

SCHEMA EVOLUTION WITHOUT DOWNTIME: A PROTOCOL FOR BREAKING CHANGES IN STREAMING PIPELINES**Santosh Kumar Maddali**

Sr. Data Engineer

ABSTRACT:

Streaming pipelines bind producers, brokers, stream processors, and analytical sinks together through a shared serialization contract. When that contract must change in a way existing consumers cannot interpret, the conventional remedy is a coordinated stop, redeploy, and replay, which consumes the availability budget and violates service level objectives on precisely the pipelines that can least afford it. This paper presents a systematic treatment of breaking schema change in always on streaming systems. We define a compatibility model over producer and consumer schema sets and show that the pairwise compatibility checks enforced by conventional schema registries are necessary but not sufficient for pipeline safety, because they reason only about the wire encoding and ignore three additional binding surfaces: checkpointed operator state, sink table contracts, and semantic redefinition of syntactically stable fields. We show that the transitive compatibility modes registries actually provide cannot admit contraction when a field lacked a default in an earlier registered schema, and replace them with a live compatibility relation quantified over retained writers and leaseholding readers. We introduce a taxonomy of seven classes of breaking change and map each class to a safe rewriting. Building on this, we specify an expand, migrate, contract protocol that decomposes an arbitrary breaking change into a sequence of individually compatible transitions, none of which requires a synchronized restart, together with a reference architecture comprising a policy enforcing contract registry, a schema aware projection layer, and a state migration mechanism that rebuilds keyed operator state without stopping the incumbent job. We then derive a cost model expressing the protocol's overhead in record width, storage, state rebuild time, and elapsed migration duration as closed form functions of deployment parameters. Although storage cost scales linearly with event rate, the quantities governing operational tolerability do not: relative record growth is rate independent, and state rebuild time depends only on the operator's state horizon and the ratio of replay to live throughput, so a pipeline at one million events per second rebuilds no more slowly than one at ten thousand given proportionate resources.

Keywords

schema evolution, stream processing, data contracts, backward compatibility, zero downtime deployment, Apache Kafka, Avro, stateful operators, data engineering

I. INTRODUCTION

A production streaming pipeline is a distributed agreement about the shape of data. Producers encode records against a schema, brokers persist the encoded bytes, stream processors decode them into typed representations, and sinks project them into tables whose columns are themselves a schema. Every one of these components holds an independent copy of an assumption about structure, and those copies are updated by different teams on different release cadences. The pipeline is correct only while the assumptions agree.

Schema change is therefore not an occasional event but the steady state condition of production data platforms. A study of ten large open source database applications spanning more than 160,000 revisions found that schemas evolve frequently across the application lifecycle, and that schema changes induce substantial modification in the surrounding code [9]. Where schemas do not change, the reason is often not that change is unnecessary: longitudinal analysis reports a pronounced gravitation to rigidity, with most tables living quiet lives and activity concentrated early in a database's life [20]. We read this as a symptom rather than a steady state, since teams stop evolving structures whose dependents they cannot safely update. The same pressure applies with greater force to streaming platforms, because a single topic is frequently consumed by dozens of independently owned applications, and the fan out means the cost of a change is borne by parties who did not request it [18].

Modern serialization frameworks anticipate a subset of this problem. Avro and Protocol Buffers define resolution rules under which a reader using one schema can interpret bytes written under another [4], [26]. Schema registries operationalize these rules by rejecting registrations that violate a configured compatibility mode. This machinery handles the easy cases well. Adding an optional field with a default, or removing a field that already carries a default, is safe by construction and requires no coordination at all.

The difficulty is the residue. A meaningful fraction of the changes a business actually needs are breaking by definition: narrowing a type, making an optional field required, renaming a key, splitting one field into two, changing the unit or the timezone of a value, or altering the partitioning key that determines how state is distributed. For these, the registry has nothing constructive to say. It can only refuse. The remaining option is the coordinated cutover: quiesce the producers, drain the topic, stop every consumer, deploy new binaries everywhere, and restart. On a pipeline with a strict recovery time objective, this is precisely the operation that is not permitted.

This paper argues that the coordinated cutover is avoidable in essentially all cases, and that the reason it persists is a modeling gap rather than a fundamental limit. Registry compatibility is a property of a pair of schemas. Pipeline safety is a property of a system that also includes checkpointed operator state, downstream table definitions, and the semantics that consumers attach to field values. Once the full set of binding surfaces is made explicit, breaking changes decompose into sequences of nonbreaking ones.

We make four contributions.

- A compatibility model over the four binding surfaces of a streaming pipeline, with proofs that registry level transitive compatibility is both insufficient for pipeline safety and, for contraction, unsatisfiable in principle; and a live compatibility relation that is reachable (Section III).
- A taxonomy of seven classes of breaking change, each with a characterization of why serialization level resolution fails and a corresponding safe rewriting (Section III, Table I).
- The expand, migrate, contract protocol, which factors any breaking change into individually compatible transitions and admits rollback at every intermediate state, with a reference architecture implementing it (Sections IV and V).
- A cost model deriving the protocol's storage, record width, rebuild time, and duration overheads in closed form from deployment parameters, with a worked instantiation spanning three orders of magnitude in event rate (Section VI).

II. BACKGROUND AND RELATED WORK

A. Schema Versioning and Evolution in Databases

The problem of changing structure under live traffic predates streaming systems. Roddick's survey established the vocabulary that remains in use, distinguishing schema modification, schema evolution, and schema versioning [1]. The PRISM workbench demonstrated that a well chosen algebra of schema modification operators makes migrations invertible and query rewriting mechanical [3], and the follow up work generalized this into an automated evolution process with provable information preservation properties [8].

The closest antecedent to our protocol is the online, asynchronous schema change mechanism in F1 [7]. Faced with a distributed database whose servers could not be assumed to hold the same schema version simultaneously, the authors decompose a change into intermediate states, each of which is safe against the concurrent presence of adjacent versions. Our protocol adopts the same insight but transposes it to a setting with materially different constraints: the shared substrate is an append only log rather than a mutable store, records already written cannot be rewritten in place, and the reader population is open ended and not centrally administered.

In the practitioner literature, the expand and contract idea appears as a database refactoring pattern under the name parallel change [2], and the general discipline of decoupling service releases through tolerant readers is a standing recommendation in microservice architecture [18]. Empirical work on web APIs shows that even where compatibility discipline is nominally in force, coupling reasserts itself and consumers break [13]. Large scale analysis of GraphQL schemas similarly finds that prescribed naming and design conventions are followed only inconsistently in practice [25]. Studies of schema and code co-evolution confirm that the application layer changes in lockstep with structure, so any protocol that assumes code can be held constant during a migration is unrealistic [9]. Case study work on long lived operational databases further shows that reconstructing the history of a schema is itself a substantial undertaking, because the record of what changed and why is seldom maintained alongside the schema [12].

B. Serialization Formats and Compatibility Modes

Avro's resolution rules define when a reader schema can decode data written under a writer schema, keyed on field names and defaults [26]. Protocol Buffers achieves a similar effect through numeric field tags, which decouples compatibility from field naming at the cost of requiring tag discipline [4]. Avro additionally allows a reader to declare aliases, so that a field renamed at the writer can still be matched at the reader [26]. Aliases are less general than they appear. Because the alias is declared in the reader schema, the safe deployment order inverts: every reader must carry the alias before any writer emits the new name, the reverse of the ordering this paper argues for elsewhere. Nor does a stale reader fail cleanly. If the renamed field carries a default, that reader silently

substitutes the default for the value actually written, a corruption no structural check detects. Registries expose these rules as configurable modes, conventionally backward, forward, full, and their transitive variants [22], [28]. A backward compatible change permits a new reader to consume old data; a forward compatible change permits an old reader to consume new data; full compatibility requires both.

The critical observation, developed in Section III, is that these modes quantify over schema pairs and say nothing about the state a consumer has already accumulated, nor about the tables into which it writes.

C. Streaming Platforms and Stateful Processing

Log structured brokers provide the durable, replayable substrate our protocol depends upon [5], [16], and the log as unifying abstraction framing clarifies why the same bytes must serve many independently evolving readers [10], [19]. Processing semantics over unbounded, out of order data are governed by the Dataflow model [14], with fault tolerant realizations in MillWheel [6], Flink [15], Samza [21], and Structured Streaming [23]. All of these maintain keyed operator state across checkpoints, and it is exactly this state that a naive schema migration invalidates. Downstream, transactional table formats such as Delta Lake enforce their own column level contract at commit time, adding a fourth binding surface [27]. Work on elastic scaling and automatic reconfiguration [11], [24] addresses operational change to topology rather than to data structure, and is complementary to what we propose. General architectural treatments of these systems provide the design context [17], [22].

To our knowledge, no prior work provides a unified protocol for breaking schema change that simultaneously accounts for wire format, operator state, sink contracts, and semantic drift in an open consumer population.

III. PROBLEM FORMULATION

A. System Model

We model a pipeline as a directed acyclic graph $G = (V, E)$ whose vertices are processing stages and whose edges are topics. Let P denote the set of producers and C the set of consumers of a topic t . At any wall clock instant, the producers of t collectively write under a set of writer schemas $W(t)$, and its consumers read under a set of reader schemas $R(t)$. In a heterogeneous deployment both sets have cardinality greater than one, since rolling deployments guarantee version skew.

Because a topic is a retained log, the records resident in t at time τ were written under the union of every writer schema active during the retention window $[\tau - \delta, \tau]$, not merely those active at τ . Denote this historical set $W^\delta(t)$.

Definition 1 (Wire safety): A topic t is wire safe if every reader schema in $R(t)$ can resolve every writer schema in $W^\delta(t)$.

Registry compatibility modes approximate this condition, but none expresses it. Backward transitive compatibility guarantees that the newest reader resolves all historical writers, yet places no constraint on older readers against newer writers, which is the failure mode a rolling deployment actually produces. Full transitive compatibility quantifies in both directions, which appears to be what wire safety requires. It is not, and the difference is decisive.

The quantification is wrong. Definition 1 ranges over $W^\delta(t)$, the writer schemas still retained, and over the readers actually live. Transitive modes range over every version ever registered under the subject, a set that only grows and from which nothing ages out. That set is strictly larger than the one safety depends on, and the excess is not conservatism but permanent disablement.

Proposition 1 (Contraction is unreachable under transitive modes): Let σ_1 be a registered schema containing symbol s declared without a default. Then no schema σ' omitting s is ever accepted under full transitive compatibility, irrespective of which intermediate schemas are registered first.

Proof sketch: Full transitive compatibility requires forward compatibility against every previously registered version, so σ_1 must resolve data written under σ' . Records written under σ' omit s . Resolution of a field present in the reader and absent from the writer succeeds only if the reader supplies a default, and σ_1 supplies none. Resolution fails, and σ' is rejected. Registering an intermediate σ^+ that gives s a default does not alter σ_1 , which remains in the checked set for the lifetime of the subject. The rejection is permanent.

The consequence is severe for any protocol ending in contraction. A field can be removed only if every historical reader schema against which the registry checks already supplied a default for it; nullability is not sufficient, since a union typed field declared without a default fails resolution exactly as a scalar one does. Absent that, the deprecated symbols can never be removed and the expanded representation must be carried indefinitely. What is needed is a compatibility relation quantified over the set safety actually depends on. Let $L(\tau)$ denote the reader schemas held under an unexpired lease at time τ , in the sense made precise in Section V.

Definition 2 (Live compatibility): A transition to σ' is live compatible at time τ if σ' resolves every writer schema in $W^\delta(t)$, and every reader schema in $L(\tau)$ resolves σ' .

Live compatibility implies wire safety by construction, and unlike the transitive modes it is reachable, though not for the reason one might first suppose. Retained records that still carry s are not themselves an obstacle: a reader whose schema omits s simply ignores it, since writer fields absent from the reader are discarded during resolution. The binding constraint is the second clause. A writer may omit s once no leaseholding reader requires it, and that set is bounded by leases rather than by the complete history of the subject, which is precisely what the transitive modes cannot express. Retention enters through the first clause instead, and bites when the target schema requires information historical writers never wrote. The protocol of Section IV assumes live compatibility rather than a stock registry mode, and Section VII returns to what that costs.

B. The Four Binding Surfaces

Wire safety is necessary but not sufficient. A consumer binds to a topic across four surfaces, and a change is safe only if it respects all four.

S1, the wire encoding. The byte level resolution relationship between writer and reader schemas. This is what registries check.

S2, the operator state. A stateful consumer maintains keyed state, checkpointed and restored across restarts. That state was serialized under a schema. If a change alters the key extraction function or the value layout, the restored state is either unreadable or, worse, readable but misassigned to the wrong keys. Registries have no visibility into checkpoint contents.

S3, the sink contract. A consumer that writes to a transactional table format or a warehouse binds to a column set with declared types and nullability [27]. A change that is legal on the wire may still be rejected at commit, or silently coerced, at the sink.

S4, the semantic contract. The meaning a consumer attaches to a value. Changing a field from local time to coordinated universal time, or an amount from dollars to cents, is invisible to every structural check yet corrupts every downstream aggregate.

Proposition 2: Wire safety on S1 does not imply pipeline safety.

Proof sketch: It suffices to exhibit a change that is wire safe and unsafe. Let a keyed aggregation consume t partitioned by `record_key`, and let the change replace `record_key` with a globally unique `global_key` while retaining the old field with a default. Every reader in $R(t)$ resolves every writer in $W^\delta(t)$, so t is wire safe. On restart, however, the operator restores state keyed by the old identifier and merges it with records keyed by the new one, producing aggregates that double count the affected population. The violation is on S2, which S1 does not observe.

C. A Taxonomy of Breaking Changes

Table I enumerates seven classes drawn from the literature surveyed in Section II, the surfaces each violates, and the rewriting that renders each safe. The classes are intended to cover representative changes reported in the schema evolution literature; we do not claim exhaustiveness in general.

Three observations follow. First, every rewriting introduces a period during which both the old and the new representation are present, which is the defining characteristic of the protocol in Section IV. Second, class B6 is the only one that mandates state rebuilding, and it is correspondingly the most expensive. Third, class B7 admits no structural remedy at all: because the violation is at S4, the only safe response is to make the semantics syntactically visible so that a structural check can see it.

TABLE I TAXONOMY OF BREAKING SCHEMA CHANGES, VIOLATED BINDING SURFACES, AND SAFE REWRITINGS

Class	Change	Surfaces	Safe rewriting
B1	Field removal without default	S1, S3	Deprecate in catalog, retain during reader migration and verification, then drop once the live reader gate is satisfied
B2	Field rename	S1, S2	Add new field, dual populate, migrate readers, retire old field
B3	Type narrowing or incompatible promotion	S1, S3	Add parallel field of target type, validate range on a shadow stream, cut over, retire
B4	Optional to required	S1	Backfill defaults at the producer, assert nonnull for one full window, then tighten
B5	Cardinality change (scalar to repeated)	S1, S3	Introduce repeated field, mirror scalar as singleton, migrate readers, retire scalar
B6	Key or partition redefinition	S2	Shadow topic under the new key, replay and rebuild state, atomic reader swap
B7	Semantic redefinition (unit, timezone, enumeration)	S4	Version the field name or attach a semantic tag; never mutate meaning in place

IV. THE EXPAND, MIGRATE, CONTRACT PROTOCOL**A. Design Principles**

The protocol rests on four principles.

Additive first. No transition removes or redefines a symbol that any live reader may still bind to. Removal is deferred until observation proves it unreferenced.

Producer leads, consumer follows. Producers emit the superset representation before any consumer is permitted to depend on it. This ordering guarantees that a consumer upgraded early finds the data it expects, whereas the reverse ordering guarantees that it does not.

Reversible at every state. Each intermediate state is a valid steady state, so a migration may halt at any step and remain there indefinitely without corrective action. This removes the deployment window pressure that makes cutovers dangerous. Reversibility is not free for the one transition that carries state: reverting a completed reader swap requires the superseded job to have been kept warm, as Section IV-C makes precise.

Observation gated. Progression from one phase to the next is authorized by measured consumer behavior, specifically field level read telemetry, rather than by elapsed time or by assertion from a change ticket.

B. Phases

Let σ denote the current schema and σ' the target. The protocol constructs an intermediate σ^+ that is a structural superset of both.

Phase 1, Expand. Register σ^+ , adding every symbol required by σ' as optional with an explicit default. This transition is backward and forward compatible by construction, so it is accepted under full transitive compatibility and requires no consumer action.

Phase 2, Dual populate. Deploy producers that write both representations. The derivation of the new representation from the old is expressed as a pure function f registered alongside the schema, so that it can be applied consistently by backfill jobs and by the projection layer described in Section V. Consumers remain untouched and continue reading the old fields.

Phase 3, Migrate. Roll consumers to bind to the new representation. Because both are populated, consumers at old and new versions coexist correctly and the roll may proceed at any pace, including one application per week. For stateful consumers subject to class B6, this phase incorporates the state rebuild of Algorithm 1.

Phase 3 carries a precondition that is easy to miss and unsound to omit. Records written before Phase 2 do not contain the new representation. Schema resolution supplies the reader's default for the missing field, so those records remain decodable, but a default is not a correct value: for a rename, every pre-expand record decodes to the default rather than to the identifier actually written. A consumer that binds to the new representation while such records remain in retention therefore reads wrong data without any error being raised. Before a consumer may be migrated, one of three conditions must hold: every pre-expand record it will read is rewritten through the projection layer with the registered derivation f ; the consumer reads from a shadow topic into which history has been replayed under f ; or all pre-expand records have aged beyond δ and the consumer has no need to seek behind that point. The third is the cheapest where it applies, and it holds at no extra cost whenever the change has already

rested in Phases 1 and 2 for at least δ before any consumer is rolled. It cannot be discharged by the Phase 4 gate, which comes later: a precondition on migration is not satisfied by a wait that follows it.

Phase 4, Verify. Hold the expanded state for at least one full retention period δ plus one lease time to live T_l , while field level read telemetry accumulates. The deprecated symbols must show zero reads across the entire window before progression is authorized.

Phase 5, Contract. Register σ' , removing the deprecated symbols. Only now do records lacking the old representation begin to age into the retention window, and by construction no reader remains that requires them.

Algorithm 1 State Rebuild Under Key Redefinition (Class B6)

Require: topic t with partitions P , old key k , new key k' ,
mapping $\mu: k \rightarrow k'$, state horizon h , lag tolerance ε ,
divergence threshold θ_d

Ensure: state rekeyed to k' , no gap and no duplicate delivery

```

1:  $\tau_0 \leftarrow \text{SnapshotWallClockTime}()$ 
2:  $\omega_0 \leftarrow \text{SnapshotOffsets}(t) \triangleright$  per partition, taken with  $\tau_0$ 
3: for all  $p \in P$  do
  4:  $s_0[p] \leftarrow \text{OffsetForTimestamp}(p, \tau_0 - h)$ 
5: end for
6:  $t_s \leftarrow \text{CreateShadowTopic}(t, \text{partitioner}(k'))$ 
7:  $\text{ReplayRange}(t, [s_0, \omega_0], \text{into } t_s, \text{transform } \mu)$ 
8:  $\text{StartMirror}(t \rightarrow t_s, \text{from } \omega_0, \text{transform } \mu)$ 
9:  $J' \leftarrow \text{DeployShadowJob}(t_s, \text{state keyed by } k')$ 
10: while  $\text{Lag}(J') > \varepsilon$  do
  11: Wait
12: end while
13:  $d \leftarrow \text{Divergence}(\text{output}(J), \text{output}(J'))$ 
14: if  $d > \theta_d$  then
  15: Abort  $\triangleright J$  untouched; no impact
16: end if
17:  $\text{SwapReaders}(J \rightarrow J') \triangleright$  atomic at the serving layer
18:  $\text{KeepHot}(J) \triangleright$  consumes, does not serve
19:  $\text{Decommission}(J, \text{mirror}) \triangleright$  after rollback window

```

The essential property of Algorithm 1 is that the incumbent job J is never stopped, never reconfigured, and never has its state mutated. It continues serving throughout, and the swap at line 17 is a change of reader binding rather than a restart. If divergence exceeds the threshold θ_d , the abort at line 15 costs only the shadow resources.

The ordering of lines 1 to 8 is what makes the rebuild exact. Snapshotting ω_0 before any mirroring begins fixes a single per partition boundary; replay covers the half open interval $[s_0, \omega_0)$ and the live mirror attaches at ω_0 , so no record is carried by both paths. Starting the mirror first, then snapshotting, would admit every record produced in between along both routes, and for a non idempotent aggregation the rebuilt state would double count exactly those records. The lower bound s_0 is resolved per partition by timestamp lookup from the wall clock time τ_0 captured alongside the offset snapshot, rather than by arithmetic on offsets, since the state horizon h is a duration and offsets are not a time base. Nor can the lookup be anchored at $\omega_0[p]$ itself: an end offset names the next record to be written, not one that exists, so a timestamp query there is undefined.

C. Rollback After the Swap

Retaining J idle after the swap would not preserve reversibility. An idle job stops advancing while J' continues, so its state begins to diverge from the stream the moment it stops consuming, and a later swap back would restore an operator whose state is stale by the length of the rollback window. Line 18 therefore keeps J hot: it continues to consume and to update state, and only its serving binding is withdrawn. Rollback is then symmetric with the

forward swap and equally immediate. The cost is that both jobs run concurrently for the window, doubling the operator's footprint for that period. Where that is unacceptable, J may be retired at the swap and rollback treated as a fresh rebuild from a retained checkpoint: recoverable, but no longer immediate. Immediate reversibility is claimed only under the hot standby.

D. Correctness

Proposition 3: (Wire safety under contraction) Assume (i) every transition is live compatible in the sense of Definition 2; (ii) read telemetry observes every field access performed by every leaseholding consumer; and (iii) any consumer bound to symbol s accesses s at least once within the observation window. If Phase 5 is gated on zero observed reads of the deprecated symbols across a window of at least $\delta + T_1$, then no consumer observes an unresolvable record.

Proof sketch: Phases 1 and 2 are purely additive, so $W^\delta(t)$ grows only by schemas that are supersets of their predecessors and $R(t)$ is unchanged; wire safety is preserved. In Phase 3, $R(t)$ acquires readers of σ^+ , which resolve every member of $W^\delta(t)$ because σ^+ supplies defaults for all symbols absent from earlier writers. Phase 5 removes symbol s from writers. A reader could fail only if some $r \in R(t)$ binds to s . The gate establishes that no read of s was observed over a window of at least $\delta + T_1$; by assumption (ii) no read by a leaseholder went unobserved, by assumption (iii) a live reader bound to s would have produced one, and by Section V-D any group silent for T_1 has forfeited its claim. Hence no entitled reader binds to s .

Proposition 3 should be read narrowly. It establishes decodability on S1 and nothing more. It does not establish that resolved values are correct, which on S2 is the business of Algorithm 1, on S3 of Section V-E, on S4 of the tagging discipline, and for pre-expand records of the Phase 3 precondition. Decodability is the surface a registry can police; treating it as correctness is the conflation this paper argues against.

Assumptions (ii) and (iii) carry the argument and neither is free: (ii) fails if a consumer reads through a generic record path that never resolves field names, (iii) if it touches the field only on an infrequent branch. Section VII examines both. Residual risk is therefore concentrated in read telemetry, which motivates the instrumentation described next.

V. REFERENCE ARCHITECTURE

Figure 1 shows the four components the protocol requires. What follows specifies the obligations of each component. It describes a design, not a deployed system: no implementation underlies this paper, and the cost model of Section VI is derived rather than measured for that reason.

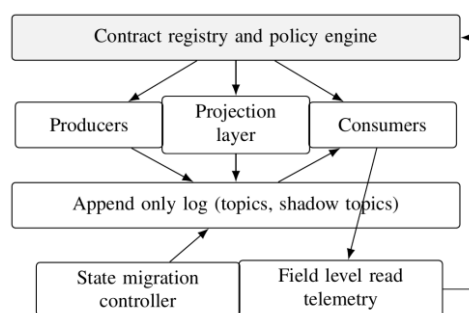


Fig. 1. Reference architecture. The registry is authoritative for schemas, derivations, and phase state; read telemetry closes the loop that gates contraction.

A. Contract Registry and Policy Engine

The registry must extend a conventional schema registry in three ways. It stores the derivation functions f that relate old and new representations, so that producers, backfill jobs, and the projection layer share one definition. It records the phase of every in flight change as first class state, which makes the migration auditable and prevents two overlapping changes to the same field. It enforces policy at registration time, admitting a transition only if it is live compatible in the sense of Definition 2, which requires the registry to know both the writer schemas still within retention and the reader schemas currently under lease. No stock compatibility mode expresses this, so the check is an extension rather than a configuration.

Class B7 needs a different mechanism. A semantic redefinition presents to the registry as an unchanged field: same name, same type, same position. No structural check detects it, so the registry cannot reject it on inspection. What it can do is make semantics part of the registered contract. Each field carries a semantic tag covering unit,

timezone, and enumeration domain; modifying a tag is then a contract change even though the field is untouched, and is rejected unless accompanied by a new field name or a tag version. This makes B7 checkable only for authors who declare the change. It is a discipline, not a guarantee.

Policy enforcement at registration is essential. If the registry is advisory, the discipline decays under delivery pressure, which is the mechanism by which API compatibility erodes in practice [13].

B. Schema Aware Projection Layer

The projection layer is a stateless transformation applied between the log and the consumer, materializing the reader's expected view from whatever representation is present. It converts upgrading n consumers into configuring one projection per consumer, and lets a consumer that cannot be modified, such as a vendor product or an unmaintained internal service, pass through a migration with no change to its binary.

The layer is deliberately restricted to total, side effect free functions over a single record. Anything requiring joins or aggregation belongs in the processing topology, not in the projection.

C. State Migration Controller

The controller is responsible for Algorithm 1: shadow topic lifecycle, replay from the state horizon, lag monitoring, and the divergence comparison gating the reader swap. Divergence is computed over a sampled key space rather than the full output, bounding comparison cost while retaining detection power for systematic errors, which are the ones that matter.

D. Consumer Leases

Definition 2 quantifies over $L(\tau)$, the reader schemas under live lease, and the protocol is only as sound as that set is well defined. An open consumer population cannot be enumerated by observation: the longest restart interval seen so far is a fact about the past, not a bound on the future, and a consumer dormant for longer than any previously observed gap can always return.

The registry therefore issues leases rather than inferring liveness. A consumer group registers the reader schema it binds to and renews by heartbeat; a lease not renewed within a fixed time to live T_1 expires, and with it the group's claim on the schemas it held. The guarantee is correspondingly explicit: a group that heartbeats is assured its reader schema will resolve every record it receives, and a group absent for longer than T_1 forfeits that assurance and must reconcile against the current schema when it returns.

This converts an unbounded obligation into a bounded one, and replaces the observational quantity R of Section VI with a policy parameter the operator chooses. The risk is relocated rather than removed: a group that consumes but fails to heartbeat is treated as absent. The registry can cross reference leases against broker side consumer group metadata and refuse to expire a lease for a group with a live offset commit.

E. Sink Contract Migration

Surface S3 requires its own sequence, since a change that is live compatible on the wire may still be rejected at commit or silently coerced by a downstream table [27]. The projection layer does not help here: it materializes a consumer's expected input view, whereas the sink constrains that consumer's output.

The sequence mirrors the wire protocol. New columns are added as nullable, which transactional table formats accept as a metadata operation. The consumer then dual writes, populating old and new columns from the same record. Historical rows are backfilled by applying the registered derivation f where downstream readers query behind the change point, and skipped where they do not. Sink side validation over the backfilled range, comparing derived against original under f , is the analogue of the divergence check in Algorithm 1. Views and queries are then cut over, and only afterwards, gated on the sink's own read telemetry, are the deprecated columns dropped.

Two asymmetries with the wire protocol are worth noting. A table has no retention analogue, so nothing ages out and the contraction gate cannot be satisfied by waiting; it must be satisfied by evidence of nonuse. And backfill cost scales with table size rather than with the retention window, so for large tables the dominant expense of a class B1, B3, or B5 change is on the sink rather than on the topic.

F. Field Level Read Telemetry

Because contraction is gated on evidence of nonuse, read telemetry accuracy determines the protocol's residual risk. Instrumentation is placed at the deserialization boundary, emitting a counter per (consumer group, schema, field) tuple on first access within a record. Sampling is applied per consumer group rather than per record, so that a low volume consumer is never sampled out and thereby misclassified as dormant. This asymmetry is deliberate: the cost of over retaining a field is storage, while the cost of contracting a field that is still read is an outage.

VI. COST MODEL

The protocol trades downtime for a bounded, temporary increase in record size, storage, and processing cost. Rather than report those costs for one configuration, this section derives them from deployment parameters, so the trade can be evaluated for a specific system. Section VI-G states what the model does not settle.

A. Parameters

Table II lists the parameters. Each is a property a team can read off an existing deployment.

B. Record Size and Storage

Dual population widens every record by w , so relative growth is w/r and does not depend on the event rate. The logical volume a topic retains is $V = \lambda r \delta$, and the additional logical volume carried during the expanded window is

$$\Delta V = \lambda w \delta \quad (1)$$

or $\rho \lambda w \delta$ once replicated. Two consequences follow. The overhead tracks the width of the changed field relative to the record rather than its absolute size, so widening one scalar in a wide record is close to free while duplicating a large nested structure is not. And since ΔV is linear in δ , retention is the dominant lever: halving retention halves the storage cost of every change.

C. State Rebuild

Class B6 is the only class that requires state to be rebuilt. The shadow job of Algorithm 1 must drain a backlog of $h\lambda$ records while the live tail continues to arrive at rate λ . With replay throughput μ the backlog drains at $\mu - \lambda$, giving

$$T = h\lambda / (\mu - \lambda) = h / (k - 1), \quad k = \mu/\lambda > 1 \quad (2)$$

Three properties follow. Rebuild converges only when $\mu > \lambda$: a shadow job that cannot outpace the live stream never catches up. When replay throughput scales with provisioning so that k is fixed, rebuild time is independent of the absolute event rate, so a pipeline at 10^6 events per second rebuilds no more slowly than one at 10^4 given proportionate resources. And the term is linear in h , which is why Algorithm 1 parameterizes replay depth on the state horizon and not on retention: at $k = 5$ a one hour window rebuilds in fifteen minutes, a twenty four hour window takes six hours, and replaying a seven day retention period would take forty two hours.

D. Duration of the Expanded Window

Phase 4 holds the expanded state for one full retention period plus one lease time to live, after which any consumer that has not heartbeated has forfeited its claim (Section V-D). The verification gate therefore cannot run shorter than $\delta + T_l$. The expanded window as a whole is longer still, since it also spans dual population and the consumer roll, and longer again where the Phase 3 precondition is discharged by waiting rather than by projection.

What matters operationally is that engineering effort does not enter the gate at all: additional staff shorten Phase 3 but cannot shorten Phase 4. Verification gate duration is set by retention and lease policy, both of which the operator chooses; shortening either shortens the mandatory verification period. The same property lets a migration pause indefinitely at any intermediate state. Figure 2 shows an illustrative schedule.

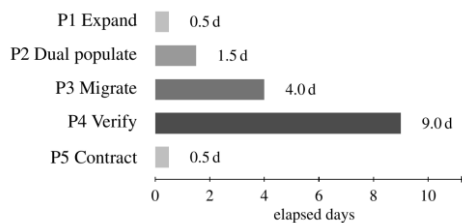


Fig. 2. Illustrative phase schedule for a class B6 key redefinition at $\delta = 7$ d, $T_l = 1$ d. The verification gate is bounded below by $\delta + T_l$.

E. Dual Population Versus Shadow Topic

Where w is large relative to r , dual population in the primary topic becomes expensive and the shadow topic of Algorithm 1 is preferable: it carries whole records of size $r + w$ but may be given a much shorter retention δ_s . The shadow is cheaper when

$$\delta_s(r + w) < \delta w, \quad \text{equivalently} \quad \delta_s < \delta \cdot w/(r + w) \quad (3)$$

For a change adding 15 percent to record width against a seven day primary retention, the shadow wins whenever it can be retained for less than roughly 22 hours. For the one hour state horizon used in the worked example, that threshold exceeds the rebuild time at any k above two.

F. Worked Example

Table III instantiates the model across three orders of magnitude in event rate, holding $r = 1.4$ KiB, $w/r = 0.15$, $\delta = 7$ days, $\rho = 3$, $h = 1$ hour, $k = 5$, and $T_l = 1$ day. These parameter values are illustrative inputs, not measurements.

Storage scales linearly with rate. The quantities determining whether a migration is operationally tolerable, namely relative record growth, rebuild time, and the verification gate, do not.

G. What the Model Does Not Settle

Three quantities are implementation dependent and cannot be derived from the parameters above.

The per record cost of the projection layer, and hence its effect on tail latency, depends on the serialization framework, the derivation function, and deployment headroom. This is the quantity most likely to decide whether the protocol is acceptable on a latency sensitive pipeline. The achievable replay throughput μ , and therefore k , depends on provisioning and on whether replay contends with live traffic; Equation (2) is only as useful as the estimate of k a team can supply. And the completeness of field level read telemetry, required by assumptions (ii) and (iii) of Proposition 3, is a property of the instrumentation rather than of the protocol, with failure modes discussed in Section VII.

Establishing these three quantities on a running deployment is the natural next step, and would turn this model into a validated one.

TABLE II
COST MODEL PARAMETERS

<i>Symbol</i>	<i>Meaning</i>
λ	<i>event rate, events per second</i>
r	<i>mean record size</i>
w	<i>width added to each record by dual population</i>
δ	<i>topic retention period</i>
ρ	<i>replication factor</i>
h	<i>minimum replay horizon sufficient to reconstruct the widest stateful operator, including any lateness allowance</i>
μ	<i>replay throughput of the shadow job</i>
k	<i>replay speedup ratio, μ/λ</i>
T_l	<i>consumer lease time to live</i>
δ_s	<i>retention of a shadow topic</i>

TABLE III
MODEL INSTANTIATED ACROSS THREE EVENT RATES

<i>Quantity</i>	$\lambda=10^4$	$\lambda=10^5$	$\lambda=10^6$
<i>Logical retained volume (TiB)</i>	7.9	78.9	788.6
<i>Added logical volume (TiB)</i>	1.2	11.8	118.3
<i>Added replicated volume (TiB)</i>	3.5	35.5	354.9
<i>Relative record growth</i>	15%	15%	15%
<i>Rebuild, $h=1$ h, $k=5$</i>	15 min	15 min	15 min
<i>Verification gate, minimum</i>	8 d	8 d	8 d

VII. LIMITATIONS AND THREATS TO VALIDITY

Four limitations bear stating.

The cost model of Section VI is derived rather than measured, and has not been validated against a running deployment. Its storage and duration terms follow from the definition of the protocol and we expect them to hold, but the projection layer's per record cost, the achievable replay ratio k , and the completeness of read telemetry are all implementation dependent, and a deployment could differ from the model in any of them.

The protocol cannot be deployed on a stock schema registry. Live compatibility requires the registry to know which writer schemas remain within retention and which reader schemas are under lease, and no conventional compatibility mode expresses either. A team unwilling to extend its registry can adopt the taxonomy, the phase

structure, and the sink sequence, but will find contraction unreachable for a field that lacked a default in an earlier schema against which compatibility is checked, and must carry the expanded representation indefinitely.

The correctness argument for contraction depends on read telemetry being complete. A consumer that reads a field through reflection, through a generic record path that does not resolve field names, or through a code path exercised only on an infrequent branch may be misclassified as dormant even while holding a lease. Leases bound which consumers are entitled to a guarantee, but they do not make the telemetry that gates contraction any more complete. Systems requiring a stronger guarantee should combine telemetry with static analysis of consumer binaries.

Class B7 is handled by declaration rather than by detection, and this is the weakest part of the design. A semantic redefinition is structurally invisible, so the registry depends entirely on the author tagging the change; an undeclared redefinition passes every check the platform performs and reaches consumers exactly as it would without the protocol. The tag discipline narrows the exposure without closing it, and closing it would require either a formal semantic annotation vocabulary or organizational discipline that no platform can enforce alone.

Finally, the protocol assumes producers can be modified. Where the producer is an external party, dual population must be relocated into an ingestion boundary stage, which preserves correctness but shifts the cost and lengthens the derivation path.

VIII. CONCLUSION

Breaking schema change in streaming pipelines is conventionally treated as an event requiring downtime. We have argued that this treatment reflects an incomplete model rather than a genuine constraint. Registry compatibility reasons about the wire encoding alone, while a consumer binds to a topic across four surfaces: encoding, operator state, sink contract, and semantics. Once all four are made explicit, breaking changes admit factorization into sequences of individually compatible transitions.

The expand, migrate, contract protocol performs that factorization, and the reference architecture combines compatibility policy, schema aware projection, state migration, consumer liveness tracking, sink migration, and field level telemetry to make those transitions operationally enforceable. The cost model shows the price of the trade to be a temporary increase in record width and retained volume, both proportional to the width of the changed field, together with a rebuild term that depends on the operator's state horizon and the ratio of replay to live throughput rather than on the event rate itself.

The result that we regard as most consequential is not the elimination of downtime but the elimination of the coordinated deadline. When every intermediate state is a valid steady state, a migration can pause indefinitely without harm, and the pressure that converts a routine change into an incident is removed. One negative result deserves emphasis alongside the protocol. The transitive compatibility modes that registries offer, and present as the safe default, cannot admit contraction of any field that lacked a default in an earlier registered version. The obstacle is not conservatism but quantification: those modes range over every version ever registered, a set that never ages, while safety depends only on the writers still retained and the readers still live. Naming the smaller set is what makes contraction reachable. Two directions follow. The model of Section VI should be validated on a running deployment, in particular its projection cost and replay ratio terms. And semantic contracts deserve the rigor now applied to structural ones, since class B7 remains handled by declaration rather than detection.

REFERENCES

- [1] J. F. Roddick, "A survey of schema versioning issues for database systems," *Information and Software Technology*, vol. 37, no. 7, pp. 383-393, 1995.
- [2] S. W. Ambler and P. J. Sadalage, *Refactoring Databases: Evolutionary Database Design*. Boston, MA, USA: Addison-Wesley, 2006.
- [3] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: The PRISM workbench," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 761-772, Aug. 2008.
- [4] K. Varda, "Protocol buffers: Google's data interchange format," Google Open Source Blog, Jul. 2008.
- [5] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. 6th Int. Workshop on Networking Meets Databases (NetDB)*, Athens, Greece, 2011, pp. 1-7.
- [6] T. Akidau et al., "MillWheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1033-1044, Aug. 2013.
- [7] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek, "Online, asynchronous schema change in F1," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1045-1056, Aug. 2013.
- [8] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," *The VLDB Journal*, vol. 22, no. 1, pp. 73-98, Feb. 2013.

- [9] D. Qiu, B. Li, and Z. Su, "An empirical analysis of the co-evolution of schema and code in database applications," in *Proc. 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, Saint Petersburg, Russia, 2013, pp. 125-135.
- [10] J. Kreps, "The log: What every software engineer should know about real-time data's unifying abstraction," LinkedIn Engineering Blog, Dec. 2013.
- [11] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447-1463, Jun. 2014.
- [12] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. Weber, "Understanding database schema evolution: A case study," *Science of Computer Programming*, vol. 97, pp. 113-121, Jan. 2015.
- [13] T. Espinha, A. Zaidman, and H.-G. Gross, "Web API growing pains: Loosely coupled yet strongly tied," *Journal of Systems and Software*, vol. 100, pp. 27-43, Feb. 2015.
- [14] 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 Endowment*, vol. 8, no. 12, pp. 1792-1803, Aug. 2015.
- [15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28-38, Dec. 2015.
- [16] G. Wang et al., "Building a replicated logging system with Apache Kafka," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1654-1655, Aug. 2015.
- [17] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Real-Time Data Systems*. Shelter Island, NY, USA: Manning, 2015.
- [18] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA, USA: O'Reilly Media, 2015.
- [19] M. Kleppmann, *Making Sense of Stream Processing*. Sebastopol, CA, USA: O'Reilly Media, 2016.
- [20] P. Vassiliadis, A. V. Zarras, and I. Skoulis, "Gravitating to rigidity: Patterns of schema evolution and its absence in the lives of tables," *Information Systems*, vol. 63, pp. 24-46, Jan. 2017.
- [21] S. A. Noghabi et al., "Samza: Stateful scalable stream processing at LinkedIn," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1634-1645, Aug. 2017.
- [22] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Sebastopol, CA, USA: O'Reilly Media, 2017.
- [23] M. Armbrust et al., "Structured Streaming: A declarative API for real-time applications in Apache Spark," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Houston, TX, USA, 2018, pp. 601-613.
- [24] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proc. 13th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, USA, 2018, pp. 783-798.
- [25] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, "An empirical study of GraphQL schemas," in *Proc. Int. Conf. on Service-Oriented Computing (ICSOC)*, Toulouse, France, 2019, pp. 3-19.
- [26] Apache Software Foundation, "Apache Avro 1.10.0 specification," 2020. [Online]. Available: <https://avro.apache.org/docs/1.10.0/spec.html>
- [27] M. Armbrust et al., "Delta Lake: High-performance ACID table storage over cloud object stores," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3411-3424, Aug. 2020.
- [28] Confluent Inc., "Schema evolution and compatibility," Confluent Platform 5.4 documentation, 2020. [Online]. Available: <https://docs.confluent.io/5.4.2/schema-registry/avro.html>