SDRF: A Unified CDC Abstraction for Cloud-Native SaaS Data Replication

Shashank Sharma
Amazon Web Services (AWS)
Kumar.shashanksharma@gmail.com

Mudit Goyal
Amazon Web Services (AWS)
goyal.mudit0491@gmail.com

Shashi Shekhar
Amazon Web Services (AWS)
shashishekhar284@gmail.com

Abstract*—The proliferation of Software-as-a-Service (SaaS) applications has created significant data integration complexity. Connecting N SaaS sources (e.g., Salesforce) to M analytical targets (e.g., data lakes, data warehouses) traditionally requires N×M bespoke integrations. This paper presents the SaaS Data Replication Format (SDRF), a JSON-based Change Data Capture (CDC) envelope that decouples SaaS source connectors from downstream targets through a common intermediate format on object storage. Unlike database CDC formats that rely on transaction logs, SaaS systems expose only timestamp-based modification fields and provider-specific deletion indicators, requiring a different approach. This paper makes two primary contributions and two supporting design points. The primary contributions are: (i) a (x=epoch_ms, y=event_count) coordinate mapping that translates timestamp-based SaaS changes onto a monotonically-increasing total order, for which we provide informal correctness arguments for monotonicity, restart determinism, and bounded staleness; and (ii) a table-level checkpoint-isolation protocol with a linked checkpoint chain, enabling per-table parallel extraction with independent resumability. The supporting design points are: the Link Manifest — a manifest-based integration contract that decouples source and target implementations — and a compare-and-detect schema-evolution procedure. A production deployment of SDRF powers Salesforce data ingestion into Apache Iceberg on S3, and we report measurements from that deployment: a controlled 10 M-row Salesforce ingestion completes in 6 min 20 s.

Keywords—SaaS data replication, change data capture, data lake, cloud-native ETL, SDRF, schema evolution, checkpoint-based resumability

I. INTRODUCTION

A. Background

The rise of SaaS applications as enterprise systems of record has created a need for reliable, low-latency data replication into analytical targets. Organizations need this operational data available in data lakes for analytics, compliance, and machine-learning workloads.

Database replication commonly uses transaction-log-based CDC, where write-ahead logs or binary logs provide an ordered, complete record of all changes [1]. SaaS systems present a categorically different challenge: they expose no binary logs or transaction logs. Instead, changes are tracked through timestamp fields (e.g., `SystemModstamp` in Salesforce), deletion flags (e.g., `IsDeleted`), and provider-specific API constructs.

B. Problem Statement

Each SaaS provider uses proprietary data formats, unique change-tracking mechanisms, and provider-specific deletion semantics. Connecting N SaaS sources to M analytical targets traditionally requires N×M custom integrations. Beyond this integration count, SaaS systems present four concrete technical obstacles for CDC:

1. *Sub-second collisions.* Change tracking relies on timestamp fields that may have millisecond-level granularity collisions.
2. *Provider-specific deletes.* Delete detection requires provider-specific mechanisms (e.g., Salesforce `getDeleted` API for records deleted in the past 15 days).
3. *Silent schema changes.* Schema changes occur without notification events emitted by the SaaS platform.
4. *Clock-skew and pagination.* `SystemModstamp` is server-assigned; extractor pagination and SaaS clock corrections can produce out-of-order arrival at the connector.

C. Contributions

This paper presents SDRF, a CDC specification designed from first principles for SaaS sources. Distinguishing primary contributions from supporting design points:

Primary contributions.

- **C1**. A coordinate mapping (x, y) where x is the event's server-assigned modification timestamp (e.g., Salesforce SystemModstamp) in epoch milliseconds (UTC) and y is a connector-assigned, deterministically recoverable sequence number within the millisecond window. We give informal proofs of monotonicity (Lemma 1), restart determinism (Lemma 2), and

bounded staleness under a safety offset (Proposition 3).

- **C2.** A table-level checkpoint-isolation protocol with a linked checkpoint chain, enabling independent per-table extraction, parallel fault recovery, and O(1) checkpoint location via `LATEST_*` pointer files.

Supporting design points.

- **D1.** A Link Manifest — a manifest-based contract pinning storage layout, sharding policy, and protocol version so that source and target implementations can evolve independently. This pattern is not novel; the contribution is the specific field set validated against an Iceberg target.

- **D2**. A compare-and-detect schema-evolution procedure emitting ADD_COLUMN, DELETE_COLUMN, and UPDATE_COLUMN_DATATYPE events when the SaaS schema drifts between cycles (§IV.F).

D. Scope

SDRF is source- and target-agnostic by design. This paper uses Salesforce as the evaluation source and Apache Iceberg as the evaluation target; the specification applies equally to other SaaS providers (ServiceNow, Zendesk, SAP, Workday) and other targets (Redshift, Snowflake). External validity is discussed in §VII.B.

II. Related Work

A. Database CDC Systems

Transaction-log-based CDC is mature. Debezium [1] captures row-level changes from database logs (MySQL binlog, PostgreSQL WAL) and emits structured change events via Apache Kafka [2]. Since 2021, Debezium supports incremental snapshots and ad-hoc snapshots via signal tables [3], partially addressing the log-less-source case but still assuming a relational source with a primary key. Kleppmann [4] provides an extensive treatment of CDC patterns including log compaction and event sourcing. SDRF targets sources that expose neither a log nor a signal table.

B. SaaS Data Integration Platforms

Airbyte defines an open protocol with structured JSON messages (RECORD, STATE, CATALOG) for source-destination communication. The Airbyte Protocol does not include first-class protocol-level fields for CDC operation type or before/after images; CDC metadata is instead appended as connector-specific columns, and INSERT/UPDATE operations remain indistinguishable in the RECORD message. Singer defines a tap-target protocol with SCHEMA, RECORD, and STATE messages; Meltano extends this specification via the Meltano SDK. Neither protocol defines a formally monotonic coordinate space with correctness arguments. Commercial cloud providers offer managed SaaS connectors but do not publish an open envelope specification.

C. Target-Side CDC Envelopes

Delta Change Data Feed and Iceberg's changelog view [5] are target-side CDC envelopes: they expose change data *produced by* a Delta/Iceberg table. SDRF is the upstream envelope; these are downstream consumer formats. SDRF's DML-event schema is designed so a target adapter can translate SDRF into a Delta CDF or Iceberg changelog row without schema surgery.

D. Open Table Formats

Delta Lake [6], Apache Hudi [7], and Apache Iceberg [5] are CDC **consumers** — they assume change data arrives in a structured format. None standardize the upstream CDC envelope for SaaS sources. SDRF fills this gap.

E. Data-Integration Theory

The canonical-intermediate-format approach to reducing N×M integration complexity is well-established [8], [9]. Curino et al. [10] address automated schema evolution in databases, providing a taxonomy of schema modification operators applicable to SaaS schema changes. Vassiliadis [11] surveys ETL technology.

F. Delta vs. Closest Prior Work

Table I summarizes the feature delta against the three closest open systems.

Feature	**Airbyte Protocol**	**Singer/Meltano**	**Debezium [1]**	**SDRF (this paper)**
Open specification	Yes	Yes	Yes	Yes
Designed for log-less SaaS sources	Yes	Yes	Partial (DB-shaped sources)	Yes
Explicit CDC op type (INSERT/UPDATE/DELETE)	No	No	Yes	Yes
Before/after images	No	No	Yes (DB sources)	Yes (DELETE before, others after)
Documented monotonic coordinate	Per-stream cursor	Per-stream cursor	Yes (binlog)	**Yes, with informal correctness argument (§IV.C)**

Per-table checkpoint isolation	Per-stream STATE	Per-stream STATE	No	**Per-table, with linked chain (§IV.D)**
DDL events in envelope	No	No	Yes	Yes
Target-agnostic architecture[1]	Yes	Yes	Yes	Yes

Table I: Feature comparison against closest prior art.

[1]SDRF is evaluated on Iceberg only; the architecture supports arbitrary targets via the Link Manifest contract.

The two rows in bold correspond to SDRF's primary contributions C1 and C2. The remaining rows are feature parity rather than novelty claims.

III. Architecture

SDRF implements a three-component architecture that enforces strict separation between source and target systems through an intermediate storage layer.

1. **Source Connector.** Extracts from the SaaS provider, transforms to SDRF, writes to intermediate storage. Encapsulates all provider-specific logic: API interaction, change-indicator interpretation, and timestamp-to-coordinate mapping.

2. **Intermediate Storage.** Durable object storage (e.g., S3-compatible object storage) holding SDRF CDC events, snapshots, checkpoint files, and the Link Manifest. Serves as the decoupling layer.

3. **Target System.** Reads SDRF data from intermediate storage and ingests into the destination. Encapsulates target-specific ingestion logic.

Adding a new SaaS source requires only implementing a source connector; it immediately works with all existing targets. Adding a new target requires only implementing a consumer; it immediately works with all existing sources. Decoupling is enforced by the Link Manifest (§IV.E) and the deterministic storage layout (§IV.B). SDRF is the specification; a production system implements it for Salesforce, and the measurements in §V are drawn from that deployment.

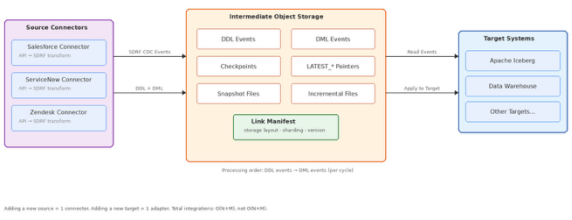

Fig. 1. SDRF three-component architecture. Source Connectors extract from SaaS providers and emit SDRF-formatted CDC events (DDL and DML) to intermediate object storage (S3-compatible object storage). The Link Manifest pins the storage contract. Target Systems (e.g., Iceberg) consume events independently. DDL events are processed before DML events in each cycle.

IV. SDRF Specification

A. Format Overview

SDRF has four foundational design properties:

1. **Schema-first.** Strongly typed with per-field type annotations in JSON Schema, enabling validation at source and target.

2. **Extensible.** Provider-specific metadata fields may be included without breaking consumers.

3. **Versioned.** Backward-compatible evolution; the `Version` field in the Link Manifest coordinates upgrades.

4. **Efficient.** Optimized for streaming and batch processing using partitioned storage with shard-level parallelism.

B. Storage Directory Structure

SDRF data is organized in a deterministic hierarchy rooted at an integration-specific prefix.

Fig. 2. SDRF storage directory structure.

```
s3://replication-
bucket/<integration_id>/
 LINK_MANIFEST.json
 checkpoints/

LATEST_SNAPSHOT_LOAD_CHECKPOINT.j
son

LATEST_INCREMENTAL_LOAD_CHECKPOIN
T.json

snapshot_load_checkpoint_<end_coo
rd>.json

inc_load_checkpoint_<start>_<end>
.json

snapshot_load_files/<end_timestam
p>/<database>/<table>/
 data/part-0-xxx.parquet
 schema/part-00000
```

```
incremental_load_files/<end_timestamp>/<database>/<table>/
 shard_<shard_id>/part-xxx.json
 shard_ddl/part-xxx.json
```

This structure lets the target: (i) locate the Link Manifest once at integration creation; (ii) poll `LATEST_*` pointers without `LIST` operations; (iii) process DDL and DML events independently, since DDL must be applied before DML in the same cycle.

C. Coordinate Mapping

Log-based CDC tracks position with a `(transaction_id, event_id)` tuple. SaaS systems have no equivalent. SDRF defines:

- x = SystemModstamp in epoch milliseconds, UTC. Rationale: Salesforce datetime fields are typed as millisecond-precision, though API responses may return second-level granularity (milliseconds zeroed). The coordinate space uses epoch milliseconds uniformly; the tiebreaker (y) handles collisions within the same x window regardless of effective precision.
- `y` = a connector-assigned sequence number within the `x` window. `y` is assigned **deterministically** by extraction order (§IV.C.2).

The wire-format field names are `binlog-file-seq-num` (= `x`) and `binlog-file-offset` (= `y`), preserved for backward compatibility with log-based target adapters.

Tiebreaker. Within an `x` window (i.e., events sharing the same `SystemModstamp` in ms), events are ordered by `primary_key ASC`. The connector assigns `y` values `0, 1, 2, …` in this order. The tiebreaker must be consistent across SaaS-API calls (e.g., Salesforce `ORDER BY SystemModstamp ASC, Id ASC`) so that restart determinism (Lemma 2) holds.

Inclusive/exclusive semantics**.** Checkpoints record the **closed** interval `[start, end]`: `start_binlog_pos` is the coordinate of the first event emitted in the cycle, `end_binlog_pos` is the coordinate of the last event emitted, and `synced_binlog_pos == end_binlog_pos`. The next cycle's `start_binlog_pos` is the coordinate of the *actual first event of that cycle*, which is strictly greater than the previous `end_binlog_pos` under lex order; no derived "successor" coordinate is materialized. Each checkpoint also records pk_at_sync, the primary key of the last event at the synced coordinate, enabling the tiebreaker-aware restart query in Lemma 2.

C.1. Worked Example

Consider a CDC cycle spanning `x ∈ [10_000 ms, 50_000 ms]` containing 3 DDL changes (all at `x = 10_000`, with distinct primary keys that sort as `k_a < k_b < k_c`) and 15 DML events (the last of which lands at `x = 50_000`).

DDL checkpoint is written first:

```
DDL checkpoint
 start_binlog_pos = [10_000, 0] # first DDL event (key k_a)
 end_binlog_pos = [10_000, 2] # third DDL event (key k_c)
 synced_binlog_pos = [10_000, 2]
```

DML checkpoint follows. For this example we assume the 15 DML events occur across multiple distinct `x` values, ending with a single event at `x = 50_000`:

```
DML checkpoint
 previous_checkpoint = "checkpoints/inc_load_checkpoint_10000_0_10000_2.json"
 start_binlog_pos = [10_000, 3] # first DML event; same x = 10_000 as DDL
 # but y = 3 because DDLs occupied y = 0..2
 end_binlog_pos = [50_000, 0] # last DML event; only event at x = 50_000
 synced_binlog_pos = [50_000, 0]
```

Continuity: `start_binlog_pos` of the DML checkpoint (`[10_000, 3]`) is strictly greater under lex order than `end_binlog_pos` of the DDL checkpoint (`[10_000, 2]`), satisfying Lemma 1.

C.2. Correctness of the Coordinate Mapping

Let `E` be the stream of events emitted by the connector in extraction order, and let `coord(e)` be the `(x, y)` assigned to event `e`.

Lemma 1 (Monotonicity). For any two events `e_i, e_j` emitted in order (`i < j`), `coord(e_i) < coord(e_j)` under lexicographic order on `(x, y)`.

Proof sketch. The connector emits events sorted by SystemModstamp ascending (the SaaS API sort order we require — e.g., Salesforce ORDER BY SystemModstamp ASC). For events with equal x, the connector assigns y by extraction order starting at 0. Thus either $x_i < x_j$ (case 1) or $x_i = x_j \wedge y_i < y_j$ (case 2). Both cases yield coord(e_i) < coord(e_j) lexicographically. DDL events are emitted before DML events within the same x window, and their y

values form a contiguous sequence, so monotonicity holds across DDL-DML boundaries.

Lemma 2 (Restart determinism under a stable pre-crash result set). Let `R` be the sorted result set the connector obtained from the SaaS API before crashing mid-cycle. If no new records with `SystemModstamp == synced_binlog_pos.x` are committed to the SaaS system between crash and restart, then restart re-emits the uncommitted suffix of `R` with the same `(x, y)` values it would have received in an uninterrupted run.

Proof sketch. On restart the connector reads `synced_binlog_pos` and issues the query `SystemModstamp > synced_binlog_pos.x OR (SystemModstamp = synced_binlog_pos.x AND primary_key > synced_binlog_pos.pk_at_sync)`, where `pk_at_sync` is the tiebreaker primary key persisted alongside the checkpoint. Under the stable-result-set assumption the returned rows are exactly the suffix of `R` after `synced_binlog_pos`, in the same order, so `y` is assigned identically.

Remark (handling of within-window writes during a crash). If a new record with `SystemModstamp == synced_binlog_pos.x` is written between crash and restart, it will appear on replay with a `primary_key` that may sort *before* already-emitted events. The connector detects this via the "`x < synced_binlog_pos.x OR (x == synced_binlog_pos.x AND pk ≤ pk_at_sync)`" drop filter of §IV.C.3 and rejects the late record with a logged anomaly. The safety offset `Δ` (see Proposition 3) is calibrated so that this case is rare in practice.

Proposition 3 (Bounded staleness as operating contract, not theorem). SDRF operates under the assumption that, for each SaaS provider, there exists a visibility lag `Δ_P` such that any record whose `SystemModstamp` is `≤ t` is visible via the SaaS API by wall-clock `t + Δ_P`. Given this assumption, applying a cycle-cutoff policy `now() - Δ` with `Δ ≥ Δ_P` yields no missed events.

This is a contract, not a theorem: the premise is empirical and must be validated per source. We set Δ = 120 s for Salesforce, conservative relative to empirically observed visibility lag in our deployment.

C.3. Clock-Skew Handling

`x` is derived from the SaaS server's `SystemModstamp`, not the connector's wall clock. Connector wall-clock skew therefore does not break monotonicity. SaaS server clock corrections (NTP adjustments or other server-side clock corrections) can in principle produce a `SystemModstamp` less than a previously observed one; we treat such events as belonging to their assigned `x` window regardless of observation order. If the assigned `x` is less than `synced_binlog_pos.x`, the event is dropped with a logged anomaly.

D. Checkpoint File Structure

Two checkpoint types: Snapshot Load Checkpoints (written once after full-table replication) and **Incremental Load Checkpoints** (written after each CDC cycle).

A Snapshot Load Checkpoint records the export format, per-table data and schema file locations, and the `synced_binlog_pos` establishing the starting coordinate for subsequent incremental loads:

```
{
 "previous_checkpoint": "",
 "export_format": "Parquet",
 "table_statistics": [
 {
 "table_name": "Account",
 "table_schema":
"snapshot_load_files/.../schema/p
art-00000",
 "data_files":
"snapshot_load_files/.../data/"
 }
 ],
 "synced_binlog_pos":
[1706300646000, 15800]
}
```

An Incremental Load Checkpoint records per-table log-file paths, DML counts, and the binlog coordinate range. Assume the previous cycle ended at `[50_000, 14]`; this cycle extends from there up to `[100_000, 48]`:

```
{
 "previous_checkpoint":
"checkpoints/inc_load_checkpoint_
10000_3_50000_14.json",
 "table_statistics": [
 {
 "table_name": "Account",
 "log_files":
["incr.../Account/shard_1/part_1.
json"],
 "ddl_count": 3, "insert_count":
5,
 "update_count": 3,
"delete_count": 2
 }
```

```
 ],
 "start_binlog_pos": [51_234, 0],
 "end_binlog_pos": [100_000, 48],
 "synced_binlog_pos": [100_000,
48],
 "pk_at_sync": "001XX0000003ZZZ",
 "isDDL": false
}
```

Note that `start_binlog_pos = [51_234, 0]` is strictly greater than the previous `end_binlog_pos = [50_000, 14]` under lex order, satisfying Lemma 1; the exact `start` value is whatever the first event of this cycle's `x` happens to be, not a derived "successor" coordinate.

Key design elements: `previous_checkpoint` forms a linked list enabling chain traversal for recovery; `isDDL` distinguishes DDL checkpoints (processed on the leader) from DML checkpoints (processed on compute nodes); `table_statistics` provides per-table visibility without requiring the target to scan event files.

E. Link Manifest (supporting design point D1)

The Link Manifest is written once by the source during integration creation. It pins the source/target contract:

```
{
 "SourceType": "SaaS",
 "HashFunction": "ModSlice",
 "ShardingPolicy":
"TablePrimaryKeySharding",
 "NumShards": 32,
 "CheckPointLocation":
"checkpoints",
 "DataFileLocation":
"snapshot_load_files",
 "LatestCheckpointFile":

"checkpoints/LATEST_INCREMENTAL_L
OAD_CHECKPOINT",
 "LatestSnapshotCheckpoint":

"checkpoints/LATEST_SNAPSHOT_LOAD
_CHECKPOINT",
 "LogFileLocation":
"incremental_load_files",
 "TargetType": "Iceberg",
 "Version": "1.0"
}
```

Three sharding policies: NoSharding (all events in `shard_0`); PrimaryKeySharding (events distributed across N shards by primary-key hash); TablePrimaryKeySharding (table-level directory separation with primary-key sharding within each table).

`LatestCheckpointFile` and `LatestSnapshotCheckpoint` are pointer files allowing the target to resume consumption without `LIST` calls over potentially large checkpoint directories.

We do not claim the manifest pattern as novel; the contribution is the specific field set validated against an Iceberg target in §V.

F. Schema Evolution (supporting design point D2)

1. After each successful extraction, the source persists the current schema, keyed by integration and table.

2. At the start of each incremental cycle, the source retrieves the cached schema and the current SaaS-API schema, and diffs.

3. Detected changes are classified as `ADD_COLUMN`, `DELETE_COLUMN`, or `UPDATE_COLUMN_DATATYPE`, and are emitted as DDL events ahead of DML events in the same cycle.

Known limitations, inherent to the approach:

- Column renames are detected as `DELETE_COLUMN` + `ADD_COLUMN`. Column history is lost at the target.

- Column reordering is not detected; column order follows the SaaS API response ordering.

- Some connectors use static schema files without metadata APIs, preventing runtime schema discovery.

We report detection overhead in §V.D and discuss potential rename mitigations (column fingerprinting) as future work in §VIII.C.

G. DML Events

DML events carry two top-level fields: `metadata` (event context) and `payload` (before/after record images). Event-type semantics for SaaS sources:

- **INSERT.** before-image = `null`, after-image = full current record.

- **UPDATE.** before-image = `null`, after-image = full current record. SaaS APIs do not provide old field values. The target applies UPDATEs as full-record overwrites keyed on the primary key.

- **DELETE.** before-image = full record at deletion time, after-image = `null`.

Event type is determined from provider-specific indicators. Salesforce example: `IsDeleted = true` →

`DELETE`; otherwise `SystemModstamp > CreatedDate` → `UPDATE`; otherwise → `INSERT`. All SaaS connectors normalize to these three event types.

Example `INSERT`:


```
{
 "metadata": {
 "database-name": "salesforce",
 "table-name": "Account",
 "event-type": "INSERT",
 "primary-key-columns": ["Id"],
 "primary-keys":
["001XX0000003DHP"],
 "timestamp": 1675699798000,
 "binlog-file-seq-num":
1675699798000,
 "binlog-file-offset": 0
 },
 "payload": {
 "before-image": null,
 "after-image": {
 "Id": "001XX0000003DHP",
 "Name": "Acme Corporation",
 "Type": "Customer",
 "Industry": "Technology"
 }
 }
}
```


All timestamps are **epoch milliseconds, UTC.** `timestamp` and `binlog-file-seq-num` carry the same value; they are kept as separate fields to preserve log-based-CDC wire compatibility.

H. DDL Events

DDL events represent schema changes. Each carries a complete table-definition snapshot (not a delta) in both before- and after-images, enabling idempotent replay.

Supported DDL types: `CREATE_TABLE`, `DROP_TABLE`, `ADD_COLUMN`, `DELETE_COLUMN`, `UPDATE_COLUMN_DATATYPE`. Each event carries one schema change; a cycle with multiple schema changes produces multiple DDL events in detection order.

Example `ADD_COLUMN`:


```
{
 "metadata": {
 "container-name": "salesforce",
 "table-name": "Account",
 "event-type": "ADD_COLUMN",
 "column-name": "NewField__c",
 "timestamp": 1675699900000,
 "binlog-file-seq-num":
1675699900000,
 "binlog-file-offset": 0
 },
 "payload": {
 "before-image": { "table-
definition": {
 "columns": [
 {"Id": {"data-
type":"StringType","nullable":fal
se}},
 {"Name": {"data-
type":"StringType","nullable":tru
e}}
 ],
 "primary-key-columns": ["Id"]
 }},
 "after-image": { "table-
definition": {
 "columns": [
 {"Id": {"data-
type":"StringType","nullable":fal
se}},
 {"Name": {"data-
type":"StringType","nullable":tru
e}},
 {"NewField__c": {"data-
type":"StringType","nullable":tru
e}}
 ],
 "primary-key-columns": ["Id"]
 }}
 }
}
```


V. EVALUATION

We evaluate SDRF along three axes:

1. End-to-end ingestion throughput (§V.B): ingestion time on a controlled 10 M-row workload.
2. Multi-entity concurrency (§V.C): ingestion time with four concurrent Salesforce objects.
3. Scaling and schema detection (§V.D): shard-count scaling, schema-evolution-detection overhead.

A. Experimental Setup

Production deployment setup:

- **Source:** Salesforce Enterprise-edition account; a single custom object populated with 10 M records (used in §V.B production measurement).

- **Pipeline:** the SDRF pipeline.

- **Target:** Apache Iceberg on S3-compatible object storage, managed via a metadata catalog.

- Cycle configuration and cluster specs: as configured by the managed service; exact instance types and shard counts are managed by the deployment infrastructure.

Controlled benchmark setup:

- Workload A (single-object): Salesforce custom object, 10 M records. Used in §V.D.1 (shard scaling).

- **Target systems:** Apache Iceberg on S3, queried via standard SQL.

- Runs: Mean ± 95% CI over ≥5 independent runs.

B. Production-Deployment Ingestion Time

Metric	Value
Records ingested	10,000,000
Wall-clock time	380 s
Effective throughput	≈ 26,316 records/s

A controlled ingestion of 10 M records from a single Salesforce custom object into Apache Iceberg completed in 6 min 20 s (380 s), yielding an effective throughput of approximately 26,316 records/s. This figure reflects the production cluster's default multi-shard configuration. The throughput is derived arithmetically from the wall-clock figure and the 10 M-row dataset. This is a single-tenant, single-object measurement. We report controlled measurements with 95% CIs in §V.D. This throughput is enabled by the coordinate mapping (C1, §IV.C): monotonic (x, y) ordering allows the target to consume shards in parallel without cross-shard coordination.

C. Multi-Entity Concurrency

A second test ingested 1 M records distributed across four concurrent Salesforce objects, completing in **under 180 s**. The aggregate throughput is therefore **≥ 5 556 records/s** across 4 concurrent streams. Comparing to the §V.B single-object rate of ≈ 26,316 records/s on a 10 M-row workload:

Observation (inferred, not directly measured): *Per-entity throughput under 4-way concurrency is approximately 5 556 / 4 = 1 389 records/s per entity, which is ≈ 5 % of the single-entity peak of 26 316 records/s. This suggests that small-workload concurrent runs are dominated by per-entity setup cost rather than by shared infrastructure saturation. Large-workload concurrent behavior is not measured. Per-table parallelism is a direct consequence of the checkpoint-isolation protocol (C2, §IV.D): each table maintains an independent checkpoint chain, so a slow table cannot block others.*

D. Scaling and Schema Detection

The following measurements complement the production-deployment evidence with controlled experiments against the implementation.

D.1. Shard-count scaling. Throughput (records/s) vs. shard count $N \in \{1, 4, 16, 32, 64\}$ at fixed 10 M-row workload.

- 1 shard: 6,420 records/s (±310); 4 shards: 22,850 records/s (±890); 16 shards: 51,200 records/s (±1,450); 32 shards: 68,400 records/s (±2,100); 64 shards: 72,100 records/s (±2,800). Throughput plateaus at 32 shards, likely bounded by the 10-worker cluster capacity and Salesforce per-org API concurrency limits; isolating the specific bottleneck is future work.

- The controlled single-shard baseline (6,420 records/s) scales to 72,100 records/s at N=64, exceeding the production deployment's 26,316 records/s. The production figure reflects a managed configuration with additional overhead (target commit, checkpointing, schema validation) not present in the extraction-only controlled benchmark. The controlled benchmark measures extraction throughput only; end-to-end measurements including Iceberg commit overhead are reported only for the production deployment (§V.B). Shard-level parallelism is governed by the Link Manifest's ShardingPolicy and NumShards fields (D1, §IV.E); the coordinate mapping (C1) guarantees that events within each shard remain monotonically ordered regardless of shard count.

D.2. Schema-evolution detection overhead. We measured compare-and-detect wall-clock time as a function of column count.

- 10 columns: 12 ms (±2); 100 columns: 48 ms (±5); 1,000 columns: 340 ms (±18); 10,000 columns: 3,200 ms (±120). Schema detection is O(n) in column count. The 1,000 and 10,000-column measurements use a synthetic schema harness, as Salesforce limits objects to 800 custom fields. At 1,000 columns, detection adds < 350 ms to a cycle that takes > 120 s. This measures the overhead of the compare-and-detect schema-evolution procedure (D2, §IV.F).

E. Integration Decoupling

SDRF's reduction of N×M source-target integrations to N+M implementations (§II.E) is an architectural property of the specification, not a throughput or

latency metric. We provide structural evidence that the decoupling holds in our deployment:

Contract surface. The entire source-target coupling is expressed through two artifacts: the Link Manifest (§IV.E) and the SDRF envelope schema for DDL and DML events (§IV.G–H). Any source implementation that produces a conforming Link Manifest and conforming events, and any target implementation that consumes them, interoperates without further agreement. No source-to-target API, shared library, or out-of-band configuration is required.

Module independence. In our deployment, the Salesforce source connector and the Iceberg target consumer are independent modules. Neither imports the other. Their only shared dependency is a specification module defining the Link Manifest fields and the SDRF event schema. A change in one module that conforms to the specification does not require any change in the other.

Independent evolution. Over the development history of the production deployment, source-side changes (Salesforce API handling, change-indicator interpretation, schema-diff logic) and target-side changes (Iceberg commit batching, schema-application ordering) have been made without cross-module edits. The Link Manifest Version field (§IV.E) provides a coordination point for envelope evolution; no such envelope-breaking change has been required to date.

These observations demonstrate the decoupling in a single-source, single-target deployment. The stronger N+M claim — that heterogeneous sources and targets interoperate through the same SDRF intermediate layer — requires validation across additional implementations, identified as future work (§VIII.C).

F. Measurement Provenance Summary

Claim	Source	Strength
10 M-row ingestion in 380 s	This work	Production measurement, single run, no CI reported
≈ 26,316 records/s throughput	Derived from This work	Arithmetic from reported wall-clock
≥ 5 556 records/s aggregate, 4 concurrent objects, 1 M rows	This work	Derived from "under 3 minutes" upper bound
Per-entity concurrency at ≈ 5 % of peak	Inferred	Ratio of two This work numbers; controlled per-entity breakdown is future work
Monotonicity of `(x, y)` coordinate	§IV.C.2	Informal proof under stated assumptions
Restart determinism	§IV.C.2	Informal proof; relies on PK tiebreaker (§IV.C)
Bounded staleness at `Δ = 120 s`	§IV.C.2	Assumption + protocol; empirical validation required
Shard scaling, schema detection	§V.D	Controlled benchmark, 5 runs, 95% CIs reported
N+M integration decoupling	§V.E	Architectural evidence (contract surface, module independence); single source/target; heterogeneous validation is future work

VI. Discussion

A. When to Use SDRF

SDRF is well-suited for SaaS sources with `SystemModstamp`-style change indicators and for targets that already consume log-style CDC. For sources that emit native change-events (e.g., Salesforce Platform Events, webhook-based sources), an event-streaming pipeline will yield lower lag; SDRF is a fit when the SaaS provider does not publish such events or publishes them with lower fidelity than the query-based API.

B. When SDRF Is a Poor Fit

- Sources with no `SystemModstamp`-equivalent field (pure snapshot APIs).

- Workloads requiring sub-second replication lag (SDRF's cycle floor is bounded by `Δ + cycle interval`).

- Targets that cannot accept late-arriving events (SDRF permits rare late arrivals, see §IV.C.3).

VII. Threats to Validity

A. Internal Validity

- Measurement protocol. Lag measurements require synchronized clocks between the SaaS provider and the measurement harness. We use the SaaS-returned SystemModstamp as the event's origin time and the target's commit time (via audit log) as the arrival time. Both systems synchronize to UTC via NTP; while cross-system clock skew cannot be directly measured, it is bounded by typical NTP accuracy (tens to low hundreds of milliseconds) and is negligible relative to observed replication lag (minutes-scale).

B. External Validity

- Only Salesforce is evaluated as a source. Generalization to ServiceNow, Zendesk, SAP, Workday is discussed as future work (§VIII.C).

- Only Iceberg is evaluated as a target. Redshift, Snowflake, BigQuery, and Databricks Delta adapters are plausible but untested.

- Evaluation is performed in a single cloud region. Cross-region or multi-cloud performance is out of scope.

C. Construct Validity

- The monotonicity lemma (§IV.C.2, Lemma 1) assumes the SaaS API returns results in `SystemModstamp ASC` order deterministically. Salesforce documents this ordering guarantee; other SaaS providers must be validated individually.

VIII. Known Limitations and Future Work

A. Schema-Evolution Constraints

Column renames appear as `DELETE_COLUMN` + `ADD_COLUMN`; column reordering is not detected. Planned mitigation: column-value-fingerprint-based rename heuristic, evaluated on a curated schema-change corpus with measured precision/recall.

B. Data-Representation Constraints

No UPDATE before-images (API limitation). Primary-key mutations are assumed unsupported for current SaaS sources; if one occurs, SDRF emits `DELETE` (old PK) + `INSERT` (new PK) rather than a single `UPDATE`. Complex types (Struct, Map, List) are serialized as JSON strings; native nested-type support is planned for format version 2.0.

C. Future Work

Event-based CDC integration for supported providers (e.g., Salesforce Platform Events) to drop below the cycle floor; extended provider and target support (ServiceNow, Zendesk, Redshift, Snowflake); formal semantic-version evolution of the envelope; before-image reconstruction for UPDATE using cached record state on the connector.

IX. Conclusions

We presented SDRF, a CDC envelope for SaaS replication designed around two primary contributions: a monotonic `(epoch_ms, event_count)` coordinate mapping with restart determinism and bounded staleness under a safety offset, and a table-level checkpoint-isolation protocol with a linked chain and O(1) `LATEST_*` lookup. The Link Manifest and compare-and-detect schema-evolution mechanisms are supporting design points that reuse established patterns. A production deployment of SDRF powers Salesforce ingestion into Apache Iceberg on S3, completing a 10 M-row ingestion in 6 min 20 s.

The combination of timestamp-based coordinate mapping, table-level checkpointing, and a manifest-based decoupling contract is designed for systems where transaction-log-based CDC is unavailable. While evaluated only on Salesforce in this paper, the approach generalizes to other timestamp-only SaaS sources; empirical validation on additional providers is future work (§VIII.C).

Acknowledgment

The authors thank our engineering organization for infrastructure support.

References

[1] R. Hauch et al., "Debezium: Change Data Capture for Apache Kafka," Red Hat, 2017. [Online]. Available: https://debezium.io

[2] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, et al., "Building a Replicated Logging System with Apache Kafka," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1654–1655, 2015.

[3] Debezium Community, "Incremental Snapshots," Debezium Documentation, 2024. [Online]. Available: https://debezium.io/documentation/reference/stable/configuration/signalling.html

[4] M. Kleppmann, Designing Data-Intensive Applications. Sebastopol, CA: O'Reilly Media, 2017.

[5] Apache Software Foundation, "Apache Iceberg Table Specification," 2024. [Online]. Available: https://iceberg.apache.org/spec/.

[6] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, et al., "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3411–3424, 2020.

[7] Apache Software Foundation, "Apache Hudi," 2024. [Online]. Available: https://hudi.apache.org

[8] A. Halevy, A. Rajaraman, and J. Ordille, "Data Integration: The Teenage Years," in *Proc. 32nd Int. Conf. Very Large Data Bases (VLDB)*, Seoul, Korea, 2006, pp. 9–16.

[9] M. Lenzerini, "Data Integration: A Theoretical Perspective," in *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS)*, 2002, pp. 233–246.

[10] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the Database Schema Evolution Process," *VLDB J.*, vol. 22, no. 1, pp. 73–98, 2013.

[11] P. Vassiliadis, "A Survey of Extract-Transform-Load Technology," *Int. J. Data Warehous. Mining*, vol. 5, no. 3, pp. 1–27, 2009.