Guard, and the Moves that the Lenient Guard Adds. A Repeated State Is Not A Transition. The Strict Table Has 24 Moves and the Lenient Mode Adds 15. State Codes Are Those of Table 7

From	Strict Table	Added By The Lenient Mode
INI	PEN, PRG, EXC	REV, APP, REJ, CMP
PEN	REV, APP, REJ, EXC	PRG, CMP
REV	PEN, APP, REJ, EXC	PRG, CMP
APP	PRG, CMP, EXC	REJ
REJ	PEN, CMP, EXC	REV, APP, PRG
PRG	PEN, CMP, EXC	REJ
CMP	none	none
EXC	PEN, REV, PRG, CMP	APP, REJ

Events that cannot be applied cleanly are handled by six rules, which are stated here and are not part of the experiments.

- *E1 Duplicate.* An event whose source, stream and sequence number have already been processed is dropped and counted.
- *E2 Same state.* An event that maps to the current state updates the time stamps and the provenance and records no transition.
- *E3 Late event.* An event with a sequence number below the highest one processed for the same source and stream is added to the history with a mark and does not change the current state.
- *E4 Gap.* If a source supplies contiguous sequence numbers and one is missing, the extractor re-reads the entity, and the stream is marked possibly stale until the re-read arrives.
- *E5 Illegal transition.* The event goes to the quarantine queue with the reason, the stream is flagged, and the current state stays at the last accepted one. The dashboard shows that state with a marker.
- *E6 Unmapped.* The event goes to the quarantine queue with the reason *unmapped*.

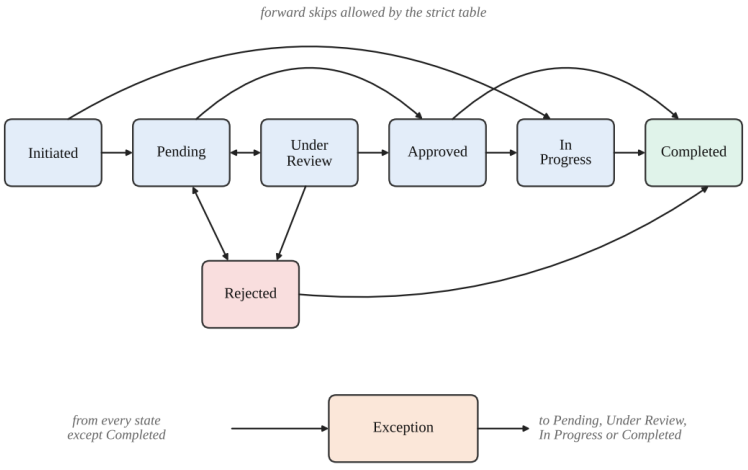


Figure 4. Canonical States and the Moves of the Strict Guard. The Arcs above the Top Row are The Forward Skips That the Strict Table Allows. Moves into Exception from Every State but Completed, and Out of It, are Drawn as Labelled Stubs, and The Return From in Progress to Pending is Not Drawn. Table 8 Lists Every Move, Including Those That Only the Lenient Guard Accepts

Staff resolve a quarantine item by accepting the transition as legal for that stream, by correcting the mapping, or by recording a defect of the source, and the resolved event then re-enters the state service. A source can report the Exception state itself. That is a status of the process, whereas quarantine is how the federation handles statuses that it cannot apply, and the two are kept apart.

4.6. Status Store and Dashboard

The status store holds one current row per stream, with the canonical state, the previous state, the time since which the state holds, the source, the native code and qualifiers, the mapping version, the event identifier and a flag. An append-only history table keeps every applied transition. The store is derived data: it can be rebuilt from the log (Section V). Indexes serve the three reads that the dashboard makes, namely counts by state, lists of exceptions and quarantined items, and lookups by encounter. The dashboard also shows the freshness of every source, which is the age of its latest event or heartbeat. A reader who sees a stream in Pending can therefore also see how old the news from its source is.

Fig. 5 contrasts the two ways to serve such a dashboard. A federated path translates at read time: the dashboard queries every source, translates with the registry and merges. The data are as current as the sources, but the response time depends on the slowest source, the availability on all sources at once, and the load on the transactional systems on the number of dashboard users. The pre-normalized path moves translation to write time. It needs a store and a synchronization step, and it adds a delay between a commit in a source and its appearance on the dashboard. Section VII quantifies both paths.

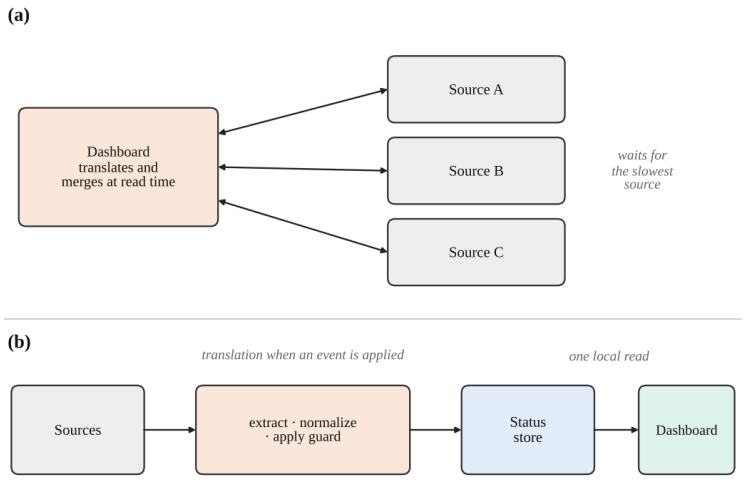


Figure 5. Two Query Paths. In The Federated Path (A) the Dashboard Waits For Every Source and Translates At Read Time. In The Pre-Normalized Path (B) Translation Happens When an Event Is Applied, and the Dashboard Reads One Store

4.7. Security, Governance and Extensibility

The store holds patient identifiers, so it is a repository of protected health information and is governed as one (Table IX). The pattern supports the controls in the table, but compliance remains a property of each deployment. Role-based access follows long-standing models [56], [57]: a registrar sees registration and insurance streams, a pharmacist sees pharmacy streams, and a supervisor sees counts and exceptions across processes. Extractor accounts hold read-only privileges and no more than they need [58]. Requirements for such controls appear in the HIPAA Security and Privacy Rules [59], [60], control catalogs give implementation detail [61], and authentication between services follows zero-trust principles [62].

A new source is added in six steps. The integration team inventories the statuses and qualifiers of the source, writes the registry rules, chooses an extraction technique, replays recorded events through the proposed rules, runs the source in a shadow period in which flags are reviewed and nothing is shown, and then activates it. The artifacts that this requires are one adapter and one set of rules, and the existing adapters, rules and components stay as they are (Proposition 1).

Table 9. Controls for the Status Store and the Federation, and the Guidance behind Each

Area	Measure	Basis
Authentication	Strong authentication of users; mutual authentication between services	Zero-trust principles
Authorization	Role-based views by process and responsibility; read-only extractor ac-counts	Role-based access con-trol
Encryption	Encryption in transit and at rest for the log, the store and retained raw rows	HIPAA Security Rule
Audit	Log of patient-level reads and registry changes, in tamper-evident stor-age	HIPAA Security Rule
Integrity	Sequence numbers and idempotent application; approver differs from proposer	Separation of duties
Minimum neces-sary	Views show the fewest identifiers that the task needs; supervisors see counts	HIPAA Privacy Rule
Retention	Periods for events, raw rows and history set by policy; scheduled deletion	Organizational policy
Recovery	Replay rebuilds the store; extractors resume from their last position	Contingency planning

5. Formal Model

5.1. Definitions

Let $S = \{1, \ldots, n\}$ be the source systems. Source s has a set Σ_s of native status codes and a set Q_s of qualifier values, which includes the value none. Let C be the set of canonical states, with $|C| = k = 8$, and let $\perp \notin C$ mark an unmapped event. A version v of the registry gives each source a mapping

$$\mu_s^v : \Sigma_s \times Q_s \to C \cup \{\perp\}. \qquad (1)$$

Selection of the most specific matching rule makes μ a function, because ties are rejected at approval. An event is $e = (s, x, \sigma, q, a, t)$, where x is the key of the status stream, σ the native code, q the qualifiers, a the sequence number that the source assigns, and t the commit time. The registry history is a piecewise constant function $v(t)$, and corrections enter it with backdated effective times. The canonical state of an event is

$$c(e) = \mu_s^{v(t)}(\sigma, q). \qquad (2)$$

The transition relations are $T_{\text{strict}} \subseteq C \times C$ and $T_{\text{lenient}} \supseteq T_{\text{strict}}$. A guard g accepts a move from c to c' if $c = c'$ or $(c, c') \in T_g$, which is rule E2 for repeated states. Algorithm 1 applies one event to its stream. The set *seen* holds the triples (s, x, a) of events that have been processed in any way, and $a_x[s]$ is the highest sequence number that has been processed for stream x from source s. An event that a person has reconciled is applied directly, as in step 8, with the resolution recorded.

Algorithm 1. Application of one event to its status stream

1: $c \leftarrow c(e)$ from the registry version in force at t; read the state c_x of stream x and the number $a' = a_x[s]$
2: **if** $(s, x, a) \in$ *seen* **then** count a duplicate and **return** (E1); otherwise add it to *seen*
3: $a_x[s] \leftarrow \max(a', a)$
4: **if** $a > a' + 1$ **and** the source numbers its events without gaps **then** request a re-read of x and mark x possibly stale (E4)
5: **if** $c = \perp$ **then** quarantine e as *unmapped* and **return** (E6)
6: **if** $a < a'$ **then** add e to the history, marked late, and **return** (E3)
7: **if** $c = c_x$ **then** update times and provenance and **return** (E2)
8: **if** c_x is not yet defined **or** $(c_x, c) \in T_g$ **then** write the new state, the previous state, the mapping version and e to the store and the history
9: **else** quarantine e as *illegal transition* and flag x (E5)

The algorithm does constant work per event apart from the lookup of the rule and the read and write of one stream.

5.2. Mapping Artifacts

A point-to-point network of translators needs one translator for each ordered pair of systems that must see each other's statuses. A hub needs one adapter for each system. With n systems,

$$N_{\mathrm{p2p}}(n) = n(n-1), \qquad N_{\mathrm{hub}}(n) = n. \tag{3}$$

$$E_{\mathrm{p2p}}(n) = 2(n-1), \qquad E_{\mathrm{hub}} = 1 \tag{4}$$

A change in the vocabulary of one system touches artifacts, since the system takes part in $n - 1$ translators as source and in $n - 1$ as target, and the hub has a single mapping for it. *Proposition 1 (artifact counts).* For n systems with m native (code, qualifier) entries each, point-to-point translation needs $n(n - 1)$ translators and $n(n - 1)m$ entries, and the hub needs n adapters and nm entries. A vocabulary change touches $2(n - 1)$ translators or one mapping, and the addition of a system to n existing ones needs $2n$ new translators or one adapter. *Proof.* Each ordered pair of distinct systems has its own translator, and each translator holds one entry for each of the m entries of its source. A new system adds n translators from it and n translators to it. In the hub, each system has one adapter with one mapping of m entries.

The count is the worst case in which every system needs the statuses of every other. If only a fraction ρ of the ordered pairs is needed, the hub needs fewer artifacts when

$$\rho\, n\,(n-1) > n \Leftrightarrow n > 1 + 1/\rho. \tag{5}$$

For ρ = 1/4 this holds from six systems onward, and at five systems the two counts are equal.

5.3. Replay

Proposition 2 (replay). Let the log hold events $e_1, \ldots, e_N$ in a fixed order, and let the registry history and the guard mode be fixed. Applying the events to an empty store with Algorithm 1 gives a store that depends only on the log, the registry history and the guard mode. Presenting any event a second time, at any later point, leaves the store unchanged. *Proof.* Every step of the algorithm depends on the event, on the registry version at its commit time, and on the state that the earlier steps produced. The result is therefore a function of the log and the registry history. A repeated event has the triple of an event in *seen*, which step 2 drops, and *seen* is not part of the store that readers see.

The proposition justifies two operations that the architecture relies on. The store can be rebuilt from the log after a loss, and the events of a corrected rule can be normalized again, because both are replays under a registry history.

5.4. Guard Comparison

Let a mapped sequence be $c_1, \ldots, c_r$. A guard flags it when some move is not accepted:

$$F_g(c_1, \ldots, c_r) = \exists\, i < r : c_i \neq c_{i+1} \wedge (c_i, c_{i+1}) \notin T_g. \tag{6}$$

An encounter is *affected* by a mapping fault when its mapped sequence differs from its true canonical sequence. With π the share of affected encounters, the quantities of the study are

$$\mathrm{det}_g = \Pr\big[F_g \mid \text{affected}\big], \quad \mathrm{fa}_g = \Pr\big[F_g \mid \text{unaffected}\big], \quad \mathrm{prec}_g = \frac{\pi\, \mathrm{det}_g}{\pi\, \mathrm{det}_g + (1-\pi)\, \mathrm{fa}_g}. \tag{7}$$

Detection and false alarms are the recall and the false positive rate of the guard as a classifier of affected encounters, and precision follows from them and from π [63].

Proposition 3 (guard comparison). For every mapped sequence, F_{lenient} implies F_{strict}, whether the sequence is only classified or the guard also quarantines moves and compares later moves with the last accepted state. The detection rate and the false-alarm rate of the lenient guard are therefore no higher than those of the strict guard on the same encounters. *Proof.* The lenient mode adds moves to the strict table and removes none, so $T_{\mathrm{strict}} \subseteq T_{\mathrm{lenient}}$. In the classifying case, a move that the lenient guard rejects is rejected by the strict guard. In the quarantining case, let i and j be the first positions at which the strict and the lenient guard reject a move. Before $\min(i, j)$ neither guard has quarantined anything, so both compare the same pairs, and the inclusion of the tables gives $i \leq j$ whenever j exists. Rates are expectations of the flags over the same encounters, and the inequality between flags carries over.

A guard sees only pairs of consecutive states. Two canonical states with the same predecessors and the same successors cannot be told apart by it, so a mapping fault that swaps them is invisible. No two states of Table VIII have both the same predecessors and the same successors. Approved and Rejected have the same predecessors, Pending and Under Review, and different successors, so a swap between them shows only through what follows the swapped state. Rejected and In Progress have the same successors and different predecessors. The check is mechanical, and a designer of a transition table can run it for every pair of states.

5.5. Query Path

Let the latencies $L_1, \ldots, L_n$ of n sources be independent and identically distributed, let q_p be the p-quantile of one of them, m the time to merge the answers, and a the availability of one source. A federated query waits for all sources and needs all of them:

$$L_n = \max(L_1, \ldots, L_n) + m, \qquad \Pr[L_n - m > q_p] = 1 - p^n, \qquad A_n = a^n. \qquad (8)$$

The probability of exceeding the 99th percentile of a single source is $1 - 0.99^n$, so it is 11% for twelve sources, as the tail-at-scale argument of [52] predicts. The availability falls with n in the same way. A pre-normalized store answers from one place, and its latency and availability do not depend on n.

6. Evaluation Method

The study has four parts, and each addresses one design question: how many artifacts a hub saves (Part A), which extraction technique loses which changes (Part B), what the read path costs (Part C), and how much of the mapping error the transition guard exposes (Part D). All parts use synthetic data generated by the code that accompanies the paper. No clinical, order or patient record is used. The parameters are illustrative: they are chosen to be plausible and to make the mechanisms visible, and they are not calibrated to any installation. The results therefore show how the mechanisms behave and what they depend on; they are not predictions for any particular hospital.

6.1. Part A: Mapping Artifacts

The counts of Section V are evaluated for n from 2 to 16 systems with $m = 12$ native values per system. The part is analytic and has no random element.

6.2. Part B: Change Extraction

The generator produces the committed status changes of one day for 5,000 records. Table X lists the parameters. Each record passes through several versions. A version has a row timestamp that is set when its transaction starts, and it becomes visible when the transaction commits, which is how a long transaction can commit behind a watermark that other rows have already advanced. Most commits follow the start within tens of milliseconds, and a small share are long transactions. Some updates leave the timestamp column unchanged, as happens when an application updates a status column without touching the audit column, and some updates do not write the outbox, as happens when a batch job or a script changes the table directly. The extraction pipeline has outages, during which polls are skipped and log reads wait until the pipeline is back.

Table 10. Parameters of the Extraction Study

Parameter	Value
Records	5,000, one status row each
Horizon	86,400 s; later updates are dropped
Versions per record	3 + Poisson(2.5); mean 27,016.5 updates per run
First version	Uniform over the first 85% of the day
Gap between versions	Lognormal, median 1,200 s, σ = 1.2
Commit lag, long transaction	Probability 0.5%; lognormal, median 20 s, σ = 1.0
Commit lag, otherwise	Lognormal, median 0.03 s, σ = 0.6
Timestamp unchanged	2% of updates after the first
Bypass of the outbox	3% of versions
Poll interval I	60 s
Overlap margin δ	300 s
Pipeline outages	2 per day, mean duration 300 s

Log capture latency	Lognormal, median 0.5 s, σ = 0.5
Seeds	8 in the default scenario, 4 in each sweep

Six strategies read the same stream of updates, so that differences between strategies are paired by seed [64].

- *TS.* Every *I* seconds the extractor reads the rows whose timestamp is later than a watermark, which is the highest timestamp read so far.
- *TS+overlap.* The same, but each poll also re-reads the rows within δ of the watermark and discards duplicates by record and version.
- *Snapshot.* Every *I* seconds the extractor reads the whole table and compares it with the previous read. Changes within one interval collapse into the last version.
- *Log.* Every committed version is delivered after a short capture latency, or after the end of an outage that was in progress at the commit.
- *Outbox log.* The same, for the versions that wrote the outbox.
- *Outbox poll.* The outbox rows are polled with a timestamp watermark, as for TS.

Four measures describe a strategy. *Capture* is the share of committed versions that were delivered at least once. *Stale records* is the share of records whose last delivered version is not their final version at the end of the run, which is what a reader of the status store would see as a wrong state. *Latency* is the time from commit to delivery, reported as the median and the 95th and 99th percentiles over delivered versions. *Read amplification* is the number of rows that the extractor read for each delivered version. Each run continues for an hour and three poll intervals after the day ends, so that the last polls can finish.

The default scenario uses 8 seeds. Four one-factor sweeps, each with the first 4 seeds, vary the poll interval *I* over 5, 15, 60 and 300 s, the share of long transactions over 0, 0.5%, 2% and 5%, the share of updates that leave the timestamp unchanged over 0, 2% and 5%, and the share of updates that bypass the outbox over 0, 3% and 10%.

6.3. Part C: Query Path

The first measurement is a micro-benchmark in SQLite, in one process on one machine, with a warm cache. Each of *n* sources, with *n* = 2, 5 and 8, has a table of 100,000 rows with an identifier, a native code, a qualifier, a timestamp and a payload, with an index on code, qualifier and timestamp, and a mapping table translates, for each source, the 36 combinations of 12 native codes and 3 qualifier values into the eight canonical states. The federated variant translates at query time: it joins each source table with the mapping table, runs one query per source and merges the results in the client. The pre-normalized variant has one table with the source, the identifier, the canonical state and the timestamp, built once by a backfill step and indexed on state and time and on the identifier. Three queries are timed (Table XI): counts by canonical state (Q1), the first 50 exception rows in timestamp order (Q2), and eight lookups by identifier (Q3), where the federated variant asks each source in turn. After one warm-up run, Q1 is repeated 9 times, Q2 15 times and Q3 100 times, and the median and the 95th percentile are reported. This measures the cost of translating at query time inside one database engine. It has no network, no concurrent writers and no contention, so it does not measure a federated query across real systems.

Table 11. Parameters of the Query-Path Measurements

Parameter	Value
SQLite sources *n*	2, 5 and 8
Rows per source	100,000
Mapping entries	12 native codes × 3 qualifier values = 36 per source
Q1	Count of rows by canonical state
Q2	First 50 rows in state Exception, in timestamp order
Q3	Eight lookups by identifier
Repetitions	Q1 9, Q2 15, Q3 100, after one warm-up run
Source latency	Lognormal, median 80 ms, ratio of 99th percentile to median 6
Store latency	Lognormal, median 25 ms, ratio 4

Merge time	5 ms
Availability	99.5% per source, 99.95% for the store
Fan-out model	n = 1, 2, 3, 5, 8, 12; 60,000 draws per n

The second measurement is a Monte Carlo model of fan-out. The latency of one source is lognormal with a median of 80 ms and a ratio of 99th percentile to median of 6. The pre-normalized store has a median of 25 ms and a ratio of 4. A federated query takes the maximum over n independent source latencies and adds 5 ms for merging, for n = 1, 2, 3, 5, 8 and 12, with 60,000 draws for each n. The availability of a source is 99.5% and that of the store is 99.95%. These values are assumptions, and the results follow from them; the model is meant to show how the effect grows with n, and the closed forms of Section V are checked against the draws.

6.4. Part D: Transition Guard

A source-to-canonical mapping is built for three source systems, each with one to three native codes for every canonical state. Encounters are random walks over the canonical states that start at Initiated and use the probabilities of Table XII. A walk ends at Completed or, in about 0.1% of the walks, when it reaches 20 statuses. Each status that an encounter passes through is reported with a native code drawn from the codes of that state in any of the three systems. Interior statuses are left unreported with a probability of 5%, which models changes that no extractor observed. Faults are injected into the mapping. A fault changes the canonical state of one native code to a different state: in the *adjacent* regime to a neighbor of the true state, as listed in Table XII, and in the *random* regime to any other state. One, two or four codes are faulty in a trial, and the codes are chosen at random. A scenario has 300 trials, each with a new vocabulary and a new fault set, and 1,000 encounters per trial.

Table 12. Transition Probabilities of the Encounter Walks in the Guard Study and the Neighbor States Used For Adjacent Faults

From	Next State, Probability	Adjacent Substitutes
INI	PEN 0.80, PRG 0.12, EXC 0.08	PEN
PEN	REV 0.55, APP 0.30, REJ 0.08, EXC 0.07	INI, REV
REV	APP 0.70, REJ 0.15, PEN 0.08, EXC 0.07	PEN, APP
APP	PRG 0.75, CMP 0.17, EXC 0.08	REV, PRG
REJ	PEN 0.60, CMP 0.30, EXC 0.10	EXC, PEN
PRG	CMP 0.88, PEN 0.04, EXC 0.08	APP, CMP
CMP	none	PRG
EXC	PEN 0.35, REV 0.20, PRG 0.30, CMP 0.15	REJ, PEN

The guard classifies an encounter as flagged when any consecutive pair of its mapped states is outside the accepted relation, in the sense of Section V, for the strict table and for the lenient mode. The measures are those of Section V: the share of affected encounters, detection, false alarms and precision. One more measure concerns what a person would find: *surfaced codes* is the share of faulty codes that appear in at least one flagged encounter, among the faulty codes that appear in any affected encounter. A code that is never surfaced would go unnoticed even if someone reviewed every flagged encounter.

6.5. Statistics

For Part B, means are given with 95% intervals based on the t distribution over seeds (t = 2.36 for eight seeds and 3.18 for four) [64]. The percentiles of latency are computed over all delivered versions of a run and then averaged over seeds, and the 99th percentile under outages is noisy, because a few versions wait for the end of an outage. Rates in Part D are pooled over trials, and their 95% intervals come from the spread across trials, since the encounters of a trial share one set of faulty codes and would give an interval that is too narrow if treated as independent. The micro-benchmark of Part C is one run for each configuration, and its repetitions are not independent of one another, so the percentiles describe the spread of repeated execution on one machine rather than a confidence interval.

7. Results

7.1. Mapping Artifacts

Table XIII gives the counts of Proposition 1 for $m = 12$ native values per system. The two approaches need the same number of artifacts at two systems. The ratio of translators to adapters is $n - 1$, so it grows linearly with the number of systems, and the difference between the two counts grows with its square. At eight systems point-to-point translation needs 56 translators with 672 entries, against 8 adapters with 96 entries for the hub. A change of one vocabulary touches 14 translators at eight systems and 30 at sixteen, and one mapping in the hub. Adding a ninth system to eight needs 16 translators or one adapter. The figures follow from the formulas and are not evidence about any organization, but they show how quickly pairwise translation grows, and Fig. 6 plots them.

Table 13. Artifacts of Point-To-Point Translation and of The Hub With 12 Native Values Per System. A Vocabulary Change Touches One Mapping In the Hub

N	Translators, Pairwise	Adapters, Hub	Ratio	Entries, Pairwise	Entries, Hub	Edits Per Change, Pairwise
2	2	2	1	24	24	2
4	12	4	3	144	48	6
8	56	8	7	672	96	14
12	132	12	11	1,584	144	22
16	240	16	15	2,880	192	30

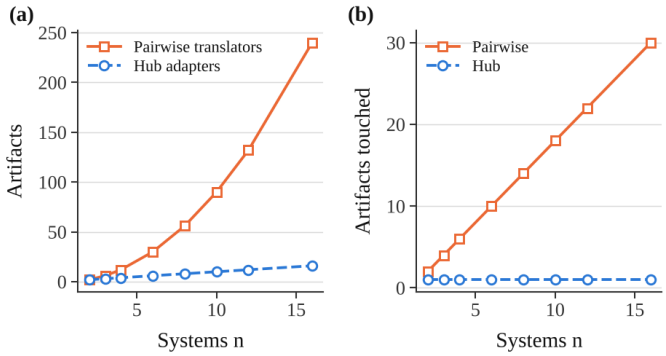


Figure 6. Artifacts Needed By Point-To-Point Translation and By a Canonical Hub As the Number of Systems Grows (A), and the Number of Artifacts Touched By a Change to One Vocabulary (B)

7.2. Change Extraction

Table XIV gives the results of the default scenario, and Fig. 7 shows them.

Table 14. Extraction Strategies in the Default Scenario. Capture and Stale Records Are Means Over Seeds With 95% Intervals. Latency Is the Time from Commit to Delivery

Strategy	Capture (%)	Stale Records (%)	Median (S)	95th (S)	99th (S)	Rows Read Per Version
Timestamp polling	97.97 ± 0.09	2.25 ± 0.18	30.0	57.3	59.7 ± 0.2	1.0
Polling with overlap	98.37 ± 0.07	1.82 ± 0.14	30.0	57.3	59.7 ± 0.2	5.8
Snapshot comparison	99.81 ± 0.06	0.00 ± 0.00	30.0	57.3	59.7 ± 0.2	164.5
Log-based capture	100.00 ± 0.00	0.00 ± 0.00	0.5	1.2	10.5 ± 15.8	1.0
Outbox, log reader	96.98 ± 0.08	3.22 ± 0.18	0.5	1.2	11.2 ± 16.7	1.0
Outbox, polled	96.79 ± 0.08	3.40 ± 0.19	30.0	57.3	68.2 ± 14.6	1.0

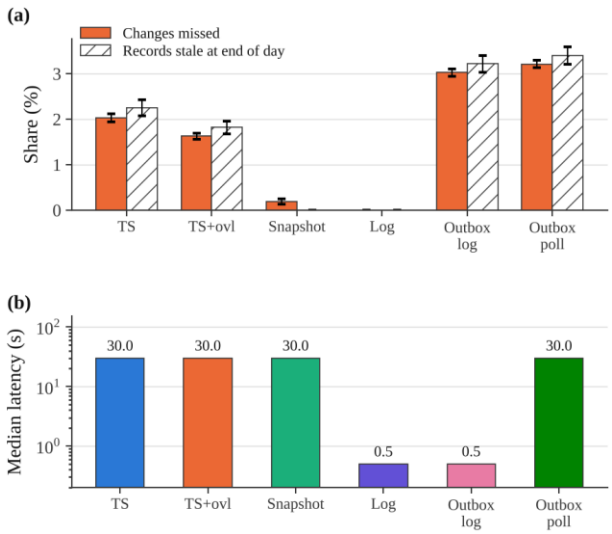


Figure 7. Extraction Strategies in the Default Scenario. (A) Share of Changes That Were Not Captured and Share of Records Left Stale at the End of the Day. (B) Median Latency from Commit to Delivery, On a Logarithmic Scale. Bars Show Means Over Seeds, and the Error Bars in (A) Are 95% Intervals

Log-based capture delivered every committed change (100.0%) and left no record stale. Its median latency was 0.5 s and its 95th percentile 1.2 s. The 99th percentile was 10.5 s with an interval of ±15.8 s, because a few versions committed during outages and waited for the pipeline to return.

Timestamp polling missed 2.0% of the changes (±0.1%) and left 2.2% of the records stale at the end of the day. Its median latency of 30.0 s is half the poll interval, as expected for a uniform wait, and 60 times that of the log. The misses have three causes: updates that leave the timestamp unchanged are invisible to it, a long transaction can commit behind a watermark that later rows have already advanced, and a version that is overwritten before the next poll is never read. The sweeps below attribute most of the loss to the first cause. The overlap window recovered part of it, raising capture to 98.4% and lowering the share of stale records to 1.8%, a gain of about 0.4 percentage points on each measure. The cost was a read amplification of 5.8, as every poll re-read the rows within the margin.

Snapshot comparison captured 99.8% of the changes and left no record stale, because it sees updates that leave the timestamp unchanged and always reads the final version of a record. Its misses were the versions that were overwritten within one interval. Its cost was a read amplification of 164.5: it read that many rows for every version that it delivered.

The outbox variants lost the writes that bypassed the outbox. Capture was 97.0% for the log reader and 96.8% for the poller, which is close to what a bypass share of 3% of the versions implies, and 3.2% and 3.4% of the records were stale. The poller also had the latency of polling, with a median of 30.0 s.

7.3. Sensitivity of Extraction

Table XV and Fig. 8 vary one factor at a time. Log-based capture is not affected by any of them and captured every change in all rows.

Table 15. Capture (%) As One Factor Varies, With The Other Factors At Their Default Values. Log-Based Capture Was 100% In Every Row. Means Over Four Seeds

Factor	Setting	TS	Overlap	Snapshot	Outbox Log	Outbox Poll
Poll interval (s)	5	97.9	98.5	100.0	97.0	96.6
	15	98.0	98.5	100.0	97.0	96.7
	60	98.0	98.4	99.9	97.0	96.8

	300	94.4	94.7	96.0	97.0	97.0
Long transactions	0%	98.2	98.4	99.9	97.0	97.0
	0.5%	98.0	98.4	99.9	97.0	96.8
	2%	97.5	98.4	99.8	97.0	96.3
	5%	96.3	98.4	99.8	97.0	95.1
Timestamp unchanged	0%	99.7	99.8	99.9	97.0	96.8
	2%	98.0	98.4	99.9	97.0	96.8
	5%	95.6	96.3	99.9	97.0	96.8
Outbox bypass	0%	98.0	98.4	99.9	100.0	99.8
	3%	98.0	98.4	99.9	97.0	96.8
	10%	98.0	98.4	99.9	90.0	89.9

A longer poll interval hurts the strategies that read the current row of a table. At I = 300 s the median latency of polling grew to 146 s, and capture fell to 94.4% for TS, 94.7% with overlap, and 96.0% for snapshots, from 99.9% for snapshots at 60 s. A longer interval overwrites more versions before they are read. The polled outbox keeps every row, so it did not lose more at 300 s (capture 97.0% against 96.8% at 60 s); only its latency grew. Shorter intervals than the default change little in capture and cut latency in proportion, but they need more polls.

The share of long transactions moves TS from 98.2% with none to 96.3% at 5%, while the overlap window holds at about 98.4% across the range. The median long transaction of 20 s is far shorter than the margin of 300 s, which is why the overlap absorbs it. With no long transactions the overlap still gained a little (98.2% for TS and 98.4% with overlap), because the margin also re-reads rows whose timestamp was not updated, as long as that timestamp is still within the margin. A larger share of updates that leave the timestamp unchanged hurts both timestamp strategies, and the overlap window does not remove this loss: with 5% of updates affected, TS captured 95.6% and TS with overlap 96.3%, while snapshots captured 99.9%. The outbox variants follow the share of writes that bypass the outbox one to one: at 10% bypass the log reader captured 90.0%, and with no bypass it captured everything.

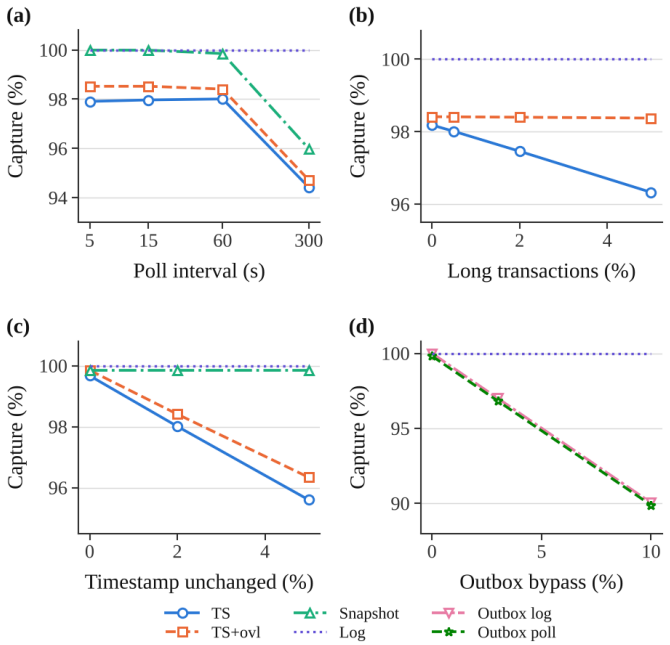


Figure 8. Capture of Changes As One Factor Varies, For the Strategies That the Factor Affects. (A) Poll Interval. (B) Share of Long Transactions. (C) Share of Updates That Leave the Timestamp Unchanged. (D) Share of Writes That Bypass the Outbox. Lines Show the Mean Over Four Seeds

7.4. Query Path

Table XVI gives the results of the SQLite benchmark. The federated variant slowed in proportion to the number of sources, while the pre-normalized table grew more slowly. For Q1, counts by state, the pre-normalized table took 12.9 ms at two sources and 35.4 ms at eight, against 41.9 and 166.1 ms, which is a speedup of 3.2 to 5.0 times. For Q2, the first 50 exceptions, the pre-normalized time stayed near 0.04 ms at every *n* and the federated time grew from 0.13 to 0.51 ms, a speedup of 3.3 to 13.2 times. For Q3, lookups by identifier, the speedup was 1.4 to 4.3 times, since the federated variant must ask every source and the store asks one. The backfill of the three pre-normalized tables, which hold 1.5 million rows in all, took 2.8 s. These times describe one engine, one machine and a warm cache, and the speedups depend on the sizes and indexes chosen here.

Table 16. Results of the Sqlite Benchmark. Speedup Is the Ratio of the Federated to the Pre-Normalized Median

Query	*N*	Federated Median (Ms)	95th	Pre-Normalized Median (Ms)	95th	Speedup
Q1	2	41.9	70.5	12.9	16.0	3.2
	5	110.1	121.3	22.1	23.7	5.0
	8	166.1	222.0	35.4	36.7	4.7
Q2	2	0.13	0.21	0.04	0.05	3.3
	5	0.31	0.36	0.04	0.05	8.1
	8	0.51	0.55	0.04	0.07	13.2
Q3	2	0.10	0.14	0.07	0.09	1.4
	5	0.21	0.25	0.06	0.09	3.4
	8	0.35	0.43	0.08	0.10	4.3

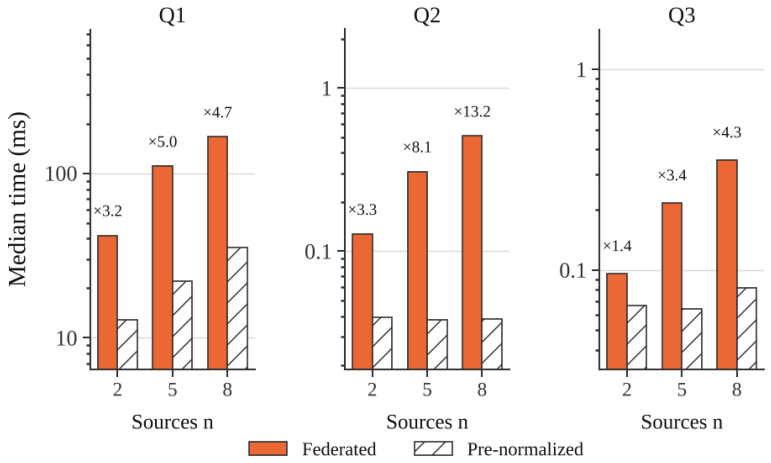


Figure 9. Median Time of the Three Queries at Two, Five and Eight Sources, With Query-Time Translation (Federated) and With the Pre-Normalized Table. The Vertical Axis Is Logarithmic

Table XVII gives the fan-out model. With one source the 99th percentile was 472 ms, and with twelve it was 903 ms, which is 1.9 times as much, while the median grew from 85 to 277 ms. The share of queries that exceeded the 99th percentile of a single source was 11.5% at twelve sources in the draws and 11.4% in the closed form, and the two differed by at most 0.25 percentage points over all *n*. Availability fell to 94.2% at twelve sources. The store answered with a 99th percentile of 100 ms, which is 9.0 times lower than the twelve-source query, and its unavailability was about 1/117 of that of twelve sources. These figures follow from the assumed latencies and availabilities. They show the direction and the rate of the effect; a real installation would have its own numbers.

Table 17. Fan-Out Model: Latency of a Federated Query over N Sources, the Share of Queries That Exceed the 99th Percentile of a Single Source, And Availability. The Last Row Is the Pre-Normalized Store

N	Median (Ms)	95th (Ms)	99th (Ms)	Exceed Single 99th, Draws	Exceed, Closed Form	Availability
1	85	288	472	0.90%	1.00%	99.50%
2	126	363	582	1.92%	1.99%	99.00%
3	155	413	648	2.96%	2.97%	98.51%
5	196	488	740	5.15%	4.90%	97.52%
8	238	547	824	7.71%	7.73%	96.07%
12	277	615	903	11.53%	11.36%	94.16%
Store	25	67	100	–	–	99.95%

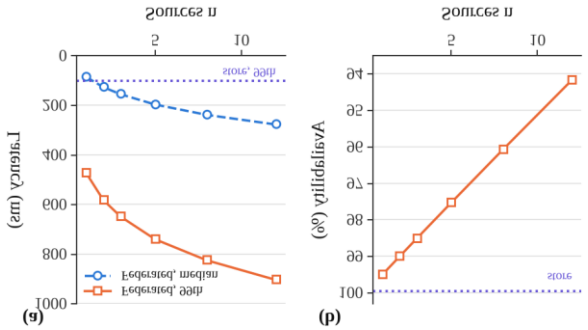


Figure 10. Fan-Out Model. (A) The Median and the 99th Percentile of the Latency of a Federated Query against the Number of Sources. The Dotted Line Is the 99th Percentile of yhe Pre-Normalized Store. (B) Availability of a Federated Query against the Number of Sources. The Dotted Line Is the Availability of the Store

7.5. Transition Guard

Table XVIII gives the results of the guard study. Encounters affected by a mapping fault made up between 15% and 47% of all encounters, depending on the number of faulty codes. The strict guard detected 41% (±3) of the affected encounters when two codes were mapped to a neighboring state, and 66% (±3) when they were mapped to a random state. The lenient guard detected 6% and 35% in the same scenarios. A fault to a neighbor replaces a state by one next to it in the workflow, so the substitute often remains a legal move from the true predecessor or to the true successor, which is why such faults are harder to see than faults to a random state. The encounters of the study follow legal moves only, so without skipped statuses neither guard raised a false alarm.

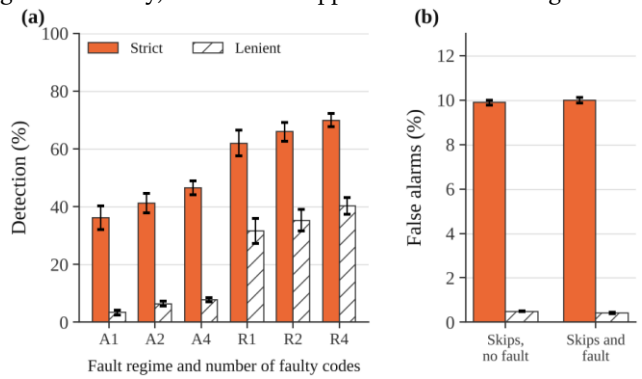


Figure 11. Detection of Affected Encounters By the Strict and the Lenient Guard Under Adjacent (A) and Random (R) Substitutions With One, Two or Four Faulty Codes (A), and False Alarms When 5% Of Sta-tuses are Skipped, Without a Fault and together with Two Adjacent Faulty Codes (B). Error Bars Show 95% Intervals over Trials

Table 18. Results of the Guard Study, Strict / Lenient, Pooled Over 300 Trials. Affected Is the Share of Encounters with a Wrong Mapped Sequence. The Other Measures are defined in the Text. Intervals for Detection over Trials are At Most ±4.5 Points

Scenario	Affected	Detected, Strict / Le-nient	False Alarms, Strict / Lenient	Codes Surfaced, Strict / Lenient
Adjacent, 1 code	15%	36% / 3%	0.0% / 0.0%	87% / 32%
Adjacent, 2 codes	27%	41% / 6%	0.0% / 0.0%	95% / 61%
Adjacent, 4 codes	47%	47% / 8%	0.0% / 0.0%	99% / 81%
Random, 1 code	15%	62% / 32%	0.0% / 0.0%	84% / 51%
Random, 2 codes	27%	66% / 35%	0.0% / 0.0%	99% / 74%
Random, 4 codes	47%	70% / 40%	0.0% / 0.0%	100% / 93%
Skips only	0%	–	9.9% / 0.5%	–
Adjacent, 2 codes, and skips	26%	49% / 7%	10.0% / 0.4%	100% / 79%

False alarms appear once sources skip statuses. With 5% of interior statuses unreported and no fault, the strict guard flagged 9.9% of the encounters, and the lenient guard flagged 0.5%. A replicate with other seeds gave 9.8% and 0.5%. In the combined scenario, with two faulty codes mapped to neighbors and skipped statuses, 26% of the encounters were affected. The strict guard detected 49% of them and raised false alarms on 10.0% of the unaffected ones, so its precision was 64%. The lenient guard detected 7%, raised false alarms on 0.4%, and had a precision of 85%. The results agree with Proposition 3: in no scenario of Table XVIII does the lenient guard have a higher detection rate or a higher false-alarm rate than the strict guard.

The code-level measure shows a second use of the guard. Even when it flags fewer than half of the affected encounters, the strict guard surfaced 100% of the faulty codes in the combined scenario, in the sense that each of them appears in at least one flagged encounter, and the lenient guard surfaced 79%. A review that groups the flagged encounters by native code would therefore see every faulty code among the codes involved, although it would see codes that are not faulty as well, and 20% of all encounters were flagged. Reviewing that many encounters one by one would be a large task, and the native codes are fewer than the encounters. How much a review by code would save depends on how well the faulty codes rank among the others, which the study did not measure.

8. Discussion

8.1. Claims and Evidence

Table XIX lists the claims that the paper makes, with the basis of each. A few follow from the definitions and need no experiment. Others were observed in the synthetic study; they hold under its parameters, and their magnitudes would differ elsewhere. The rest are hypotheses that only a deployment can test, and the paper does not claim them.

Table 19. Claims of the Paper, the Basis of Each, and the Evidence

Claim	Basis	Evidence
A hub needs n adapters where pairwise translation needs $n(n-1)$ translators	By construction	Proposition 1
The store can be rebuilt from the log, and corrected rules can be applied again	By construction	Proposition 2
A lenient guard flags a subset of what a strict guard flags	By construction	Proposition 3; every scenario agrees
Log-based capture misses no committed change and has the lowest latency	Observed in the synthetic study	100% captured; median 0.5 s; needs log access
Timestamp polling loses changes, mostly updates that leave the timestamp unchanged	Observed in the synthetic study	2.0% missed; the loss follows the share of unchanged

		timestamps
A pre-normalized store answers aggregate queries faster than query-time translation	Observed in an in-process benchmark	3.2 to 5.0 times for Q1; no net-work
Query-time fan-out lowers availability and raises tail latency as sources are added	Follows from the model; mag-nitude from assumed parame-ters	Availability 94.2% at twelve sources
A strict guard exposes more mapping faults than a lenient guard, and raises more false alarms when sources skip statuses	Observed in the synthetic study	Detection 41% against 6%; false alarms 9.9% against 0.5%
The pattern reduces manual status checks and im-proves coordination	Hypothesis	Not tested
Sub-second dashboard response from commit to screen	Design target	Not measured
Accuracy of the mapping on real events	Not examined	Needs labeled data

The synthetic results are best read as an account of mechanisms. Timestamp polling loses changes because of how timestamps are written, and the share lost depends on how many updates leave the timestamp unchanged. Query-time translation is slower because it repeats the translation and waits for the slowest source, and the gap widens with the number of sources. A strict guard finds more faults than a lenient one and raises more false alarms when sources skip statuses, and the balance depends on how often they do. In each case the mechanism and the factor it depends on would carry over to a deployment, while the numbers would not.

8.2. Appraisal against Six Criteria

The design goals can be judged against modularity, interoperability, extensibility, consistency, traceability and security (Table XX). The appraisal is analytic for the most part. The column on support says what in this paper bears on the judgment, and the column on risk says what remains open.

Table 20. Appraisal of the Pattern against Six Criteria

Criterion	How the pattern addresses it	Support in this paper	Open risk
Modularity	Extractors, registry, engine, state ser-vice, store and dashboard are separate components	Layer structure; the registry is configuration and not code	Coupling through the envelope; changes to it need coordination
Interoperability	Sources meet in the envelope and the canonical states, not in each other's schemas	Proposition 1; projection onto FHIR Task status	The meaning of the states needs organizational agree-ment
Extensibility	A new source needs one adapter and one set of rules	Proposition 1	Effort of mapping and testing grows with the size of a vocab-ulary
Consistency	One canonical state per stream; illegal moves are flagged; rules for duplicate and late events	Proposition 2; guard study	Asynchrony leaves a lag; rules E1 to E6 are untested
Traceability	Native value, event and mapping ver-sion stored with each state; registry changes audited	Replay under a recorded version (Proposition 2)	Retention limits can remove raw rows
Security	Role-based access, encryption, audit, least privilege	List of controls	Not tested; compliance depends on the deployment

8.3. Choosing an Extraction Technique

The extraction results suggest a way to choose a technique for each source (Table XXI). The choice depends on facts about the source that can be tested before the choice is made. One test compares a week of changes found by a snapshot comparison with the changes found by the proposed watermark query, which exposes updates that leave the timestamp unchanged and late commits at low cost. Another asks the owner of the source which writers touch the status table, which exposes paths that would bypass an outbox.

Table 21. Choosing an Extraction Technique for a Source

Situation	Suggested technique	Reason
Log access and vendor support are available	Log-based capture	No committed change is lost; lowest latency; the source is unchanged
The owner accepts a change in the source and log access is not available	Outbox, read from the log of the outbox table where possible	Complete for writes that use the outbox; audit every writer first
Read access only; every writer maintains the timestamp; few long transactions	Timestamp watermark with an overlap margin above the longest transaction	Cheap in reads; depends on every writer updating the timestamp; deletions stay invisible
Read access only; timestamps are unreliable; the table is small	Snapshot comparison	Sees unchanged timestamps and deletions; read cost grows with the table
Deletions must be seen	Log-based capture or snapshot comparison	A timestamp query cannot see a deleted row
Latency must be below a poll interval	Log-based capture	Polling latency is about half the interval

8.4. The Mapping Is the Weak Point

A wrong mapping shows staff a state that is false, and the error is silent unless something checks for it. The pattern has four defenses against this. The registry gives every rule an owner, a test and a version, so that errors have a place to be corrected. The store keeps the native code beside the canonical state, so a reader can see what the source said. The guard turns impossible progressions into flags. Grouping the flags by native code, as the code-level measure of Section VII suggests, directs attention to the rules that need review. These defenses reduce the risk without removing it, and the detection rates of Section VII show how partial the guard is: even the strict guard exposed fewer than half of the encounters affected by faults to a neighboring state.

Over-standardization is a separate risk. Eight canonical states are coarser than the vocabularies of the sources. A difference that matters clinically, such as the reason for a hold, may be lost in the mapping to Exception. The native code and qualifiers stay in the store for that reason, and the dashboard can show them on drill-down. Choosing the canonical states is a design decision that the organization owns, and the eight states of this paper are a starting point.

8.5. Alternatives

A federated query layer, or data virtualization, translates at query time [9], [32]. It needs no store, no capture and no synchronization, and its data are as current as the sources. The study quantifies what it costs in latency and availability, and a design with few sources, small tables and a tolerance for the sources' availability may reasonably accept that. A warehouse with batch loads is simpler still and answers analytical questions well, but its status view is as old as the last load. Such a warehouse can also be a consumer of the same event log, so the two are not exclusive. An interface engine with pairwise translation is familiar to integration teams, and its artifact counts follow Proposition 1. A FHIR facade over each system gives standard access to the data [6], [27], and it can serve as an adapter for the pattern, but it does not remove the need to map local status values to a common vocabulary.

8.6. Operational Considerations

The mapping registry needs people with authority over the meaning of each source's statuses, and the separation of proposer and approver needs a procedure that the organization can staff. The dashboard is a coordination tool, and it must not be mistaken for a record of care or a source of clinical decisions. Its freshness display is part of its safety, since a view that looks current and is not can mislead. Retention of events and of the raw rows referenced by them is a policy decision with privacy consequences, and retention is also what makes replay possible. After a loss the store is rebuilt from the log, so it is reproduced rather than restored from a backup.

9. Limitations

All data are synthetic, and the parameters are illustrative. The study shows how mechanisms behave under stated assumptions, and it does not estimate how a real hospital would fare.

Part B models one status table per source. Real sources have several tables, deletions, triggers, bulk updates and clocks that drift, and none of these is modeled. The outage process is simple, and the lag of the extraction pipeline after an outage is not modeled beyond the wait for its end.

Part C measures one database engine in one process, with a warm cache, no network and no concurrent writers. The fan-out model assumes independent sources with lognormal latencies. Real sources often share infrastructure and slow down together, which would change both the tail and the availability product. The model's output follows from its inputs, and what it shows is the rate at which the effect grows.

Part D uses randomly generated vocabularies, one transition table and one table of walk probabilities. Faults occur only in the mapping, and skips are uniform. Statuses of one encounter come from several sources without clock skew or reordering. The false-alarm rate of the strict guard depends on the assumed skip rate, and a real source may skip far more or far less. The study counts flagged encounters, and it does not model how people resolve them or how many flags a team can handle.

The rules E1 to E6 for duplicate, late, missing and illegal events are specified but not evaluated. Identity resolution across systems is assumed. The design is read-only and does not treat corrections and retroactive changes to source data beyond what replay offers. The security controls are listed but not tested, and no compliance assessment was made. No user study was conducted, so the paper has no evidence about staff time, the number of status calls, coordination or patient outcomes. The code has not been reviewed independently, and no independent replication exists.

10. Future Work

An evaluation of the pattern in practice would proceed in stages. A shadow-mode study would replay recorded events from one or two sources in a staging environment, compare the captured changes with the source of truth, and measure end-to-end propagation delay and dashboard response against the sub-second target. A labeled sample of events, with canonical states assigned by the stewards of each source, would give the accuracy of the mapping per source and per code. Transition tables could be derived from logs by process mining [43], [44] and the strict and lenient modes calibrated against observed skip rates. Rules E3 and E4 should be tested under injected reordering and clock skew across sources. Load tests with real networks and concurrent readers should replace the in-process benchmark. A study with staff should measure the time needed to answer where a patient's process stands, the number of telephone calls and the use of the quarantine queue. A security review should test the controls of Table IX against a threat model of the store.

11. CONCLUSION

Departmental systems describe workflow status in their own words, and a coordinator who needs one answer must combine several. We specified a pattern that extracts committed changes, translates native statuses into eight canonical states with a versioned and auditable registry, checks every status stream against a transition table, and serves a dashboard from a read-only store, with no change to the sources. A formal model showed that a hub needs n adapters where pairwise translation needs $n(n-1)$ translators, that the store can be rebuilt by replay, and that a lenient guard flags a subset of what a strict guard flags.

In a synthetic study with illustrative parameters, timestamp polling missed 2.0% of the changes against none for log-based capture, which also delivered the median change in 0.5 s against 30.0 s. A pre-normalized store answered aggregate queries 3.2 to 5.0 times faster than query-time translation in an in-process benchmark, and in a fan-out model the availability of a twelve-source query fell to 94.2%. With two faulty codes mapped to neighboring states, the strict guard exposed 41% of the affected encounters and the lenient guard 6%. When sources skipped statuses, the strict guard raised false alarms on 9.9% of the encounters and the lenient guard on 0.5%.

The study shows how the mechanisms behave and what they depend on. It does not show that the pattern reduces manual status checks or improves coordination, nor does it measure the sub-second response that the design aims at. These are questions for a staged evaluation on real systems.

References

[1] E. Kihuba *et al.*, "Assessing the ability of health information systems in hospitals to support evidence-informed decisions in Kenya," *Glob. Health Action*, vol. 7, no. 1, Art. no. 24859, 2014, doi: 10.3402/gha.v7.24859.
[2] J. Adler-Milstein, A. J. Holmgren, P. Kralovec, C. Worzala, T. Searcy, and V. Patel, "Electronic health record adoption in US hospitals: The emergence of a digital "advanced use" divide," *J. Am. Med. Inform. Assoc.*, vol. 24, no. 6, pp. 1142–1148, 2017, doi: 10.1093/jamia/ocx080.
[3] A. S. O'Malley, K. Draper, R. Gourevitch, D. A. Cross, and S. H. Scholle, "Electronic health records and support for primary care teamwork," *J. Am. Med. Inform. Assoc.*, vol. 22, no. 2, pp. 426–434, 2015, doi: 10.1093/jamia/ocu029.
[4] A. Friedman *et al.*, "A typology of electronic health record workarounds in small-to-medium size primary care practices," *J. Am. Med. Inform. Assoc.*, vol. 21, no. e1, pp. e78–e83, 2014, doi: 10.1136/amiajnl-2013-001686.
[5] Health Level Seven International, *HL7 Messaging Standard Version 2.5.1*. Ann Arbor, MI, USA: HL7 International, 2007.
[6] D. Bender and K. Sartipi, "HL7 FHIR: An Agile and RESTful approach to healthcare information exchange," in *Proc. 26th IEEE Int. Symp. Comput.-Based Med. Syst. (CBMS)*, 2013, pp. 326–331, doi: 10.1109/CBMS.2013.6627810.
[7] HL7 International, "FHIR Release 4 (v4.0.1)," 2019. [Online]. Available: https://hl7.org/fhir/R4/
[8] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley, 2003.
[9] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Comput. Surv.*, vol. 22, no. 3, pp. 183–236, 1990, doi: 10.1145/96602.96604.
[10] M. Lenzerini, "Data integration: A theoretical perspective," in *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst. (PODS)*, 2002, pp. 233–246, doi: 10.1145/543613.543644.
[11] S. Das *et al.*, "All aboard the Databus!: LinkedIn's scalable consistent change data capture platform," in *Proc. 3rd ACM Symp. Cloud Comput. (SoCC)*, 2012, Art. no. 18, doi: 10.1145/2391229.2391247.
[12] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. 6th Int. Workshop Netw. Meets Databases (NetDB)*, Athens, Greece, 2011, pp. 1–7.
[13] D. Harel, "Statecharts: a visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987, doi: 10.1016/0167-6423(87)90035-9.
[14] C. Richardson, *Microservices Patterns*. Shelter Island, NY, USA: Manning, 2018.
[15] M. Kleppmann, *Designing Data-Intensive Applications*. Sebastopol, CA, USA: O'Reilly Media, 2017.
[16] A. J. Holmgren, V. Patel, and J. Adler-Milstein, "Progress in interoperability: Measuring US hospitals' engagement in sharing patient data," *Health Aff.*, vol. 36, no. 10, pp. 1820–1827, 2017, doi: 10.1377/hlthaff.2017.0546.
[17] B. W. Pickering *et al.*, "The implementation of clinician designed, human-centered electronic medical record viewer in the intensive care unit: A pilot step-wedge cluster randomized trial," *Int. J. Med. Inform.*, vol. 84, no. 5, pp. 299–307, 2015, doi: 10.1016/j.ijmedinf.2015.01.017.
[18] D. Dowding *et al.*, "Dashboards for improving patient care: Review of the literature," *Int. J. Med. Inform.*, vol. 84, no. 2, pp. 87–100, 2015, doi: 10.1016/j.ijmedinf.2014.10.001.
[19] W. Martinez, A. L. Threatt, S. T. Rosenbloom, K. A. Wallston, G. B. Hickson, and T. A. Elasy, "A patient-facing diabetes dashboard embedded in a patient web portal: Design sprint and usability testing," *JMIR Hum. Factors*, vol. 5, no. 3, Art. no. e26, 2018, doi: 10.2196/humanfactors.9569.
[20] B. Frandji, J. Schot, M. Joubert, I. Soadyh, and A. Kilsdonk, "The RICHE reference architecture," *Med. Inform.*, vol. 19, no. 1, pp. 1–11, 1994, doi: 10.3109/14639239409044716.
[21] A. Winter, B. Brigl, G. Funkat, A. Häber, O. Heller, and T. Wendt, "3LGM2-Modeling to support management of health information systems," *Int. J. Med. Inform.*, vol. 76, no. 2-3, pp. 145–150, 2007, doi: 10.1016/j.ijmedinf.2006.07.007.
[22] J. D. Duke *et al.*, "Regenstrief Institute's Medical Gopher: A next-generation homegrown electronic medical record system," *Int. J. Med. Inform.*, vol. 83, no. 3, pp. 170–179, 2014, doi: 10.1016/j.ijmedinf.2013.11.004.
[23] P. Rajkovic, D. Jankovic, and A. Milenkovic, "Developing and deploying medical information systems for Serbian public healthcare: Challenges, lessons learned and guidelines," *Comput. Sci. Inf. Syst.*, vol. 10, no. 3, pp. 1429–1454, 2013, doi: 10.2298/CSIS120523056R.
[24] C. Tsavatewa, P. F. Musa, and I. Ramsingh, "Integration of Footprints information systems in palliative care: The case of Medical Center of Central Georgia," *J. Med. Syst.*, vol. 36, no. 3, pp. 1511–1521, 2012, doi: 10.1007/s10916-010-9612-y.
[25] Ö. Köksal and B. Tekinerdogan, "Obstacles in data distribution service middleware: A systematic review," *Future Gener. Comput. Syst.*, vol. 68, pp. 191–210, 2017, doi: 10.1016/j.future.2016.09.020.
[26] S. Sancar Gozukara, B. Tekinerdogan, and C. Catal, "Obstacles of on-premise enterprise resource planning systems and solution directions," *J. Comput. Inf. Syst.*, vol. 62, no. 1, pp. 141–152, 2022, doi: 10.1080/08874417.2020.1739579.
[27] J. C. Mandel, D. A. Kreda, K. D. Mandl, I. S. Kohane, and R. B. Ramoni, "SMART on FHIR: a standards-based, interoperable apps platform for electronic health records," *J. Am. Med. Inform. Assoc.*, vol. 23, no. 5, pp. 899–908, 2016, doi: 10.1093/jamia/ocv189.

[28] W. E. Hammond, "The making and adoption of health data standards," *Health Aff.*, vol. 24, no. 5, pp. 1205–1213, 2005, doi: 10.1377/hlthaff.24.5.1205.
[29] A. Doan, A. Halevy, and Z. Ives, *Principles of Data Integration*. Waltham, MA, USA: Morgan Kaufmann, 2012.
[30] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *VLDB J.*, vol. 10, no. 4, pp. 334–350, 2001, doi: 10.1007/s007780100057.
[31] P. Vassiliadis, "A survey of extract–transform–load technology," *Int. J. Data Warehous. Min.*, vol. 5, no. 3, pp. 1–27, 2009, doi: 10.4018/jdwm.2009070101.
[32] R. F. van der Lans, *Data Virtualization for Business Intelligence Systems*. Waltham, MA, USA: Morgan Kaufmann, 2012.
[33] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA, USA: Addison-Wesley, 2003.
[34] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and tables: Two sides of the same coin," in *Proc. Int. Workshop Real-Time Bus. Intell. Anal. (BIRTE)*, Rio de Janeiro, Brazil, 2018, Art. no. 1, doi: 10.1145/3242153.3242155.
[35] G. Wang *et al.*, "Consistency and completeness: Rethinking distributed stream processing in Apache Kafka," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2021, pp. 2602–2613, doi: 10.1145/3448016.3457556.
[36] P. Helland, "Idempotence is not a medical condition," *Commun. ACM*, vol. 55, no. 5, pp. 56–65, 2012, doi: 10.1145/2160718.2160734.
[37] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, 2015, doi: 10.14778/2824032.2824076.
[38] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–62, 2012, doi: 10.1145/2187671.2187677.
[39] 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.
[40] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data management in microservices: State of the practice, challenges, and research directions," *Proc. VLDB Endow.*, vol. 14, no. 13, pp. 3348–3361, 2021, doi: 10.14778/3484224.3484232.
[41] W. M. P. van der Aalst, "The application of Petri nets to workflow management," *J. Circuits Syst. Comput.*, vol. 8, no. 1, pp. 21–66, 1998, doi: 10.1142/S0218126698000043.
[42] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003, doi: 10.1023/A:1022883727209.
[43] W. van der Aalst *et al.*, "Process mining manifesto," in *Business Process Management Workshops (BPM 2011)*, ser. Lecture Notes in Business Information Processing, vol. 99. Berlin, Germany: Springer, 2012, pp. 169–194, doi: 10.1007/978-3-642-28108-2_19.
[44] E. Rojas, J. Munoz-Gama, M. Sepúlveda, and D. Capurro, "Process mining in healthcare: A literature review," *J. Biomed. Inform.*, vol. 61, pp. 224–236, 2016, doi: 10.1016/j.jbi.2016.04.007.
[45] M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers, *Fundamentals of Business Process Management*, 2nd ed. Berlin, Germany: Springer, 2018, doi: 10.1007/978-3-662-56509-4.
[46] E. Brewer, "CAP twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012, doi: 10.1109/MC.2012.37.
[47] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002, doi: 10.1145/564585.564601.
[48] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, 2009, doi: 10.1145/1435417.1435432.
[49] P. Helland, "Life beyond distributed transactions: An apostate's opinion," in *Proc. 3rd Biennial Conf. Innov. Data Syst. Res. (CIDR)*, Asilomar, CA, USA, 2007, pp. 132–141.
[50] H. Garcia-Molina and K. Salem, "Sagas," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1987, pp. 249–259, doi: 10.1145/38713.38742.
[51] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978, doi: 10.1145/359545.359563.
[52] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013, doi: 10.1145/2408776.2408794.
[53] 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.
[54] 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. 18, 2016, doi: 10.13063/2327-9214.1244.
[55] I. P. Fellegi and A. B. Sunter, "A theory for record linkage," *J. Amer. Statist. Assoc.*, vol. 64, no. 328, pp. 1183–1210, 1969, doi: 10.1080/01621459.1969.10501049.
[56] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, 1996, doi: 10.1109/2.485845.
[57] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM Trans. Inf. Syst. Secur.*, vol. 4, no. 3, pp. 224–274, 2001, doi: 10.1145/501978.501980.
[58] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975, doi: 10.1109/PROC.1975.9939.
[59] U.S. Department of Health and Human Services, "Security standards for the protection of electronic protected health information," 45 C.F.R. pt. 164, subpt. C.
[60] U.S. Department of Health and Human Services, "Privacy of individually identifiable health information," 45 C.F.R. pt. 164, subpt. E.

[61] Joint Task Force, *Security and Privacy Controls for Information Systems and Organizations*, NIST Special Publication 800-53 Rev. 5, Gaithersburg, MD, USA, 2020, doi: 10.6028/NIST.SP.800-53r5.
[62] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, *Zero Trust Architecture*, NIST Special Publication 800-207, Gaithersburg, MD, USA, 2020, doi: 10.6028/NIST.SP.800-207.
[63] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Inf. Process. Manag.*, vol. 45, no. 4, pp. 427–437, 2009, doi: 10.1016/j.ipm.2009.03.002.
[64] A. M. Law, *Simulation Modeling and Analysis*, 5th ed. New York, NY, USA: McGraw-Hill, 2015.