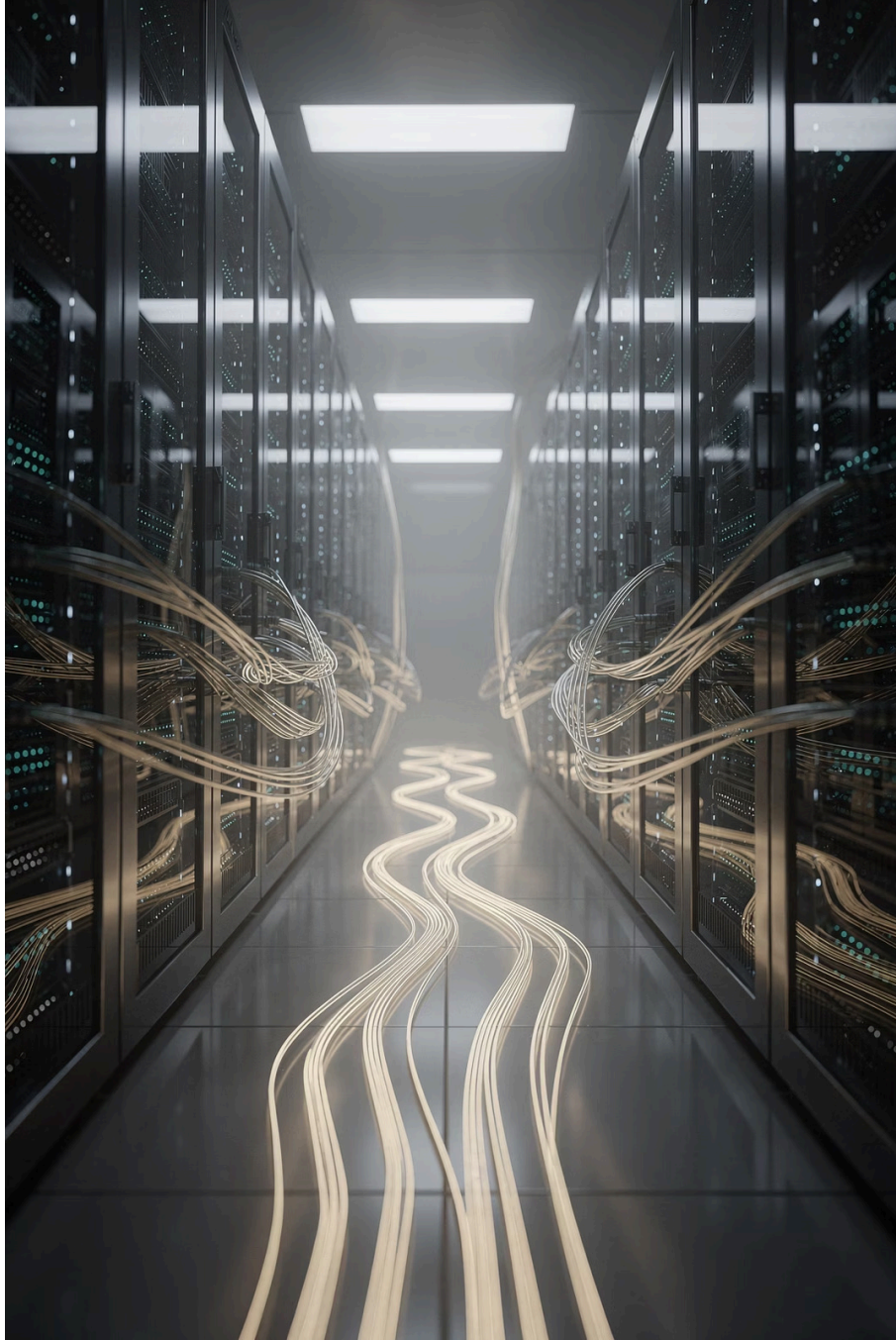
CDC in a Hybrid World

From Data Movement to Data Coordination — PostgreSQL Edition

A practical framework for hybrid Postgres environments that actually stay consistent

PostgresConf San Jose 2026

Presented by Amit Parikh & Phil Rodas



THE REALITY

Hybrid Isn't Temporary. It's the Default.

Every org running PostgreSQL at scale is also running something else. That's not a failure of strategy — it's the reality of enterprise data architecture. PostgreSQL owns the transactional core: high write velocity, rich schemas, referential integrity. The rest of the stack fills in around it.

🐘 PostgreSQL

High write velocity, schema-rich, WAL-native CDC, referential integrity at core

🔷 Oracle

Legacy OLTP, stored procs, owned by compliance teams, slow to migrate

❄️ Snowflake

Analytical workloads, semi-structured data, no native transactional guarantees

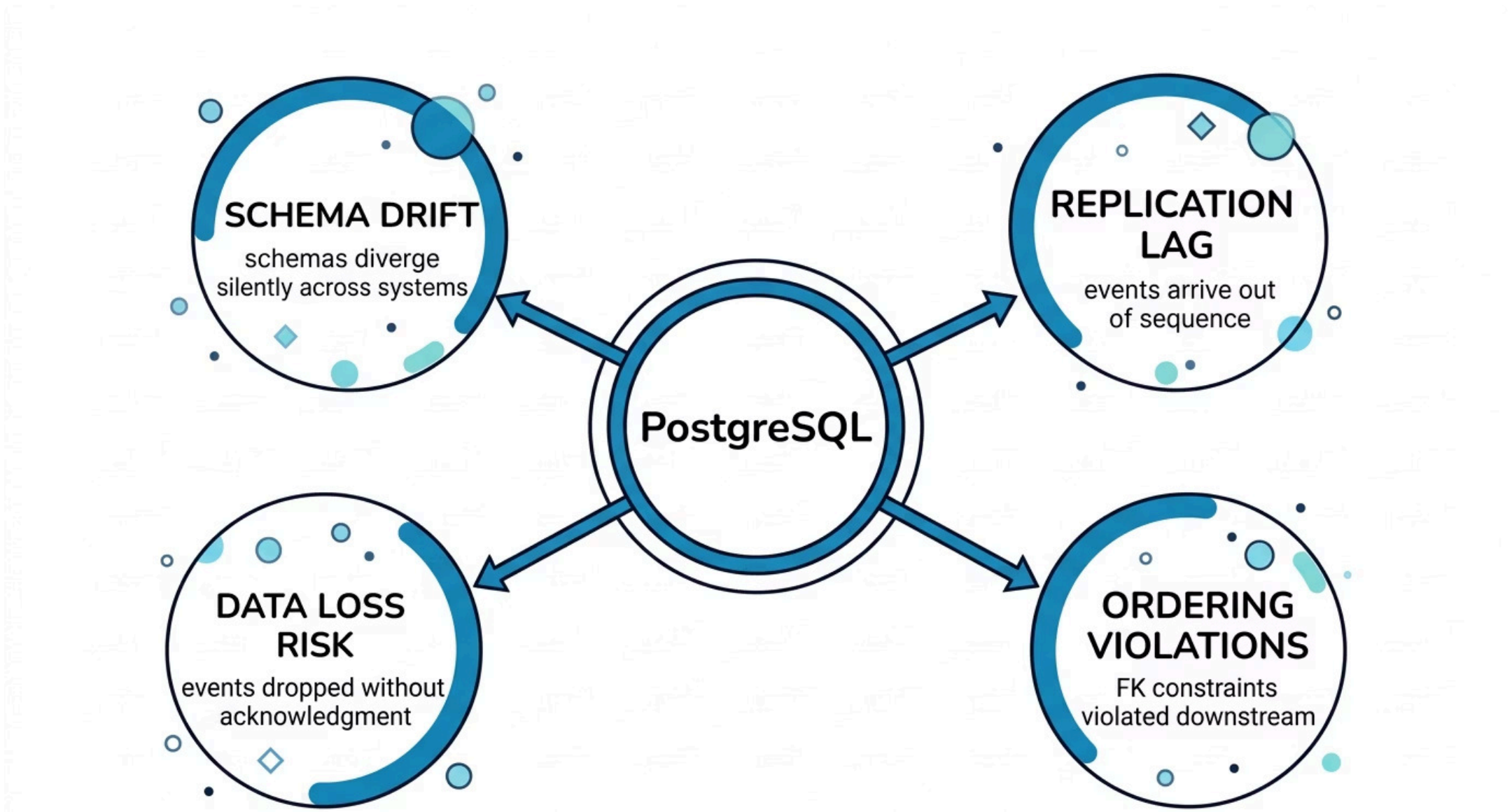
🔥 Databricks

Lakehouse ingestion, Delta Lake, ML pipelines — eventual consistency by design

CORE PROBLEM

This Isn't a Migration Problem — It's a Coordination Problem

You're not moving data from A to B once. You're keeping A, B, C, and D consistent *while they all keep changing*. PostgreSQL sits in the middle — and the coordination failures are well-documented, painful, and entirely predictable.



Key insight: The problem space is not transport. It's ordering, consistency, and coordination across heterogeneous systems that were never designed to talk to each other.

NATIVE REPLICATION

Native Replication Works... Until It Doesn't

✅ The Happy Path

Inside PostgreSQL

- Logical replication via `pg_logical`
- WAL stream with full row-level events
- Native slot management, LSN tracking
- Schema-aware, transactionally consistent
- Works beautifully PG → PG

❌ Where It Breaks

Crossing the boundary

- **PG → Snowflake:** Type coercions, JSONB explosions, no FK enforcement
- **PG → Oracle:** Sequence conflicts, schema mismatches, stored proc drift
- **PG → Lakehouse:** Ordering not guaranteed, late-arriving events, schema-on-read chaos
- Every hop is a new dialect. Native tools don't translate.

KEY INSIGHT

Native Solves Sameness. Hybrid Requires Translation.

> "You don't need more replication. You need a coordination layer that speaks every dialect."

Homogeneous Postgres Stack

- Logical replication is sufficient
- WAL slots give you ordering guarantees
- Schema changes propagate cleanly
- One dialect, one contract

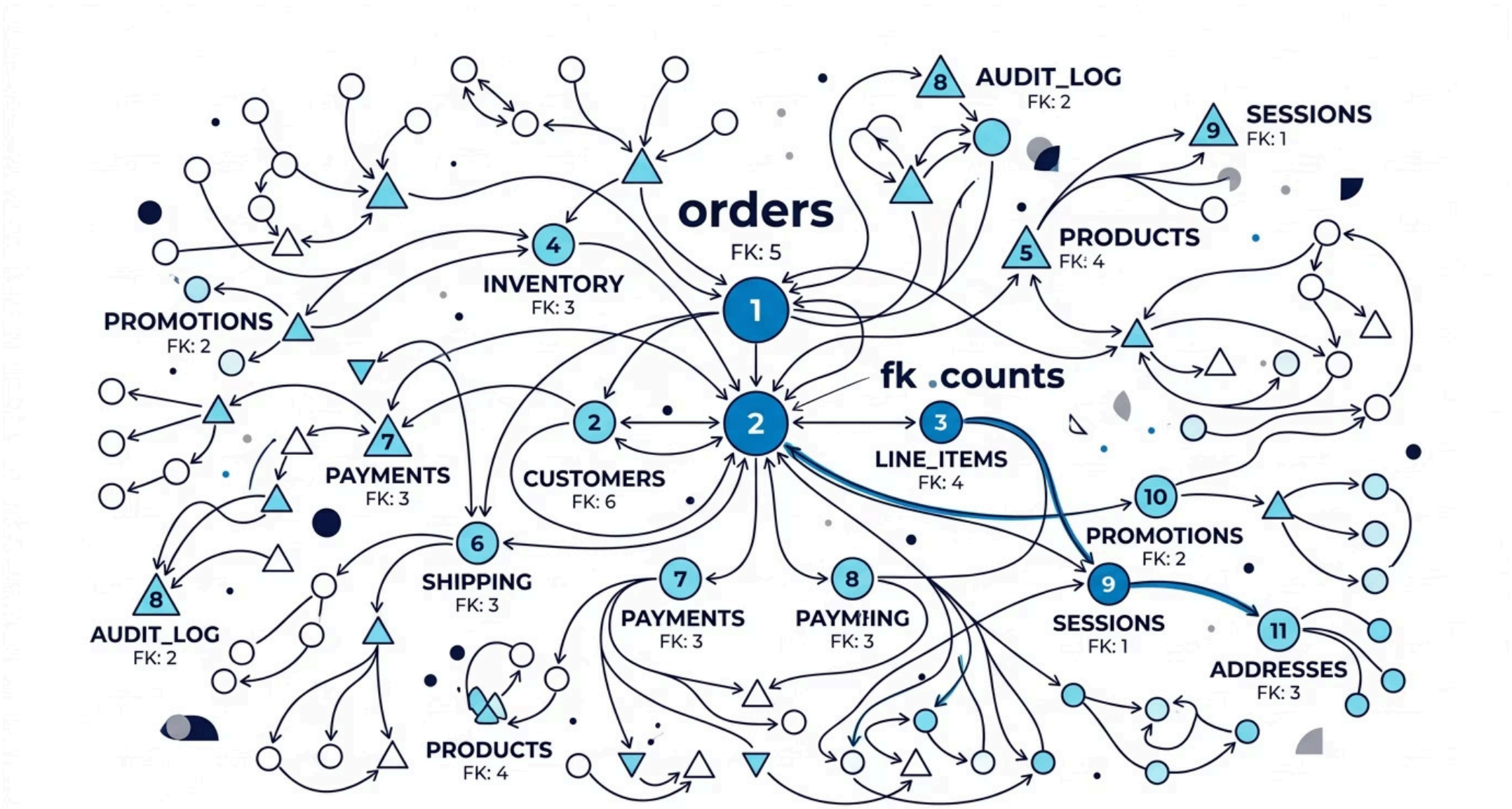
Hybrid Reality

- Every target system has different semantics
- Type systems collide at every boundary
- Ordering guarantees evaporate mid-hop
- You need a coordination contract, not a pipe

SCHEMA REALITY

This Is Your Postgres System

Not a clean ERD. Not a textbook diagram. A dense, interdependent graph of tables, foreign keys, sequences, triggers, and implicit relationships that **nobody fully understands anymore** — including the team that built it.



🐘 **DBA reality:** Real schemas are dense, interdependent, and non-obvious. You cannot reason about migration risk without the full dependency graph. `pg_depend` doesn't lie — your ERD does.

LIVE DEMO

Scoring Your Postgres Schema in Real Time

Run this directly against any production PostgreSQL instance. Every table gets a **Risk Index from 0–100** based on write velocity, FK depth, and column complexity. No extensions required.

```
WITH RECURSIVE
-- All FK edges: child → parent
fk_edges AS (
  SELECT DISTINCT
    conrelid::regclass::text  AS child_table,
    confrelid::regclass::text AS parent_table
  FROM pg_constraint
  WHERE contype = 'f'
),

-- Recursive FK depth: roots = 0, each child = max(parent depth) + 1
fk_chain AS (
  -- Base: root tables (appear in pg_stat but are never an FK child)
  SELECT relname AS table_name, 0 AS depth
  FROM pg_stat_user_tables
  WHERE schemaname = 'public'
    AND relname NOT IN (SELECT child_table FROM fk_edges)

  UNION ALL

  -- Recursive: walk child → parent edges
  SELECT e.child_table, fc.depth + 1
  FROM fk_edges e
  JOIN fk_chain fc ON e.parent_table = fc.table_name
  WHERE fc.depth < 20          -- safety limit
),

fk_depth AS (
  SELECT table_name, MAX(depth) AS fk_depth
  FROM fk_chain
  GROUP BY table_name
),

-- Write velocity (use plain relname so it matches fk_depth)
table_stats AS (
  SELECT
    relname AS table_name,
    GREATEST(1,
      (COALESCE(n_tup_ins,0) + COALESCE(n_tup_upd,0) + COALESCE(n_tup_del,0))::numeric /
      GREATEST(1,
        EXTRACT(EPOCH FROM (CURRENT_TIMESTAMP -
          COALESCE(
            GREATEST(last_analyze, last_autoanalyze),
            CURRENT_TIMESTAMP - INTERVAL '7 days'
          )
        )) / 60
      )
    ) AS writes_per_minute
  FROM pg_stat_user_tables
  WHERE schemaname = 'public'
),

-- Column complexity (use plain table_name)
column_complexity AS (
  SELECT
    table_name,
    COUNT(CASE
      WHEN data_type IN ('json','jsonb')
       OR udt_name  IN ('json','jsonb')
       OR data_type LIKE '%[]'
       OR (data_type = 'text' AND character_maximum_length IS NULL)
      THEN 1
    END) AS complexity_score
  FROM information_schema.columns
  WHERE table_schema = 'public'
  GROUP BY table_name
),

scored AS (
  SELECT
    COALESCE(ts.table_name, fk.table_name, cc.table_name) AS table_name,
    ROUND(COALESCE(ts.writes_per_minute, 0))   AS write_velocity,
    COALESCE(fk.fk_depth, 0)                   AS fk_depth,
    COALESCE(cc.complexity_score, 0)           AS column_complexity,
    ROUND(
      (0.45 * LEAST(100, COALESCE(ts.writes_per_minute, 0) / 10.0)) +
      (0.35 * LEAST(100, COALESCE(fk.fk_depth, 0) * 20)) +
      (0.20 * LEAST(100, COALESCE(cc.complexity_score, 0) * 12))
    ) AS risk_index
  FROM table_stats ts
  FULL OUTER JOIN fk_depth fk ON ts.table_name = fk.table_name
  FULL OUTER JOIN column_complexity cc ON ts.table_name = cc.table_name
)

SELECT
  table_name,
  write_velocity,
  fk_depth,
  column_complexity,
  risk_index,
  CASE
    WHEN risk_index >= 66 THEN 'Critical'
    WHEN risk_index >= 31 THEN 'Moderate'
    ELSE 'Low Risk'
  END AS risk_tier
FROM scored
WHERE table_name LIKE 'spx_%'
ORDER BY risk_index DESC;
```

RISK IDENTIFICATION

Not All Tables Are Equal

The risk-scoring query doesn't return a list of tables. It returns a **triage manifest**. Treat it like one. Your migration strategy should be completely different for each tier.

🔴 Critical — Risk Index 66–100

High write velocity, deep FK chains, complex types. CDC is the only safe path. Think: `orders`, `payments`, `inventory_reservations`.

🟡 Moderate — Risk Index 31–65

Moderate change rates, meaningful dependencies. Batch ETL risks compounding lag. CDC strongly recommended. Think: `customers`, `products`, `sessions`.

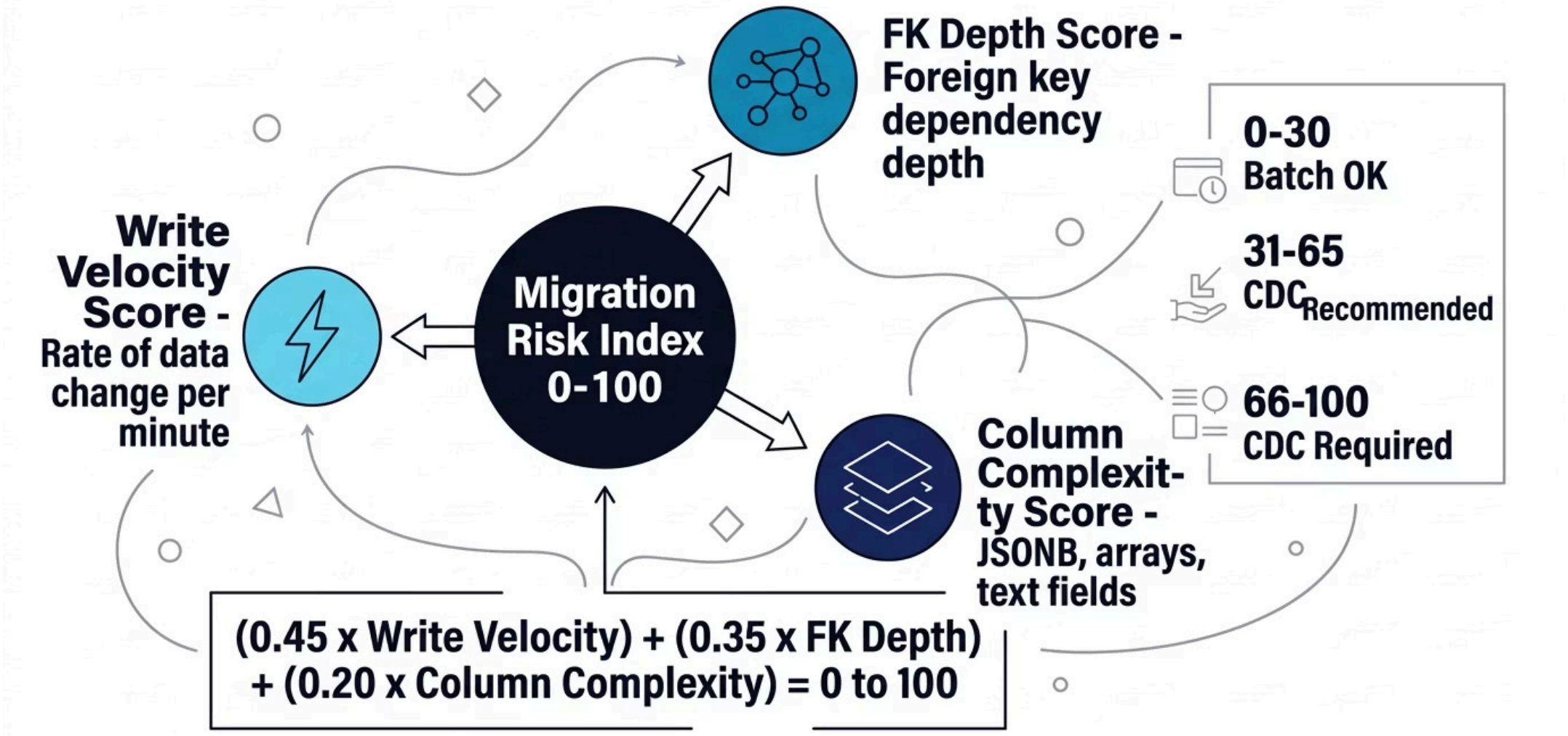
🟢 Low Risk — Risk Index 0–30

Low velocity, shallow dependencies, simple types. Batch migration is fine. Think: `reference_data`, `audit_codes`, `config_flags`.

RISK MODEL

The 3-Axis Risk Scoring Model

Migration risk is multidimensional. A single slow-moving table with 12 FK references is more dangerous than a fast table with none. The weighted formula captures what intuition misses.



Why these weights? Write velocity dominates because it determines how stale your snapshot becomes during migration. FK depth determines blast radius on failure. Complexity determines transformation error probability.

CORE TENSION

The Tables That Matter Most Are Changing While You Move Them

> "The moment you take a snapshot of a production OLTP system, the snapshot is already wrong."

This is the central, uncomfortable truth of hybrid Postgres migration. Your `orders` table writes 400 rows per minute. Your migration window is 6 hours. By the time the bulk load finishes, you have **144,000 uncommitted deltas** sitting in the WAL — unaccounted for, potentially lost.

The Batch Assumption

Assumes the source is frozen. In OLTP systems, it never is.

The WAL Reality

PostgreSQL records every change. CDC reads that record continuously — no snapshot needed.

The Coordination Solution

Keep source and target synchronized at the event level until you're ready to cut over safely.

FAILURE MODES

Where Things Actually Fail in Postgres

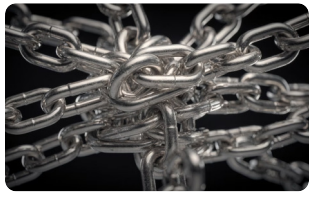
These aren't hypothetical. Every one of these failure modes has a `pg_stat_statements` entry, a 3am page, and a postmortem somewhere in production.

Batch ETL Delay

Active OLTP tables accumulate 48–72 hours of lag during bulk loads. By cutover, you're reconciling days of divergence.

Schema Drift

A column added to `orders` mid-migration. Snowflake doesn't know. Rows load with NULLs. Silent corruption — caught in QA, or by a customer.



FK Ordering Violations

Child rows arrive before parent rows. Referential integrity fails. Snowflake rejects rows. The FK graph predicted it — nobody ran the query.

BUSINESS CASE

Downtime Is a Financial Event

The false economy of skipping CDC in hybrid Postgres migrations.

The Real Numbers

$5,600

Per Minute

Average enterprise downtime cost — Gartner

72hrs

Batch Lag

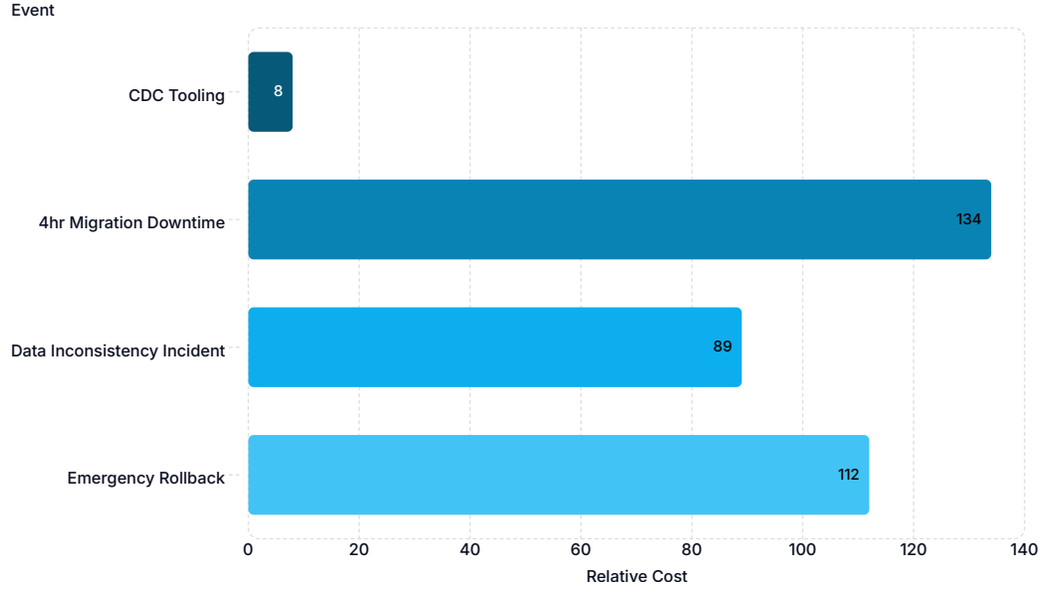
Typical lag on active OLTP systems during batch migration

3–8×

CDC ROI

Return on CDC investment vs. one avoided migration incident

Cost Comparison



CDC tooling cost is noise compared to a single avoided incident. The math isn't close.

OPERATING MODEL

CDC Turns Migration Into an Operating Model

From high-stakes one-time event to **continuous, observable, reversible process**. This is the mindset shift that separates teams that sleep during migrations from teams that don't.

Real-Time Sync | **Observable Lag** | **Safe Cutover**

- Source and target stay synchronized at the event level. No snapshot drift. No reconciliation marathon at cutover. The WAL stream is your continuous ground truth.
- Replication lag becomes a first-class metric. You can see exactly how far behind the target is — in seconds, not assumptions. Alert before the SLA breaks, not after.
- Cutover happens when lag reaches zero — not when the migration window ends. Roll back is a config change, not a disaster recovery exercise. Teams cut over and cut back with confidence.

CAPABILITIES

CDC Powers Much More Than Migration

The same coordination layer unlocks multiple strategic patterns, transforming how you leverage your data.

Event-Driven Architecture

Turn every Postgres change into a business event

Real-Time Analytics

Sub-second freshness for dashboards and BI

ML Feature Pipelines

Low-latency features for models without batch jobs

Compliance & Audit Trails

Immutable, ordered history for regulatory needs

Cross-System Synchronization

Consistent data across Postgres, Oracle, Snowflake, Databricks

Data Lake / Lakehouse Hydration

Reliable, ordered ingestion into Iceberg/Delta/Parquet

CLOSING

Modernization Isn't About Moving Data

It's about keeping it consistent while everything is moving.

What to Do Right Now

1. Run `SELECT * FROM pg_depend` — map your actual dependency graph, not your mental model
2. Execute the 3-axis risk-scoring query against your schema in staging or prod read replica
3. Classify every table into Critical / Moderate / Low — before you design a single pipeline

The Bottom Line

The teams that win hybrid migrations aren't the ones with the fastest pipelines. They're the ones with the **best coordination layer** — one that understands PostgreSQL's WAL, respects FK ordering, and treats schema change as a first-class event, not an afterthought.

> **CDC isn't a tool choice. It's an architectural commitment to consistency.**

REAL-WORLD PERSPECTIVE

What This Looks Like in the Wild

Phil Rodas — 5 minutes on real customer accounts

🐘 20+ Hybrid Postgres Migrations

Lessons learned from production environments spanning Oracle legacy lift-and-shift, Snowflake analytical offload, and Databricks lakehouse ingestion — all with PostgreSQL as the transactional source of truth.

📋 Patterns That Repeat

The same three failure modes appear in virtually every engagement: underestimated write velocity on core OLTP tables, schema drift caught too late, and FK violations surfaced only by downstream reject logs.