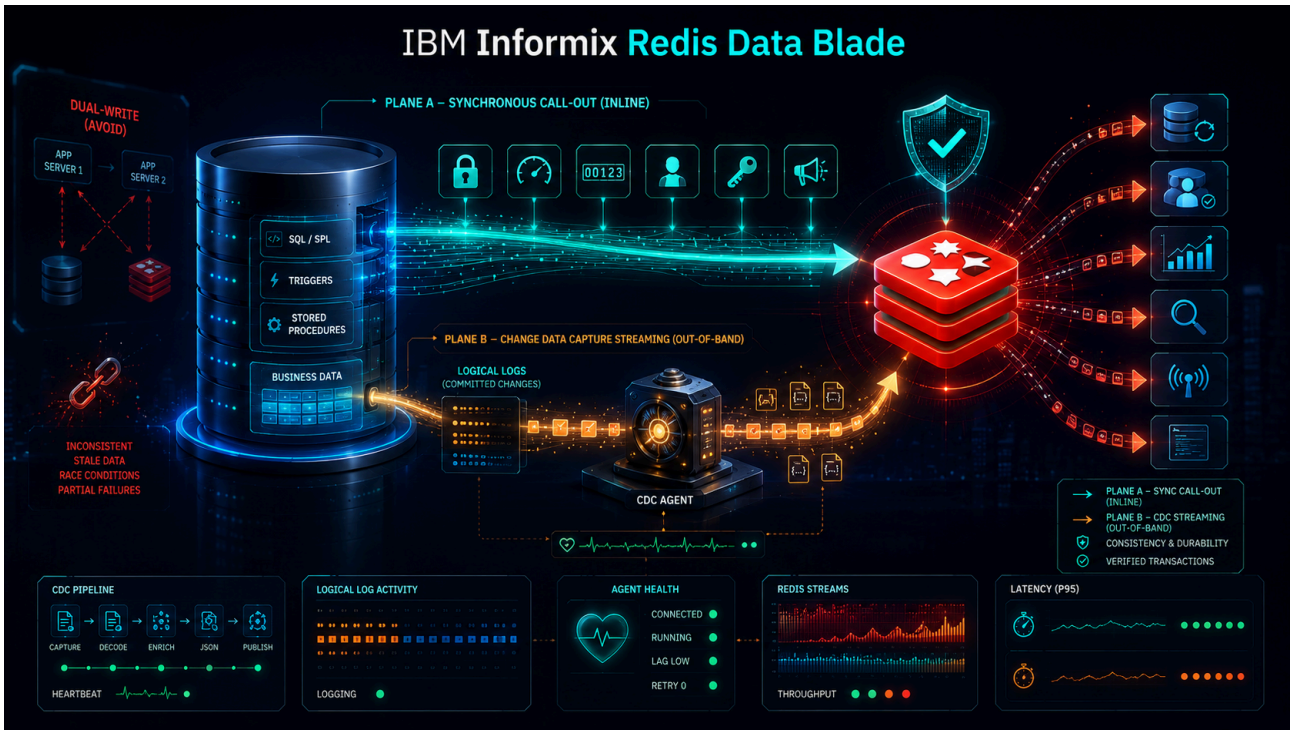
ifxtools
Document Version: 1.0 – 2026-08-13

Informix Redis DataBlade + CDC connector

Airtool

Content



This document describes an enterprise **Informix ↔ Redis** integration delivered as a C DataBlade: a set of UDRs that call **Redis directly from SQL, SPL and triggers**, plus a **Change-Data-Capture (CDC) agent** that streams committed Informix row changes as JSON onto Redis. It explains the two integration planes — a **synchronous** in-engine call-out plane and an **asynchronous** change-stream plane — when to use each, how they behave when Redis fails and recovers, what they log, and how to compile and register them.

> **Headline.** Both planes are implemented in C and proven on IBM Informix 14.10 and 15.0. The synchronous UDRs (locks, rate-limit, idempotency, counters, cache-aside, queue/pub-sub) pass a full live test suite; the CDC bridge is verified from trigger to cache by six real-scenario tests that assert the exact JSON landing on Redis streams — including lossless fixed-scale `DECIMAL`, 64-bit `BIGINT`, ISO-8601 `DATETIME`, `NULL` as JSON null, and injection-safe escaping of arbitrary user text.

1. Motivation — Two Ways the Database Talks to Redis

A great deal of enterprise infrastructure puts Redis beside an RDBMS for caching, sessions, counters, distributed locks, rate limiting and event fan-out. The recurring problem is **keeping the two in step** and **avoiding dual-writes** from application code that can forget, race, or partially fail. A cache entry that the application updates but forgets to invalidate serves stale data; an event that is published before its transaction commits is a phantom; two app servers that each "remember to update Redis" eventually disagree.

Keeping the Redis interaction **inside the database**, next to the business data, removes that class of bug: a trigger invalidates exactly the cache key whose row changed, a counter is incremented in the same logical step as the insert that earned it, and a change feed is derived from the **committed** log rather than from hopeful application code. This blade offers two complementary planes:

Table 1 — The two integration planes

Plane	Direction	Mechanism	Guarantee	Typical use
A — synchronous call-out	SQL → Redis, inline	`redis_*` C UDRs (`redis.so`)	best-effort, in transaction context	locks, rate-limit, idempotency, counters, cache-aside, event from a trigger
B — change stream	Informix → Redis, out-of-band	`redis_cdc` ESQL/C agent	at-least-once, durable	cache invalidation/sync, materialized views, search/analytics feeds, outbox

The IBM Informix CDC API may **not** be called from a user-defined routine, so Plane B is a standalone C client process, not an in-engine UDR. Both planes publish through the same shared Redis client core (connection, timeout, AUTH, failure cooldown, reply rendering and JSON escaping), so behaviour and key conventions are identical across them.

1.1. System architecture

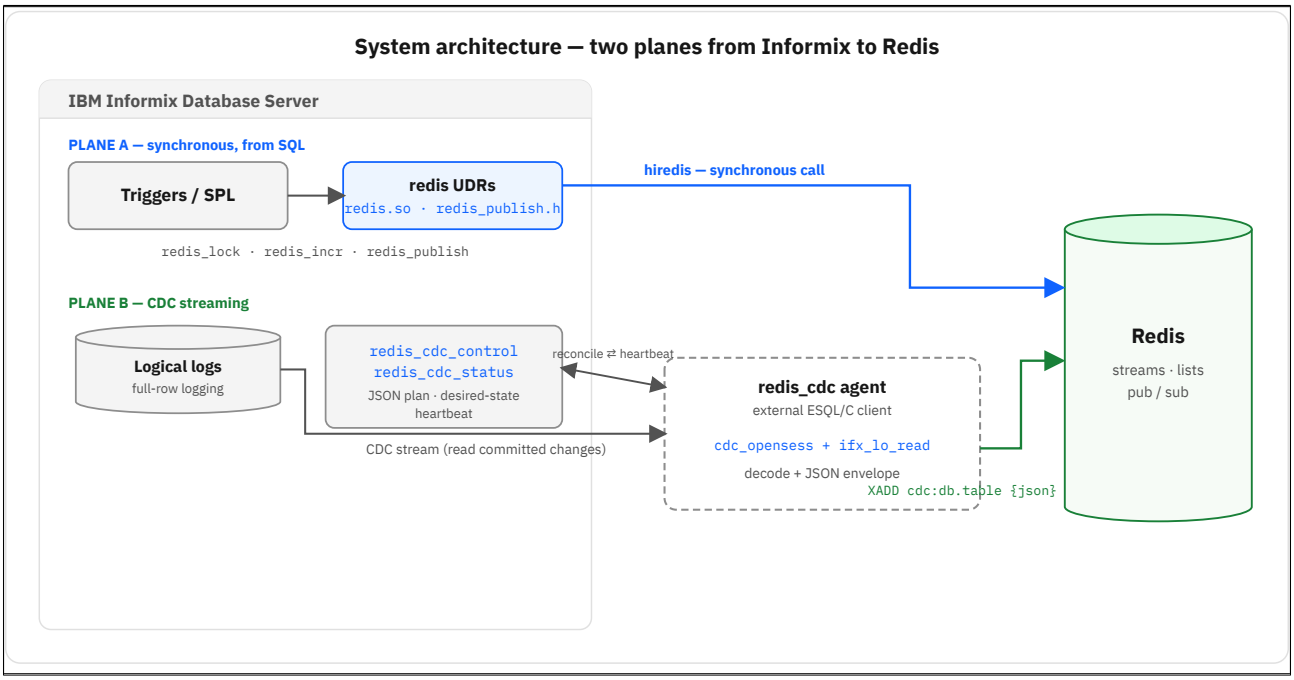


Figure 1 — System architecture — two planes from Informix to Redis

2. Plane A — Calling Redis Synchronously from SQL

Connection parameters are stored once in `PER_SYSTEM` named memory by `redis_init`, so any trigger or routine server-wide can publish without knowing the configuration. Each command runs over a short-lived connection guarded by a process-global failure cooldown (see **Failure handling** below).

Table 2 — Plane-A UDR surface (redis.so)

Function	Redis	Returns	Purpose
`redis_init(host,port,pass)`	—	lvarchar	Store endpoint in PER_SYSTEM memory (HANDLESNULLS).
`redis_getConfig` / `redis_getVersion` / `redis_isReady`	PING	lvarchar / int	Introspection and connectivity.
`redis_publish(channel,msg)`	PUBLISH	lvarchar	Pub/sub fan-out of an event.
`redis_lpush` / `redis_rpush(key,val)`	LPUSH/RPUSH	lvarchar	Durable work/event queues.
`redis_set` / `redis_get` / `redis_setex(k,v,ttl)` / `redis_del(k)`	SET/GET/SET EX/ DEL	lvarchar/int	Cache-aside reads/writes with TTL.
`redis_incr(key,by)`	INCRBY	lvarchar	Atomic counter / hi-lo id (new value).
`redis_lock(name,ttl_ms)` / `redis_unlock(name,token)`	SET NX PX + Lua	lvarchar / int	Distributed lock with a fence token.
`redis_rate_allow(key,limit,window)`	Lua INCR+EXPIRE	int (1/0)	Fixed-window rate limiter.
`redis_idempotent(key,ttl)`	SET NX EX	int (1/0)	Dedup retries (1 first-seen, 0 duplicate).
`redis_command(line)`	any	lvarchar	Generic escape hatch.

3. Plane B — Streaming Row Changes to Redis (CDC)

The CDC bridge has three parts: an in-engine **control plane** (SQL), the external **agent** (ESQL/C), and Redis. Operators drive capture entirely from SQL — store a JSON plan, flip it RUNNING/STOPPED, read status — while the agent reconciles to it.

Table 3 — CDC control-plane functions

Function	Effect
`cdc_plan_set(name, json)`	Store/replace the capture plan; bump the generation the agent re-reads.
`cdc_plan_get(name)`	Return the stored plan JSON.
`cdc_start(name)` / `cdc_stop(name)`	Set desired-state RUNNING / STOPPED.
`cdc_status(name)`	Return a status JSON (agent state, session, heartbeat, counters; "stale" if old).

The agent opens a CDC session against `syscdcv1`, enables full-row logging, reads the change stream with `ifx_lo_read`, decodes each record, and publishes a Debezium-style envelope to the table's Redis target (stream by default, or list / pub-sub). `columns:"*"` auto-expands to the table's capturable columns. Only the UPDATE after-image is published (op `u`).

3.1. The verified CDC wire format

Two facts about the raw CDC records were established empirically on IDS 15.0 and differ from older sample code:

- **Mixed byte order** — column-data integers are big-endian (the canonical log format, read with `ldlong`), but the variable-length-column size array is native little-endian. Reading the size array big-endian sends the data pointer out of bounds.
- **Record-type codes are version-specific** — read from `syscdcrectypes`, not a hardcoded enum. On 15.0: INSERT=40, DELETE=41, UPDBEF=42, UPDAFT=43, BEGINTX=1, COMMTX=2, TABSCHEMA=200, TIMEOUT=201.

Column types/sizes are resolved from the TABSCHEMA record via a temp-table `DESCRIBE`. The decoder handles INTEGER, SMALLINT, BIGINT/INT8, DECIMAL/MONEY (lossless fixed-scale string), CHAR, VARCHAR/NVARCHAR, DATETIME (ISO-8601) and NULL (JSON null). The session opens with `max_recs=1` so one read yields one record.

4. A Complete Worked Example — Order Pipeline

This worked example shows both planes on one `orders` table: a trigger that synchronously invalidates a cache and emits a notification (Plane A), and a CDC stream that durably feeds fulfilment workers (Plane B). Every step below is reproducible and is exactly what the project's JUnit reference suite asserts.

4.1. 1. Point the blade at Redis (once)

```
EXECUTE FUNCTION redis_init('redis', 6379, NULL);  -- host, port, password (NULL = none)
EXECUTE FUNCTION redis_isReady();                    -- connectivity probe (PING)
EXECUTE FUNCTION redis_getConfig();                  -- show the stored endpoint
```

```
 redis_init      -> OK
 redis_isReady   -> 1
 redis_getConfig -> Redis config: host=[redis] port=6379 auth=no
```

4.2. 2. The table (logged — CDC requires it)

```
CREATE DATABASE redisdb WITH LOG;                      -- CDC can only capture from
 a logged database
CREATE TABLE orders (
    order_id  INTEGER,
    customer  VARCHAR(40),
    status    VARCHAR(16),
    total     DECIMAL(12,2),
    created   DATETIME YEAR TO SECOND
);
```

4.3. 3. Plane A — synchronous cache invalidation + event from a trigger

On every update the trigger deletes the cached order and publishes a notification. Because these run inside the statement, the cache can never be left stale by forgetful app code.

```
CREATE TRIGGER trg_orders_upd UPDATE ON orders
  REFERENCING NEW AS n
  FOR EACH ROW (
    EXECUTE FUNCTION redis_del('cache:order:' || n.order_id),
    EXECUTE FUNCTION redis_publish('events:orders', n.order_id || ':' || n.status)
  );

-- a worker, or another statement, can also coordinate via Plane A:
SELECT redis_idempotent('order-import:' || :batch_ref, 86400);    -- 1 first time, 0
 duplicate
SELECT redis_rate_allow('rl:orders:' || :tenant, 1000, 60);      -- 1 allowed, 0 over-
limit, -1 Redis down
```

4.4. 4. Plane B — define and start the CDC capture plan

```
EXECUTE FUNCTION cdc_plan_set('orders-stream',
  '{"plan":"orders-stream","database":"redisdb","read":{"timeout_s":2},"tables":[
     {"owner":"informix","table":"orders","columns":"*",
      "target":{"type":"stream","key":"cdc:redisdb.orders"}}]}');
EXECUTE FUNCTION cdc_start('orders-stream');
```

```
# start the agent (it reconciles to the plan and begins capturing)
docker compose --profile cdc up -d redis-cdc
```

Confirm it is live before generating changes:

```
EXECUTE FUNCTION cdc_status('orders-stream');
```

```
{
 "plan":"orders-stream",
 "agent_state":"RUNNING",
 "session_id":42467367,"generation":2,
  "heartbeat_utc":1781450265,
  "stale":false,
  "last_seq":0,
  "rows_total":0,
  "tx_total":0,
  "last_error":""
  }
```

4.5. 5. Make changes — and watch them appear on the stream

The agent publishes one envelope per row change to `cdc:redisdb.orders`; the `result` below is the `payload` field of each entry, as read with `XRANGE cdc:redisdb.orders - +`.

```
INSERT INTO orders VALUES (1001, 'ACME Corp', 'NEW', 250.00, DATETIME(2026-06-14
 10:30:00) YEAR TO SECOND);
UPDATE orders SET status = 'SHIPPED' WHERE order_id = 1001;
DELETE FROM orders WHERE order_id = 1001;
```

```
{"op":"c","source":{"db":"redisdb","owner":"informix","table":"orders"},
                   "data":{"order_id":1001,"total":"250.00","created":"2026-06-14
 10:30:00","customer":"ACME Corp","status":"NEW"}}
{"op":"u","source":{"db":"redisdb","owner":"informix","table":"orders"},
                    "data":{"order_id":1001,"total":"250.00","created":"2026-06-14
 10:30:00","customer":"ACME Corp","status":"SHIPPED"}}
{"op":"d","source":{"db":"redisdb","owner":"informix","table":"orders"},
                    "data":{"order_id":1001,"total":"250.00","created":"2026-06-14
 10:30:00","customer":"ACME Corp","status":"SHIPPED"}}
```

Note the column order is fixed-length first then variable-length (the TABSCHEMA order), `DECIMAL` is a lossless fixed-scale string, and `DATETIME` is ISO-8601.

4.6. 6. Consume durably with a Redis Streams consumer group

```
XGROUP CREATE cdc:redisdb.orders fulfilment $ MKSTREAM
XREADGROUP GROUP fulfilment worker-1 COUNT 100 BLOCK 5000 STREAMS cdc:redisdb.orders >
-- process each entry's payload JSON, then acknowledge:
XACK cdc:redisdb.orders fulfilment <id>
```

Consumers must be **idempotent** (dedupe on `source` identity) because delivery is at-least-once — which is exactly what `redis_idempotent` (Plane A) is for.

4.7. Reference scenarios (the test suite)

The JUnit suite asserts the JSON for six real workloads, each also stressing an extreme of the wire format:

Table 4 — CDC reference scenarios

Scenario	Real case	Extreme asserted
Retail order lifecycle	order pipeline feeding fulfilment workers	INSERT/UPDATE/DELETE → c/u/d
Banking ledger	financial audit stream	fixed-scale DECIMAL (`99999999999999.99` , `-0.01`), ISO datetime
Telecom 64-bit ids	subscriber / CDR provisioning	BIGINT up to `2^63-1`
Customer master text	arbitrary user input	quotes/backslashes/newlines/ unicode → valid, injection-safe JSON
NULL handling	optional fields	NULL → JSON null
Bulk transaction	batch import / end-of-day posting	100 rows in one transaction, all captured

5. Failure Handling and Recovery

Both planes are designed so that a Redis outage degrades gracefully and self-heals, and so a crash of any component loses no committed change. The behaviour differs by plane.

5.1. Plane A — synchronous UDRs

- **Redis unreachable.** A connect attempt is bounded to **1.5 s** (1 s + 500 ms), tried 3 times. After a failure a process-global **5-second cooldown** is set: further calls short-circuit and return immediately without paying the connect timeout, so a Redis outage does **not** add latency to every statement.
- **The transaction is never rolled back by the UDR.** Plane A is best-effort: a failed Redis call returns an error string (or the `-1` sentinel) but does not abort the user's transaction. This is deliberate — coupling commits to an external system is the risk Plane B exists to avoid.
- **Fail-open vs fail-closed is the caller's choice.** The integer-returning functions (`redis_del`, `redis_idempotent`, `redis_rate_allow`, `redis_unlock`) return **-1** when Redis is unreachable, distinct from a legitimate **0**. So a rate limiter can fail open (`WHERE redis_rate_allow(...) <> 0`) or fail closed (`= 1`) explicitly.
- **Recovery is automatic.** When Redis returns, the cooldown expires and the next call reconnects; no operator action is needed. `redis_init` also clears the cooldown so a reconfigure takes effect immediately.
- **Engine restart.** The endpoint lives in PER_SYSTEM memory, which is lost on engine shutdown — run `redis_init` again at startup (e.g. from a scheduled task).

5.2. Plane B — the CDC agent

- **Redis unreachable.** The agent keeps reading the CDC stream but publish fails; it does not advance its position past an unpublished change, so nothing is lost. When Redis returns it resumes. (Because delivery is at-least-once, a change may be re-published after a mid-publish crash — consumers dedupe.)
- **Agent crash / restart.** On restart the agent re-opens the CDC session and resumes. **Caveat:** the persistent LSN checkpoint is a planned refinement; today a restart resumes from the current log position, so changes made while the agent was down between sessions are not back-filled. Run the agent under a supervisor (the container restart policy) to minimise the gap.
- **Logical-log retention.** While the agent is down, the changes it has not read remain in the logical logs. A long outage can pin logical-log space — monitor it exactly as you would for Enterprise Replication, and size the logs for the worst-case agent downtime.
- **Ungraceful shutdown leaves a flag.** Full-row logging is a **persistent table attribute**. The agent disables it on graceful shutdown (SIGTERM → `cdc_set_fullrowlogging(table,0)` + `cdc_closesess`); if the agent is killed with SIGKILL the flag remains and the table cannot be dropped (error 19816) until you run `cdc_set_fullrowlogging('db:owner.table', 0)` from a `syscdcv1` client. Always stop the agent gracefully.
- **Engine restart.** CDC sessions are in-memory and are lost; the agent reconnects and re-opens automatically.

6. Logging and Monitoring

The blade is observable at three levels — the engine's `online.log`, the agent's own status, and an optional trace class — each at a deliberately **moderate** volume (session-level events, never per-row, so a busy capture does not flood the logs).

6.1. online.log (the engine)

Informix records the CDC **session lifecycle** and full-row-logging changes in `online.log`. You see one entry when a capture session starts (with the LSN it snoops from), one when it ends, and one per table when full-row logging is toggled — but **not** a line per captured row. Typical entries from a real run:

```
grep -E 'CDC|FRL' $INFORMIXDIR/online.log | tail
```

```
CDC: Log Reader session 42467367 starting to snoop at the CURRENT position,
 LSN(7,96b000), page 2411.
CDC: Log Reader started.  CDC session id 42467367. SQL session id 81.
CDC: Log Reader ended.    CDC session id 42467367. SQL session id 81.
FRL: ... setting FULL row logging for table redisdb:informix.orders
```

> So: yes, capture activity is visible in `online.log`, but at the **order of magnitude of sessions and tables**, not rows. Frequent agent restarts will show as repeated "Log Reader started/ended" pairs — a useful signal that the agent is flapping.

6.2. Agent status (the operational source of truth)

The agent writes a heartbeat and counters to `redis_cdc_status` every cycle and mirrors a compact status to the Redis key `cdc:status:<plan>`. Query it from SQL with `cdc_status(plan)` (returns JSON with `agent_state`, `session_id`, `heartbeat_utc`, `rows_total`, `tx_total`, `last_error`, and a `stale` flag when the heartbeat is older than 30 s). The built-in `onstat -g cdc` shows the live session and the tables it is capturing.

6.3. The redis trace class (deep debugging)

The synchronous UDRs and the agent emit detailed traces only when explicitly enabled, so normal operation is quiet. Turn the `redis` trace class on to a file for diagnosis; the agent additionally accepts a `-v` / `--verbose` argument for per-record byte dumps (off by default).

```
EXECUTE PROCEDURE trace_on('/tmp/redis.log', 'redis 10');   -- verbose; turn off after
 diagnosis
```

7. Compilation and Registration

The blade is built from C sources and registered as UDRs. There are two compilable artefacts: the in-engine **UDR shared object** (redis.so) and the external **CDC agent** (redis_cdc). In this project the UDR is compiled **on the Informix server** (InformixBladeModule streams the source and invokes gcc through an SPL SYSTEM call, then runs the install SQL); the steps below are the manual equivalent.

7.1. 1. Compile the in-engine UDR shared object

Standard DataBlade C-UDR compile: position-independent shared object, the Informix public headers on the include path, and the runtime libraries the blade links.

```
gcc -O3 -fPIC -shared redis.c \
    -I$INFORMIXDIR/incl/public \
    -lm -lhiredis \
    -o redis.so
```

Table 5 — Libraries and flags — redis.so

Item	Why
-fPIC -shared	Build a dynamically loadable shared object (required for EXTERNAL NAME).
-I$INFORMIXDIR/incl/public	DataBlade API headers (mi.h , milib.h).
-lhiredis	The Redis C client. Install hiredis on the server (e.g. build from source into /usr/lib64 + /usr/include/hiredis).
-lm	libm, linked by the DataBlade convention.
-O3 (opt. -march=native)	Optimisation; -march=native only on the exact CPU that will run the engine.

The shared Redis core lives in the header redis_publish.h , which redis.c includes; the server-side compiler inlines it so a single self-contained source is sent to gcc.

7.2. 2. Register the routines

Each entry point is registered with CREATE FUNCTION … EXTERNAL NAME , where the <sopath> placeholder (the %s in the install SQL) is the directory of the compiled redis.so . The password argument of redis_init is legitimately NULL, hence WITH (HANDLESNULLS) ; the messaging / cache functions keep the default VARIANT marking because every call mutates Redis and must not be folded or cached by the optimiser. The complete Plane-A surface is below — one CREATE per entry point in redis_install.sql .

```
-- ============================================================================
-- redis.so — complete registration  (this is redis_install.sql)
-- (replace <sopath> / %s with the server-side directory containing redis.so)
-- ============================================================================

-- Configuration / health
CREATE FUNCTION redis_init(LVARCHAR, INTEGER, LVARCHAR) RETURNING LVARCHAR
    WITH (HANDLESNULLS) EXTERNAL NAME '<sopath>/redis.so(redis_init)' LANGUAGE C;
CREATE FUNCTION redis_getConfig() RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_getConfig)' LANGUAGE C;
CREATE FUNCTION redis_getVersion() RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_getVersion)' LANGUAGE C;
CREATE FUNCTION redis_isReady() RETURNING INTEGER
    EXTERNAL NAME '<sopath>/redis.so(redis_isReady)' LANGUAGE C;

-- Messaging / queues / cache
CREATE FUNCTION redis_publish(LVARCHAR, LVARCHAR) RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_publish)' LANGUAGE C;
CREATE FUNCTION redis_lpush(LVARCHAR, LVARCHAR) RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_lpush)' LANGUAGE C;
CREATE FUNCTION redis_rpush(LVARCHAR, LVARCHAR) RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_rpush)' LANGUAGE C;
CREATE FUNCTION redis_set(LVARCHAR, LVARCHAR) RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_set)' LANGUAGE C;
CREATE FUNCTION redis_get(LVARCHAR) RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_get)' LANGUAGE C;

-- Plane-A coordination & cache primitives
-- (redis_del / redis_idempotent / redis_rate_allow / redis_unlock return INTEGER;
--  sentinel -1 = Redis unreachable, so SQL can choose fail-open vs fail-closed)
CREATE FUNCTION redis_setex(LVARCHAR, LVARCHAR, INTEGER) RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_setex)' LANGUAGE C;
CREATE FUNCTION redis_del(LVARCHAR) RETURNING INTEGER
    EXTERNAL NAME '<sopath>/redis.so(redis_del)' LANGUAGE C;
CREATE FUNCTION redis_incr(LVARCHAR, INTEGER) RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_incr)' LANGUAGE C;
CREATE FUNCTION redis_idempotent(LVARCHAR, INTEGER) RETURNING INTEGER
    EXTERNAL NAME '<sopath>/redis.so(redis_idempotent)' LANGUAGE C;
CREATE FUNCTION redis_rate_allow(LVARCHAR, INTEGER, INTEGER) RETURNING INTEGER
    EXTERNAL NAME '<sopath>/redis.so(redis_rate_allow)' LANGUAGE C;
CREATE FUNCTION redis_lock(LVARCHAR, INTEGER) RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_lock)' LANGUAGE C;
CREATE FUNCTION redis_unlock(LVARCHAR, LVARCHAR) RETURNING INTEGER
    EXTERNAL NAME '<sopath>/redis.so(redis_unlock)' LANGUAGE C;

-- Generic escape hatch — any Redis command from SQL (admin, EXPIRE, INCR, …)
CREATE FUNCTION redis_command(LVARCHAR) RETURNING LVARCHAR
    EXTERNAL NAME '<sopath>/redis.so(redis_command)' LANGUAGE C;

-- Debug helper (enables the "redis" trace class to a file)
CREATE PROCEDURE trace_on(LVARCHAR, LVARCHAR)
    EXTERNAL NAME '<sopath>/redis.so(trace_on)' LANGUAGE C;
```

The CDC control plane is plain SPL (no C) installed from `cdc_control_install.sql`: the `redis_cdc_control` / `redis_cdc_status` tables and the `cdc_plan_set` / `cdc_plan_get` / `cdc_start` / `cdc_stop` / `cdc_status` functions.

7.3. 3. Build the external CDC agent (ESQL/C)

The agent cannot be a UDR, so it is a standalone client compiled with the IBM **Client SDK (ESQL/C)** and linked with hiredis (plus a vendored single-file JSON parser). The IBM installer requires a Java 11+ VM, `bc`, and

the `file` utility; ESQL/C binaries need the client libraries on the loader path (`$INFORMIXDIR/lib` , `/lib/esql` , `/lib/dmi`).

```
# CSDK silent install (one-time, in the build image)
./installclientsdk -i silent -f csdk.properties -DLICENSE_ACCEPTED=TRUE

# build the agent: esql preprocesses the .ec, then gcc compiles + links
esql -O3 -o redis_cdc redis_cdc.ec cJSON.c -lhiredis
```

Prerequisites on the source database: it must be created `WITH LOG` , and the `syscdcv1` database must exist (created once by `$INFORMIXDIR/etc/syscdcv1.sql`). The agent then needs no further build steps — it is driven entirely from SQL via the control plane.

> **When not to call Redis synchronously.** Plane A couples the calling statement to Redis availability (bounded by the connect timeout + cooldown, but never zero). For strict consistency, or where an outage must not touch the write path at all, prefer Plane B (CDC) or the outbox pattern — let the change stream update Redis asynchronously.

7.4. 4. Run the CDC agent

The agent is **not** a UDR and is **not** registered with `CREATE FUNCTION` — it is a standalone `main()` program, a long-running daemon you launch as an OS process next to the engine. It is configured **entirely with command-line arguments** (no environment variables — run `redis_cdc --help` for the full list) and then driven from SQL via the control plane. On start it opens two named SQL connections — `app` → `<appdb>@<server>` (the `redis_cdc_control` / `redis_cdc_status` tables) and `cdc` → `syscdcv1@<server>` (the Informix CDC API) — then enters a poll loop: it reads the plan row, and while its state is `RUNNING` it streams the captured row changes to Redis, re-starting capture whenever the plan's generation changes.

Table 6 — Agent command-line arguments

Argument	Default	Purpose
`-s, --server NAME`	— (required)	Engine name (resolved via the sqlhosts file); both connections target `…@` this server and it is passed to `cdc_opensess` .
`-d, --appdb NAME`	`redisdb`	Application database to capture from (holds the control/status tables); also the `<db>` in the default `cdc:<db>.<table>` Redis key.
`-n, --plan NAME`	`orders-stream`	Which plan in `redis_cdc_control` the agent reconciles.
`-u, --db-user USER` / `-w, --db-pass PASS`	`informix` / `informix`	Credentials for both connections (authenticated `USER … USING …` over TCP).
`-t, --read-timeout SECS`	`5`	Seconds the CDC read blocks / the idle poll sleeps between control checks.
`-H, --redis-host HOST` / `-P, --redis-port PORT` / `-a, --redis-pass PASS`	`redis` / `6379` / —	Target Redis endpoint the decoded changes are written to (LPUSH /

Argument	Default	Purpose
		PUBLISH / XADD per the plan's target type).
`-q, --sqlhosts PATH`	inherit env	sqlhosts file to use; sets `INFORMIXSQLHOSTS` for the Client SDK before connecting.
`-v, --verbose`	off	Emit verbose per-record decode tracing to stderr.

Only the Client SDK install location stays in the environment (`INFORMIXDIR` and the loader path, which the ESQL/C runtime needs before `main()` runs); everything else — including the sqlhosts file, via `--sqlhosts` — is a command-line argument. Run the binary directly, or as the supplied container (the image's `ENTRYPOINT` is `/opt/agent/redis_cdc` , so compose / `docker run` arguments flow straight through):

```
# Directly, as an OS process next to the engine:
./redis_cdc --server redis_ifx --sqlhosts $INFORMIXDIR/etc/sqlhosts \
            --appdb redisdb --plan orders-stream \
            --redis-host redis --redis-port 6379
#   -> redis_cdc: server=redis_ifx plan=orders-stream appdb=redisdb redis=redis:6379

# Or as the sidecar container (started under the 'cdc' compose profile):
docker compose --profile cdc up -d redis-cdc
```

Starting the process only connects and begins polling — nothing is captured until a plan is defined and started **from SQL**: `cdc_plan_set('orders-stream', ...)` then `cdc_start('orders-stream')` flips the control row to `RUNNING` , which the agent picks up on its next poll; `cdc_stop(...)` flips it back. Progress and liveness are observable in `redis_cdc_status` (see **Logging and Monitoring**).

> **Stop it gracefully.** The agent traps `SIGTERM` / `SIGINT` (so `docker stop` works) and on exit runs its teardown — `endcapture` + `closesess` — to release the server-side CDC session. An orphaned session keeps full-row logging on and **blocks** `DROP TABLE` on the captured tables, so avoid `kill -9` .

8. Glossary — Redis and Change-Data-Capture Terms

Table 7 — Glossary

Term	Meaning
Plane A / Plane B	The two integration planes: synchronous in-engine UDR call-outs (A) and the asynchronous CDC change stream (B).
UDR	User-Defined Routine — a C function callable from SQL; here the `redis_*` functions registered with EXTERNAL NAME.
PER_SYSTEM named memory	Informix shared-memory segment that survives across sessions; holds the Redis endpoint set by `redis_init`.
Cooldown	A process-global short window after a failed Redis connect during which calls short-circuit, so an outage does not add a connect timeout to every statement.
Sentinel (-1)	The value the integer UDRs return when Redis is unreachable, distinct from a real 0 — lets SQL choose fail-open vs fail-closed.
Fence token	The unique token `redis_lock` returns; `redis_unlock` only releases the lock if the token still matches (compare-and-delete).
Idempotency key	A `SET NX EX` key used to dedup retried work (1 = first-seen, 0 = duplicate).
CDC	Change Data Capture — reading committed row changes from the logical logs via the Informix CDC API.
CDC session	An `cdc_opensess` capture session; data is read from it with `ifx_lo_read`. Visible via `onstat -g cdc`.
Full-row logging	A persistent per-table attribute (`cdc_set_fullrowlogging`) that makes the whole row available to CDC; blocks DROP TABLE until disabled (error 19816).
syscdcv1	The system database holding the CDC API functions; the agent connects to it to capture.
TABSCHEMA record	The CDC record describing a captured table's columns; the agent resolves column types from it before decoding rows.
Change envelope	The Debezium-style JSON the agent publishes: `op` (c/u/d), `source` (db/owner/table), and `data` (the row).
Redis Stream	An append-only, replayable log (`XADD`); the default CDC target. Consumed durably via consumer groups (`XREADGROUP` / `XACK`).
At-least-once	The CDC delivery guarantee — a change may be re-published after a crash, so consumers must be idempotent.
Control plane	The `redis_cdc_control` / `redis_cdc_status` tables + `cdc_*` functions that drive and monitor the agent from SQL.

Term	Meaning
hiredis	The C Redis client library both planes link; the shared core lives in `redis_publish.h`.

9. License, Trademarks and Disclaimer

This DataBlade 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. **Redis** is a trademark of Redis Ltd. 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.