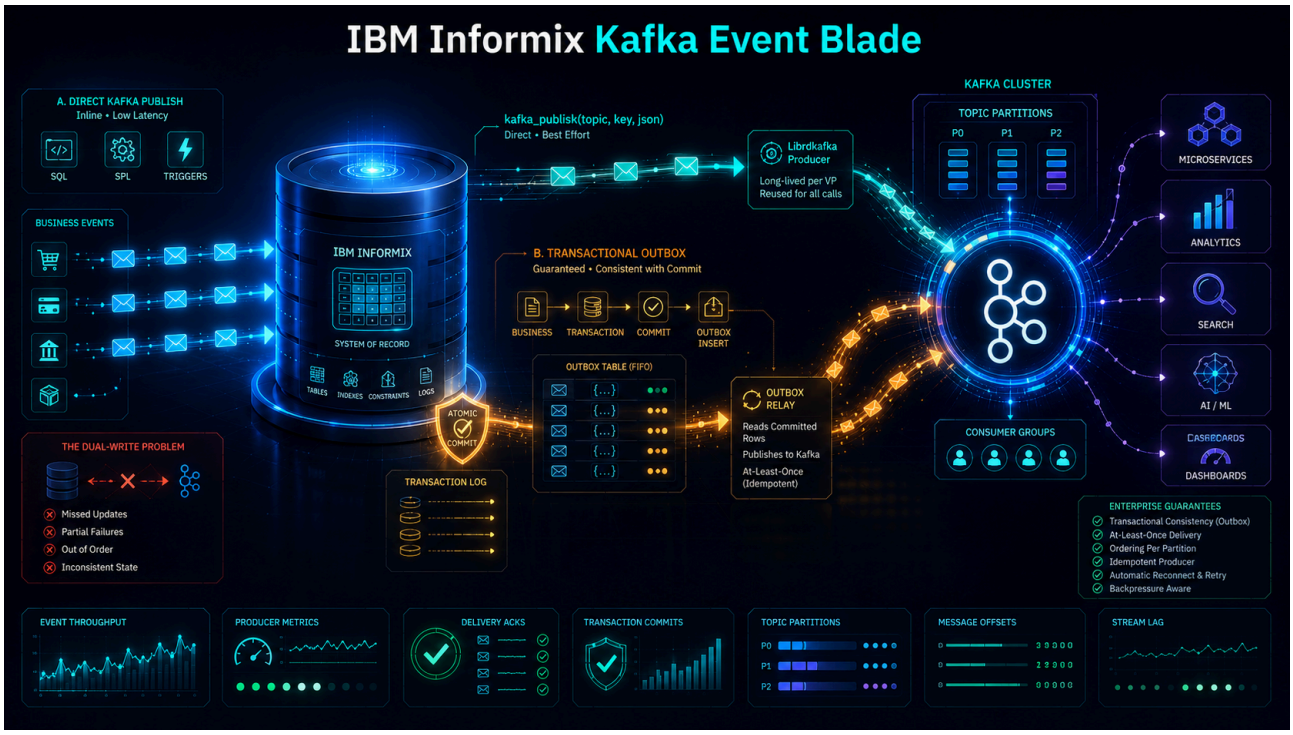
ifxtools

Document Version: 1.0 – 2026-08-13

Informix Kafka CDC Connector

Airtool

Content



This document describes an enterprise **Kafka Change Data Capture (CDC) connector** for IBM Informix: a **pure-Java, external** service that captures committed row changes from the Informix logical log and publishes a **Debezium-style JSON envelope** per change to **Apache Kafka** — with **nothing running inside the database engine**. It is the Informix equivalent of the PostgreSQL + Debezium model, built on IBM's `ifx-changestream-client` library over the Informix CDC API (`syscdcv1`).

The connector is packaged and tested like the sister DataBlades: a multi-engine Docker + JUnit harness (Informix 14 + 15 against a single-node KRaft Kafka broker) and one deployment flow. The Java source is the source of truth for the behaviour described here.

> **Headline.** Point the connector at a logged database and a list of tables, and every `INSERT`, `UPDATE` and `DELETE` becomes a Kafka event — `{"op":"c|u|d","source":{...},"before":...,"after":...}` — keyed by primary key, with a tombstone on delete, **no triggers and no application changes**. Because capture reads **committed transactions** from the log, an event exists if and only if its transaction committed: there is no dual-write problem to solve. The connector runs **outside the engine** and **checkpoints its log position**, so a restart resumes instead of re-reading from zero.

1. Motivation — Events Trapped Inside the Database

Modern architectures are event-driven: order placed, payment captured, loan approved, stock moved. The system of record for those facts is very often a relational database, and the events that other systems need are born there as row changes. Getting those facts onto **Apache Kafka**, reliably and in order, is the recurring integration problem.

PostgreSQL solved this cleanly and `outside` the database: it exposes its change stream through **logical decoding**, and the **Debezium** / Kafka Connect ecosystem turns that stream into Kafka topics with no triggers and no application changes. Informix has the same raw capability — a **Change Data Capture API** over the logical log — but historically far fewer off-the-shelf ways to land it on Kafka. This connector is that missing piece: it does for Informix exactly what Debezium does for PostgreSQL.

- **Table-driven events** — row changes streamed generically off the log, for replication, cache / search offload and event-carried state transfer.
- **Transactionally consistent** — capture reads only committed transactions, so events never diverge from the data (no dual-write problem).
- **Outside the engine** — a standalone process that never competes with the database for CPU and carries no native code in the server.

> **Why the Debezium model.** Keeping capture out of the engine is what makes Postgres+Debezium robust: the database does one thing well (write its log), and an external connector does the streaming. Informix gains the same properties here — transactional consistency, no triggers, no schema changes, and a familiar operational shape — through the standard Informix CDC API rather than any in-engine extension.

1.1. System architecture

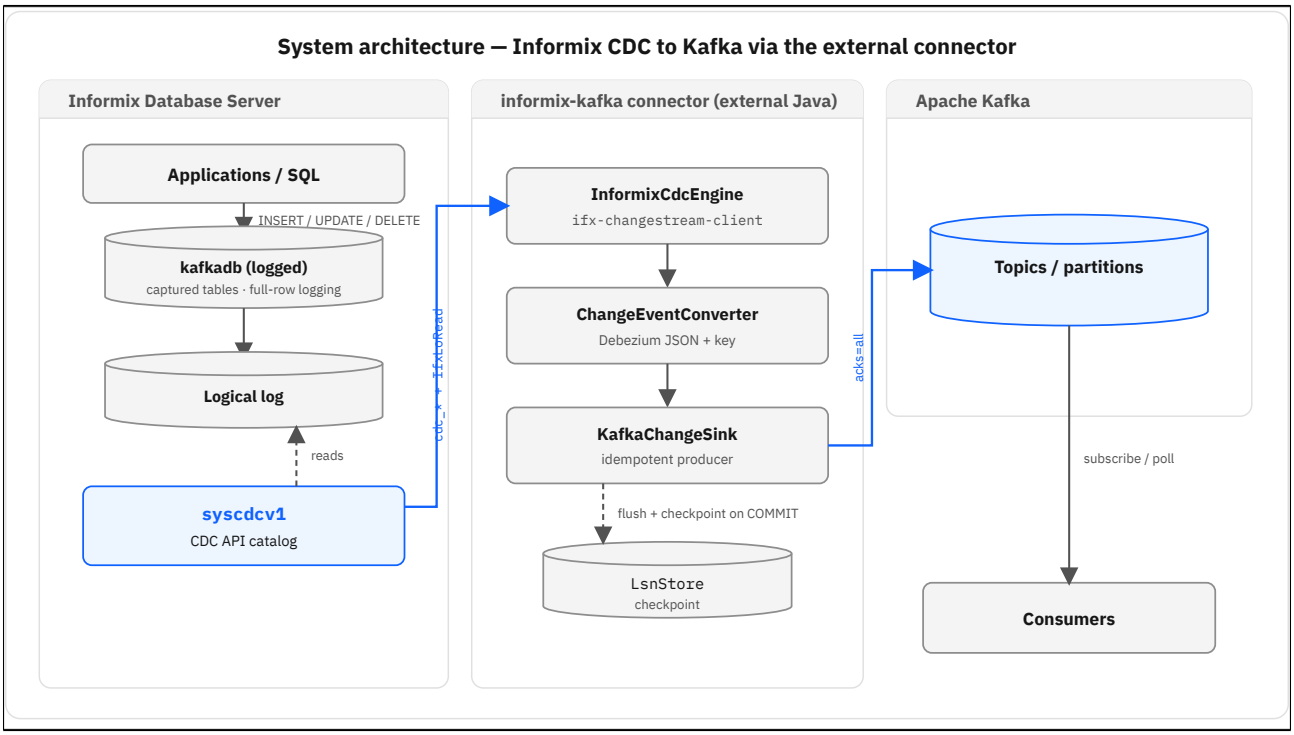


Figure 1 — System architecture — Informix CDC to Kafka via the external connector

2. The Database-to-Kafka Landscape

What other databases do, and where this connector sits. The dominant pattern for streaming database changes to Kafka is **external, log-based CDC** — and that is precisely what this connector brings to Informix.

Table 1 — Reference architectures for streaming database changes to Kafka

Database	Dominant pattern	Mechanism	Relation to this connector
PostgreSQL	Logical decoding → Debezium → Kafka Connect	WAL logical replication slots	The model this connector follows for Informix.
SQL Server	CDC tables → Debezium → Kafka	Transaction-log capture	Same shape.
MySQL	Binlog → Debezium → Kafka	Row-based binlog	Same shape.
MongoDB	Change Streams → official connector	Oplog-derived change streams	Connector-based, external to the DB.
Oracle	Transactional Event Queues (in-DB broker)	A broker `inside` the database	The opposite (in-database) approach — deliberately not taken here.
Informix (this connector)	**CDC API → Java connector → Kafka**	Logical-log capture via `ifx-changestream-client`	Brings the external, log-based Debezium model to Informix.

3. The Capture Model — How Informix CDC Works

Informix exposes change data capture as a set of SQL-callable functions in a control database, `syscdcv1`, plus a smart-blob read for the change stream. The connector drives them through IBM's `com.ibm.informix:ifx-changestream-client` library (`InformixCdcEngine`). The lifecycle:

```
cdc_opensess(server, ...)                    -> open a capture session, get a session id
cdc_set_fullrowlogging('db:owner.table', 1)    -> enable full before/after row images
 for the table
cdc_startcapture(session, 0, 'db:owner.table', 'col,col,...', LABEL)    -> capture;
 LABEL = partnum
cdc_activatesess(session, resumeLSN)           -> begin delivering from resumeLSN (0
 = from now)
loop:  IfxSmartBlob.IfxLoRead(session, buf)    -> raw change bytes
       IfxCDCRecordBuilder.buildRecord(buf)    -> a typed change record (INSERT/UPDATE/
DELETE/COMMIT/...)
cdc_endcapture / cdc_set_fullrowlogging(...,0) / cdc_closesess       -> clean teardown
```

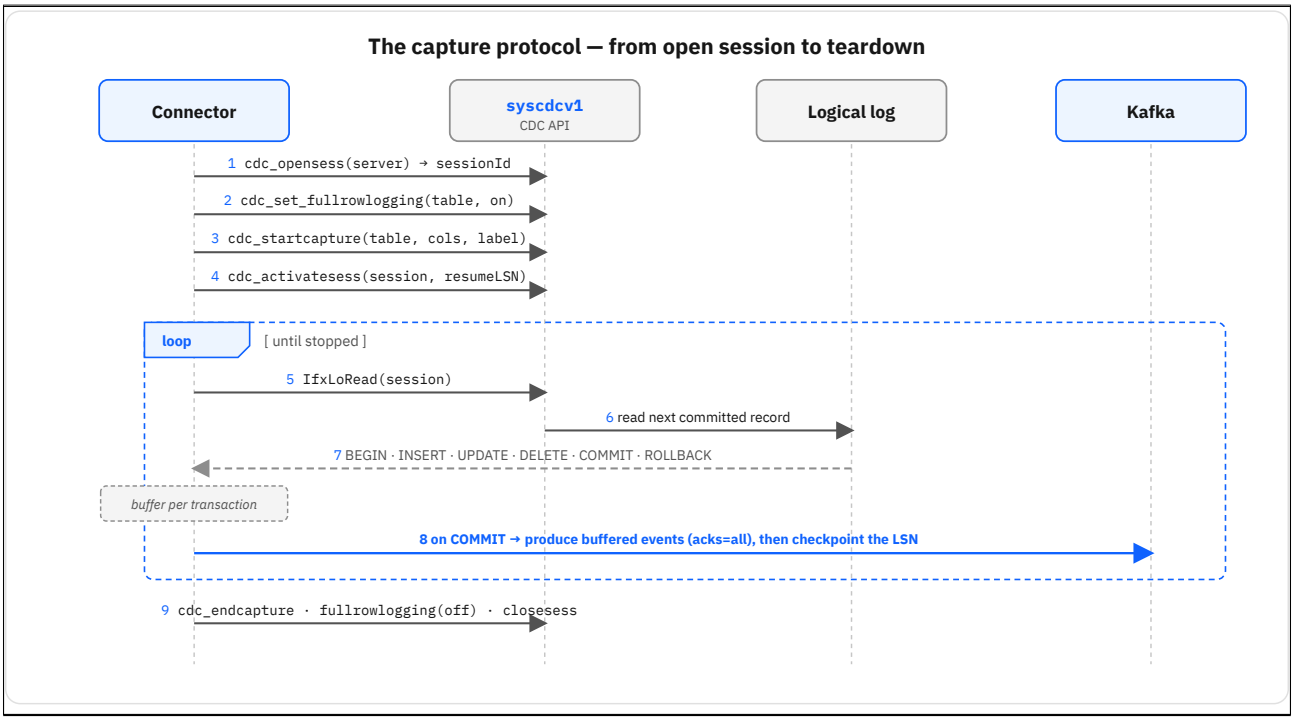


Figure 2 — The capture protocol — from open session to teardown

> `partnum` **is the join key.** Each table's capture `label` is set to its Informix partition number, so every change record resolves straight back to its source table (its columns, primary key and target topic). The library decodes each record into a typed `Map<column, value>`; the connector converts values faithfully — numbers stay numeric (DECIMAL kept exact), temporal types become ISO strings, SQL NULL becomes JSON `null`.

Full-row logging must be turned back off on shutdown. A captured table left "defined for replication" cannot be altered or dropped. The connector's graceful teardown disables full-row logging and closes the session; the Docker service is given a stop grace period so this runs before the process is killed.

4. The Change-Event Format

Each captured change becomes one JSON message, keyed by the row's primary key, shaped like Debezium's so consumers written against that shape work unchanged:

```
{
  "op": "c",                                      -- c = insert, u = update, d = delete
  "source": { "db": "kafkadb", "owner": "informix",
              "table": "orders", "partnum": 1049160,
              "seq": 25775, "txId": 42 },
  "before": null,                                          -- insert: no before image
  "after":  { "id": 1, "customer": "ACME Corp",
              "amount": 250.00, "status": "PLACED" },
  "ts_ms": 1784021205428
}
```

Table 2 — Envelope by operation

Operation	op	before	after	Extra
INSERT	c	null	the new row	—
UPDATE	u	the old row	the new row	pairs the CDC BEFORE_UPDATE + AFTER_UPDATE images
DELETE	d	the old row	null	also emits a **tombstone** (key + null value)

> The Kafka **key** is the row's primary key, so per-key ordering and log compaction work. The default topic is `cdc.<database>.<table>`, overridable per table. The message is stamped with source headers (`op`, `source_table`, `source_db`) for routing/filtering without parsing the payload.

5. Transaction Consistency and the Outbox Pattern

An event exists **if and only if its transaction committed** — so there is no dual-write problem and no phantom events. The logical log delivers a transaction's operation records followed by a COMMIT or ROLLBACK record, so the connector **buffers each transaction's changes and emits them only on its COMMIT**; a **rolled-back transaction is discarded** and produces nothing (verified by the test suite). On commit it produces the buffered events, flushes the Kafka producer, and only then advances its log-position checkpoint — so delivery is **at-least-once** and the checkpoint never runs ahead of what Kafka has durably stored. Consumers dedupe on source.seq.

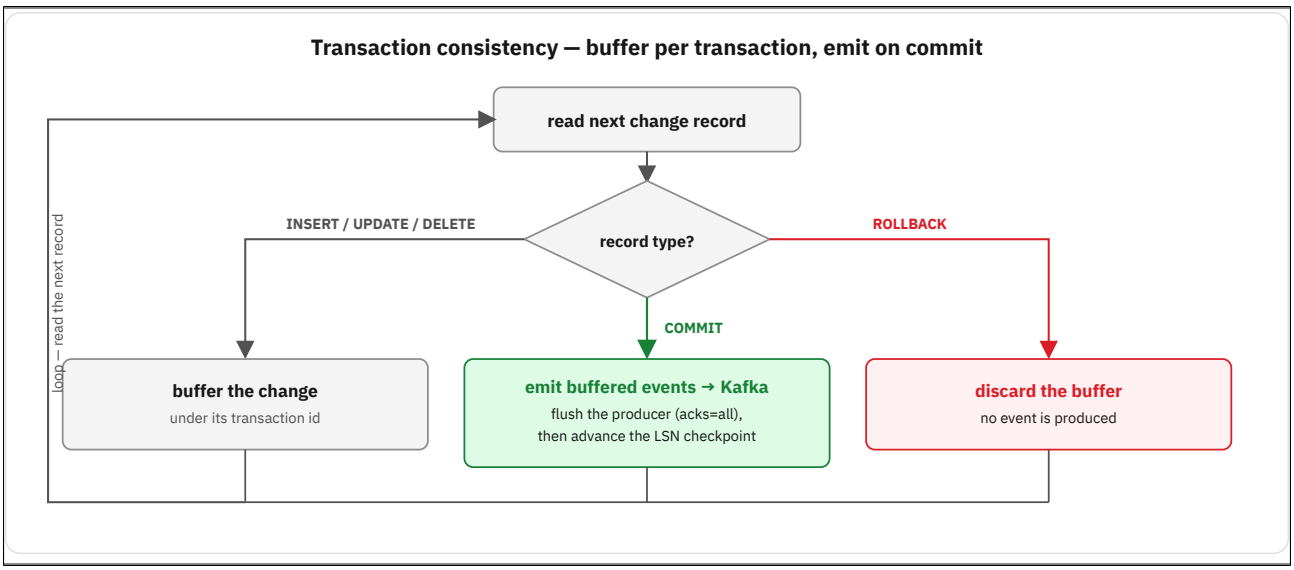


Figure 3 — Transaction consistency — buffer per transaction, emit on commit

The transactional outbox, with zero in-engine code. When a single logical event (loan_approved) spans several table writes and needs a custom payload, use Debezium's **outbox event router** pattern: the application INSERTs a row into an ordinary outbox table in the same transaction as the business change, and the connector captures that table. The event commits atomically with the business change and is streamed to Kafka — no trigger, no UDR, no relay to operate.

6. Resumable Capture — The LSN Checkpoint

The connector persists the last committed CDC **sequence number** (log position) at each commit and, on restart, passes it to `cdc_activatesess()` to resume — exactly as IBM's CDC "Restarting data capture" procedure prescribes — instead of re-reading from position zero.

```
-- at each COMMIT the connector does, in order:
producer.flush();          -- block until Kafka has every record in this transaction
lsnStore.save(commitSeq);  -- only now advance the checkpoint  (at-least-once)

-- on the next start:
long resume = lsnStore.load();          -- e.g. 25775460784
cdc_activatesess(session, resume);      -- resume from the saved log position
```

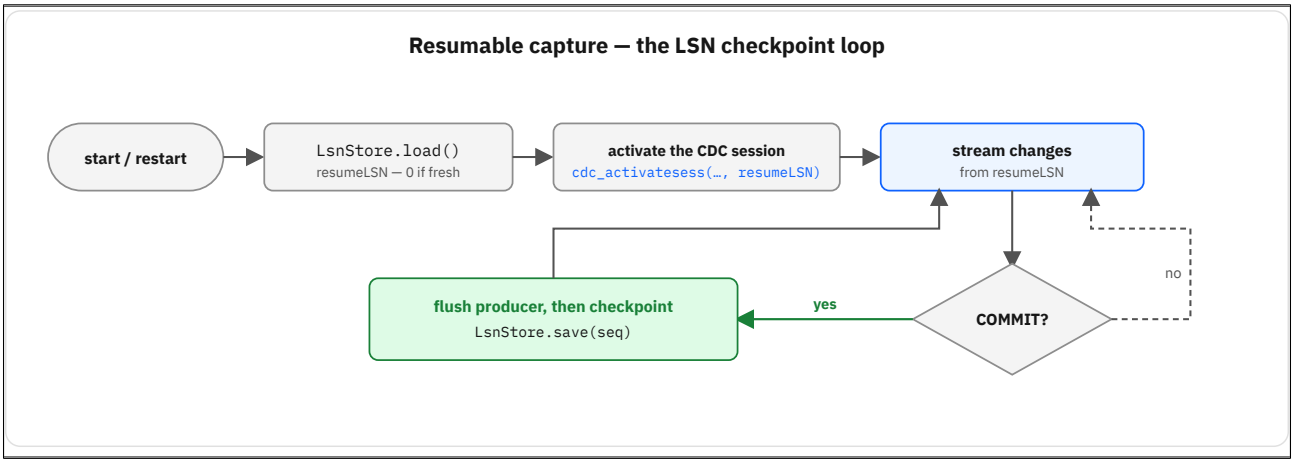


Figure 4 — Resumable capture — the LSN checkpoint loop

> A busy engine resumes seamlessly across a restart. Replaying a change committed `entirely` while no capture session is active depends on the logical log advancing — reliable under load, timing-sensitive on a fully idle engine — which is an operational nuance of Informix logical-log flushing, not a connector limitation.

7. Configuration

The connector is a single self-contained jar (`InformixKafkaCdcConnector`) configured by a `.properties` **file** passed as its argument. No SQL registration, no in-engine objects.

Table 3 — Connector configuration (a .properties file)

Key	Default	Meaning
`informix.server`	informix	the Informix server (sqlhosts) name
`informix.host` / `informix.port`	localhost / 9088	Informix host + SQLI port
`informix.user` / `informix.password`	informix / informix	connecting user (needs CDC / DBA privileges)
`informix.database`	kafkadb	application database (must be **logged**)
`informix.locale`	`(auto)`	connection locale; **auto-detected** from the database's `dbs_collate` — works for UTF-8 and ISO. Set only to override.
`cdc.tables`	—	**required** — comma-separated `owner.table[=topic]` list to capture
`cdc.topic.prefix`	cdc	default topic is `<prefix>.<db>.<table>`
`kafka.bootstrap`	localhost:19092	Kafka bootstrap servers
`kafka.<name>`	—	any Kafka **producer** property, the `kafka.` prefix stripped: `kafka.security.protocol` , `kafka.sasl.mechanism` , `kafka.sasl.jaas.config` , `kafka.ssl.truststore.location` , `kafka.compression.type` , …
`cdc.read.timeout`	5	idle-read window (seconds)
`cdc.heartbeat`	30	progress/liveness heartbeat interval (seconds); 0 disables it
`cdc.lsn.file`	cdc-offset.lsn	checkpoint file; blank → in-memory (no resume)

Configured by a file. The connector reads a single `.properties` file passed as its argument (`java -jar informix-kafka-connector.jar cdc.properties`); the only other accepted arguments are `--help` (which prints the key reference above) and `--version` . On startup it logs a copyright banner and the effective configuration with **secrets redacted**; while running it emits a periodic **heartbeat** (rows / transactions / last sequence) so operators can see it is alive and advancing; on a fatal error it logs the cause and exits non-zero.

8. Examples — a reproducible walkthrough

Each example is self-contained. The connector is an external process: you prepare Informix (a logged database + `syscdcv1`), configure the connector, run it, then perform ordinary SQL and observe the change events on Kafka as a consumer sees them (key, value). The examples mirror the JUnit suite: data → capture → asserted result.

8.1. 1 — Prepare Informix for CDC

```
-- ============================================================================
-- Example 1 — Prepare Informix for CDC
--  Goal: a logged database + the syscdcv1 CDC catalog (done once; setup.sh automates it)
-- ============================================================================
-- A logged database — CDC captures only logged databases (the logical log is the
 source):
CREATE DATABASE kafkadb WITH LOG;

-- The CDC API catalog (run as the informix user):
--   dbaccess - $INFORMIXDIR/etc/syscdcv1.sql

-- Full-row logging on captured tables is enabled by the connector at runtime; nothing
 to do here.
```

8.2. 2 — Configure and run the connector

```
-- ============================================================================
-- Example 2 — Configure and run the connector
--   Goal: capture the orders table to Kafka topic cdc.kafkadb.orders
-- ============================================================================
-- The connector reads a .properties file passed as its argument. Write cdc.properties:
--   informix.server=informix
--   informix.host=informix
--   informix.database=kafkadb
--   cdc.tables=informix.orders
--   kafka.bootstrap=kafka:9092
-- then run the daemon:
java -jar informix-kafka-connector.jar cdc.properties

-- Equivalently, as a Docker sidecar next to Informix (mounts the config file):
--   docker compose --profile cdc up -d --build kafka-cdc
```

```
discovered kafkadb:informix.orders partnum=1049160 cols=4 pk=id -> topic
 cdc.kafkadb.orders
CDC session 44040231 active on 'informix' (1 table(s))
connector started -- capturing 1 table(s) -> Kafka kafka:9092
```

8.3. 3 — Capture insert / update / delete

```
-- ============================================================================
-- Example 3 — Capture insert / update / delete
--   Topic : cdc.kafkadb.orders   (message key = primary key)
--   Goal  : every row change streams as a before/after envelope, no triggers
-- ============================================================================
CREATE TABLE orders (id INTEGER PRIMARY KEY, customer VARCHAR(40), amount DECIMAL(10,2),
 status VARCHAR(12));

INSERT INTO orders VALUES (1, 'ACME Corp', 250.00, 'PLACED');
UPDATE orders SET status = 'SHIPPED' WHERE id = 1;
DELETE FROM orders WHERE id = 1;
```

```
 -- consumer on topic 'cdc.kafkadb.orders' (op c/u/d + a delete tombstone):
 key value
 1         {"op":"c","source":{...},"before":null,"after":{"id":1,"customer":"ACME
  Corp","amount":250.00,"status":"PLACED"}}
 1              {"op":"u","source":{...},"before":{...,"status":"PLACED"},"after":
 {...,"status":"SHIPPED"}}
 1                                                {"op":"d","source":{...},"before":
 {"id":1,...,"status":"SHIPPED"},"after":null}
 1   (null value)   -- tombstone: lets a log-compacted topic drop the deleted key
```

8.4. 4 — The transactional outbox, via CDC

```
-- ============================================================================
-- Example 4 — Transactional outbox (Debezium outbox-router pattern), zero in-engine
 code
--   Topic : cdc.kafkadb.event_outbox
--   Goal  : a business event that commits atomically with the business change
-- ============================================================================
-- Capture an ordinary outbox table (add event_outbox to cdc.tables). The application
 writes the
-- event in the SAME transaction as the business change; the connector streams it
 after commit.
CREATE TABLE event_outbox (id BIGSERIAL PRIMARY KEY, aggregate VARCHAR(40), payload
 LVARCHAR(4000));

BEGIN WORK;
    INSERT INTO orders VALUES (7, 'Initech', 42.00, 'PLACED');
           INSERT   INTO   event_outbox   (aggregate,   payload)   VALUES   ('order',
 '{"event":"order_placed","id":7}');
COMMIT WORK;                -- both rows commit together, or neither does
```

```
 -- consumer on topic 'cdc.kafkadb.event_outbox' — the event appears only because
  the txn committed:
 key value
 1   {"op":"c","source":{"table":"event_outbox",...},"before":null,
       "after":{"id":1,"aggregate":"order","payload":"{\"event\":\"order_placed\",
 \"id\":7}"}}
```

8.5. 5 — Resume after a restart

```
-- ============================================================================
-- Example 5 — Resumable capture
--   Goal: a restarted connector resumes from its checkpoint, not from zero
-- ============================================================================
-- The connector checkpoints the last committed sequence to cdc.lsn.file at every
 commit.
--   cdc.lsn.file=/data/cdc-offset.lsn      (in cdc.properties)
-- ... run, capture some changes, stop the connector, then start it again:
java -jar informix-kafka-connector.jar cdc.properties
```

```
 resuming from checkpoint 25775460784 (/data/cdc-offset.lsn)
 CDC session 44040555 active on 'informix' (1 table(s))
 connector started -- capturing 1 table(s) -> Kafka kafka:9092
```

8.6. 6 — Authenticated + encrypted brokers (SASL / TLS)

```
-- ============================================================================
-- Example 6 — Authenticated + encrypted brokers (SASL / TLS)
--   Goal: publish change events to a SASL_SSL broker
-- ============================================================================
-- The Java Kafka client provides SASL/TLS natively; put the producer properties in
 cdc.properties
-- (any kafka.* key is passed to the producer with the kafka. prefix stripped):
--   kafka.bootstrap=kafka:9096
--   kafka.security.protocol=SASL_SSL
--   kafka.sasl.mechanism=PLAIN
--     kafka.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
 required username="client" password="client-secret";
--   kafka.ssl.truststore.location=/certs/kafka.truststore.p12
--   kafka.ssl.truststore.password=changeit
java -jar informix-kafka-connector.jar cdc.properties
```

```
 connector started -- capturing 1 table(s) -> Kafka kafka:9096    (change events
  delivered over TLS + SASL)
```

> **Java 24+ note.** SASL requires the bundled **kafka-clients 4.x**: the older 3.x client calls the JDK-removed `Subject.getSubject` and fails on Java 24+ with `"getSubject is not supported"`. This connector ships kafka-clients 4.x, so SASL/TLS works on modern JDKs out of the box. Producing over `SASL_SSL` is verified from capture to delivery by the test suite.

9. Deployment

The connector is a single self-contained jar — pure Java, no Informix Client SDK, no native libraries (the JDBC driver and `ifx-changestream-client` are bundled). Deploy it as a sidecar next to Informix.

9.1. Prerequisites

```
-- 1. A logged application database (CDC requires logging):
CREATE DATABASE kafkadb WITH LOG;

-- 2. The syscdcv1 CDC catalog (run once, as the informix user):
--      dbaccess - $INFORMIXDIR/etc/syscdcv1.sql

-- 3. A connecting user with CDC / DBA privileges (EXECUTE on informix.cdc_*, read
 on sysmaster).
--     Full-row logging on captured tables is enabled by the connector at runtime
 and turned back
--    off on clean shutdown.
```

Locale — auto-detected (UTF-8 and ISO). Informix CDC rejects full-row logging (SQL `-83790` / `-83796`) when the CDC session locale does not match the captured table's collation. The connector therefore reads the database's own locale (`dbs_collate` from `sysmaster:sysdbslocale`) and connects with it, so a UTF-8 database (`en_US.57372`) and an ISO database (a European `en_US.819` / `8859-15`) both work with no configuration. Captured tables must be created by a client whose locale matches the database.

Informix CDC cannot stream `BYTE`, `TEXT`, `LVARCHAR`, `BLOB` or `CLOB` columns; the connector automatically excludes those from the captured column list. The engine needs **no C compiler and no native Kafka library** — capture happens entirely in the external connector.

9.2. Build and run

```
# Build the self-contained connector jar:
mvn -DskipTests package        # -> target/informix-kafka-<version>-connector.jar

# Write a cdc.properties config file (see the Configuration table), then run:
#   informix.host=informix
#   informix.database=kafkadb
#   cdc.tables=informix.orders
#   kafka.bootstrap=kafka:9092
java -jar target/informix-kafka-*-connector.jar cdc.properties

#  Or  as  a  Docker  sidecar  against  the  test  stack  (mounts  docker/connector/
cdc.properties.example):
docker compose up -d                                    # kafka + informix 14/15
docker compose --profile cdc up -d --build kafka-cdc    # the connector
```

Graceful shutdown matters. On `SIGTERM` the connector ends capture and turns full-row logging back off, so captured tables are not left "defined for replication" (which would block `ALTER` / `DROP`). Allow it a stop grace period.

9.3. Roadmap

Table 4 — Roadmap

Item	Notes
Initial snapshot	A consistent load of existing rows before streaming (Debezium-style). The main gap today — the connector currently captures changes from the moment it starts.
Schema registry	Avro / Protobuf serialization + a schema-registry hook.
Kafka Connect packaging	Wrap the same capture engine as a `SourceConnector` / `SourceTask` for Connect-native offset management, converters and SMTs.
Richer keys / typing	Composite-key messages; preserve DECIMAL scale exactly.

10. Glossary — Kafka and Change-Data-Capture Terms

Terms specific to Kafka and change data capture, for readers fluent in Informix but new to the domain. (Informix terminology is assumed known.)

Table 5 — Glossary

Term	Definition
Apache Kafka	A distributed, partitioned, replicated commit log used as an event-streaming platform. Producers append records to topics; consumers read them at their own pace.
Topic	A named, append-only stream of records. Here, one topic per captured table by default.
Partition	A topic is split into partitions for parallelism and ordering. Order is guaranteed only within a partition; the primary-key message key keeps a row's changes in order.
Key	An optional record key; records with the same key go to the same partition. Here, the row's primary key.
CDC (change data capture)	Capturing row-level changes from a database's transaction log to propagate them downstream — the Debezium model.
Debezium	The de-facto open-source CDC platform that streams database changes into Kafka; the reference model this connector follows for Informix.
Change envelope	The `{op, source, before, after, ts_ms}` JSON shape emitted per change (Debezium-compatible).
syscdcv1	The Informix control/catalog database the CDC API functions (`cdc_opensess` , `cdc_startcapture` , ...) live in.
Full-row logging	Per-table logging of complete before/after row images, required for CDC and enabled by the connector via `cdc_set_fullrowlogging` .
partnum	An Informix table's partition number, used as the CDC capture `label` — the join key from a change record back to its source table.
LSN / sequence number	The log position of a change. The connector checkpoints the last committed sequence so a restart resumes via `cdc_activatesess` .
ifx-changestream-client	IBM's Java library that wraps the Informix CDC API and decodes each change into a typed record; the library this connector is built on.
acks	Producer durability setting: `0` (no ack), `1` (leader ack), `all` (all in-sync replicas ack — used here).

Term	Definition
Idempotent producer	A producer mode that de-duplicates retries so each message is written once per partition (requires `acks=all`).
At-least-once / exactly-once	Delivery guarantees. The connector is at-least-once; consumers dedupe on `source.seq` for effectively-once processing.
Tombstone	A record with a non-null key and a null value, used in compacted topics to signal deletion of that key. Emitted on every delete.
Transactional outbox	Writing an event to a DB table in the same transaction as the business change; captured by CDC so delivery is consistent with the commit.
Dual-write problem	Writing to two systems (DB + Kafka) without a shared commit, so one can succeed while the other fails. Log-based CDC avoids it by reading only committed transactions.
KRaft	Kafka's built-in consensus (Kafka Raft) that replaces ZooKeeper for cluster metadata; the test broker runs in KRaft mode.

11. License, Trademarks and Disclaimer

This product is a **proprietary, commercial software product** owned and published by **Airtool Technologies** (airtool.io). **All rights reserved.** It is licensed, not sold, and its use is governed by a commercial license agreement. No right to use, copy, modify, distribute, sublicense, or create derivative works is granted except as expressly permitted in a written license from Airtool Technologies. Unauthorized reproduction or distribution is prohibited. Contact info@airtool.io for licensing terms.

Purpose of this document. This document is provided for information only. It describes the product as at the date of publication, is subject to change without notice, forms no part of any agreement, and creates no representation, warranty or commitment. The terms governing any use of the software are those of the written license agreement between you and Airtool Technologies.

Warranty. Warranties, if any, are those expressly set out in that agreement. Except as expressly so provided, and to the maximum extent permitted by applicable law, the software is provided **"as is"**, and Airtool Technologies disclaims all other warranties and conditions, whether express, implied or statutory, including the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

Limitation of liability. Except as expressly provided in that agreement, and to the maximum extent permitted by applicable law: neither party is liable for indirect, incidental, special, punitive or consequential damages, or for loss of profits, revenue, data or business, however caused; and each party's total aggregate liability is limited as set out in that agreement. Nothing in this document excludes or limits liability for death or personal injury caused by negligence, for damage to tangible property caused by negligence, for fraud or fraudulent misrepresentation, for wilful misconduct, or for any other liability that cannot be excluded by law.

Trademarks — no affiliation. IBM® and **Informix®** are trademarks or registered trademarks of International Business Machines Corporation; Informix is currently developed and distributed by HCL under license. **Apache®** and **Apache Kafka®** are trademarks of the Apache Software Foundation. This project is **independent** and is **not** affiliated with, sponsored by, or endorsed by IBM or HCL. Product names are used solely for identification and interoperability (nominative fair use); no third-party logos are distributed.

Commercial support. Licensing, production support (with SLAs), custom development and consulting are available from **Airtool Technologies** — airtool.io · info@airtool.io. Support entitlements are defined by your commercial license or support agreement. Bundled third-party components retain their own licenses.