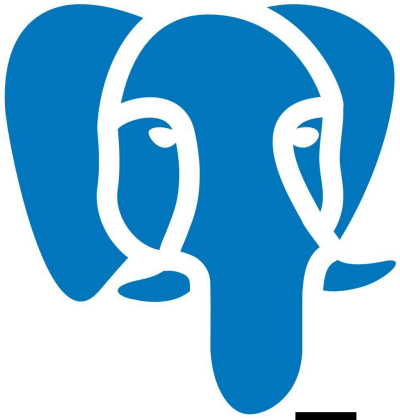
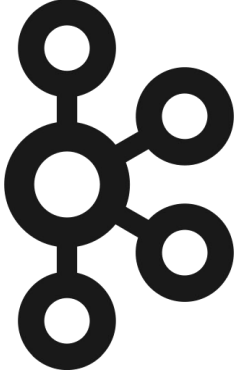
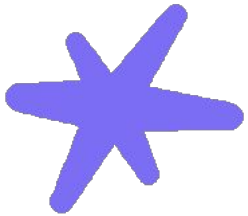






From WAL to Kafka

Turning Postgres Changes into Event Streams



About Narissa Tsuboi



Software Engineer @ Read AI

Stack: Python, AWS, Kafka, Postgres

- MSK Kafka 3.7.x to 4.1.x cluster migration, no downtime or data loss
- Integration Status Management using Change Data Capture

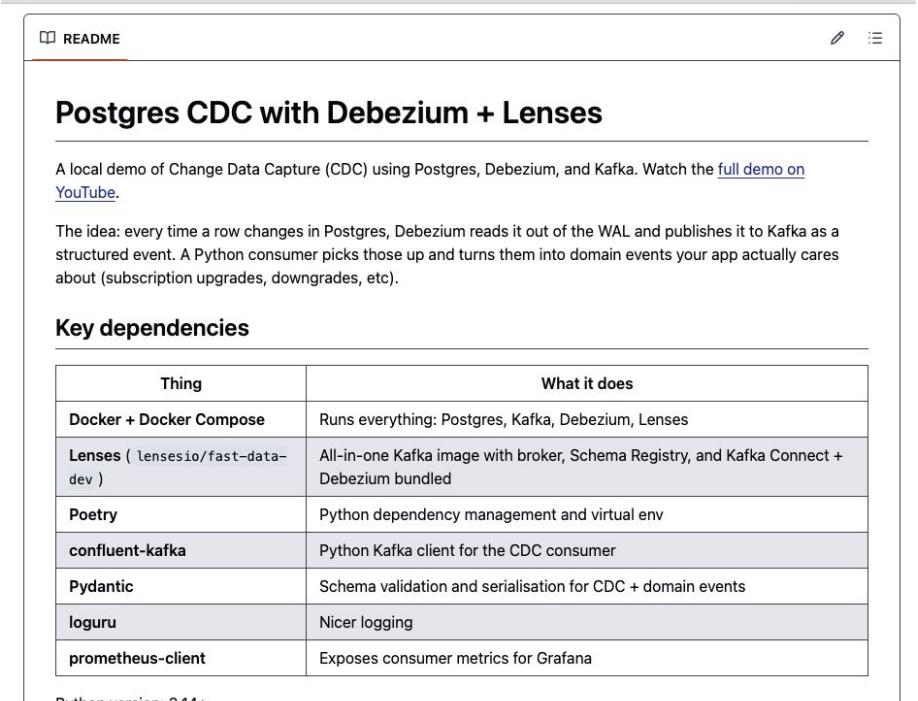
Interests: My pomeranians, weightlifting, learning new things purely for fun

https://github.com/narissatsuboi

https://www.linkedin.com/in/narissat/

Demo Code

https://github.com/narissatsuboi/postgres-cdc-kafka-demo



README

Postgres CDC with Debezium + Lenses

A local demo of Change Data Capture (CDC) using Postgres, Debezium, and Kafka. Watch the full demo on YouTube.

The idea: every time a row changes in Postgres, Debezium reads it out of the WAL and publishes it to Kafka as a structured event. A Python consumer picks those up and turns them into domain events your app actually cares about (subscription upgrades, downgrades, etc).

Key dependencies

Thing	What it does
Docker + Docker Compose	Runs everything: Postgres, Kafka, Debezium, Lenses
Lenses (`lensesio/fast-data-dev`)	All-in-one Kafka image with broker, Schema Registry, and Kafka Connect + Debezium bundled
Poetry	Python dependency management and virtual env
confluent-kafka	Python Kafka client for the CDC consumer
Pydantic	Schema validation and serialisation for CDC + domain events
loguru	Nicer logging
prometheus-client	Exposes consumer metrics for Grafana

Python version: 3.11+

Early Days @ Anthropix and Clyde

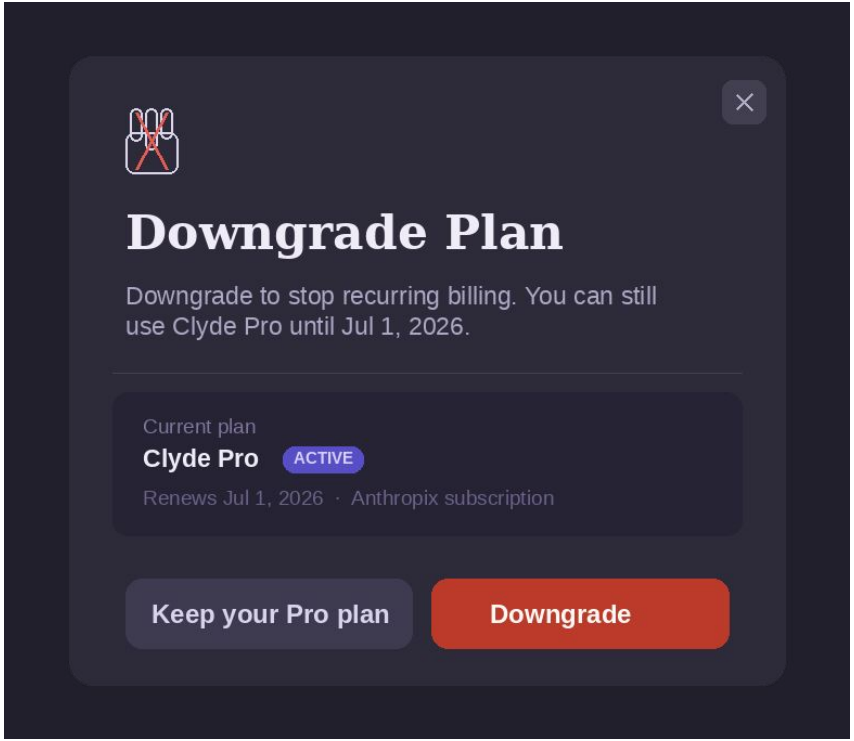
Clyde

by Anthropix



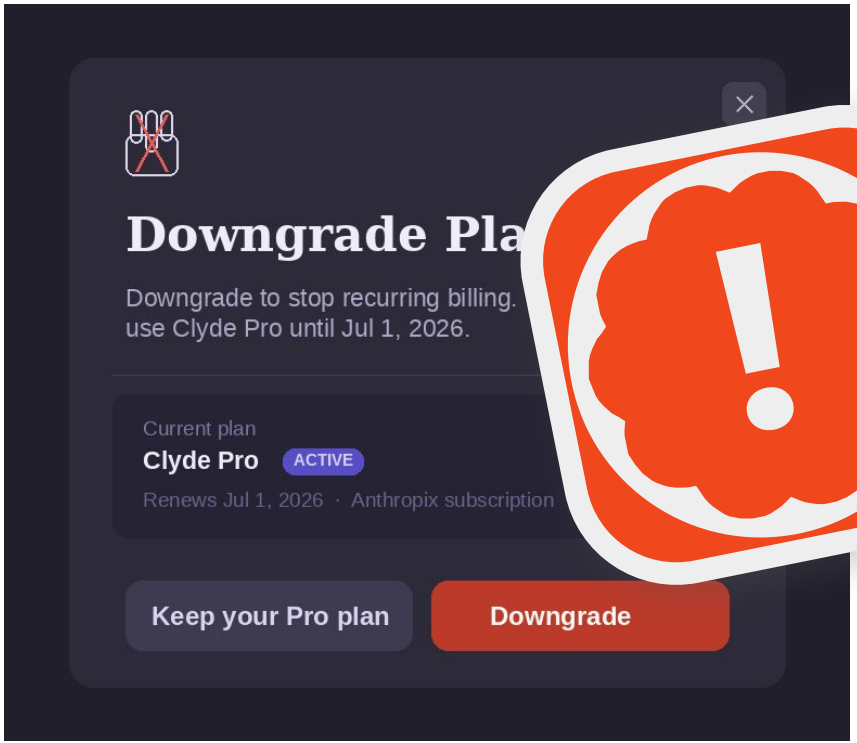
Plans: Free, Pro

Early Days @ Anthropix and Clyde



log.warning(f"{user_id} downgraded, gosh darn it")

2 Years Later @ Anthropix and Clyde

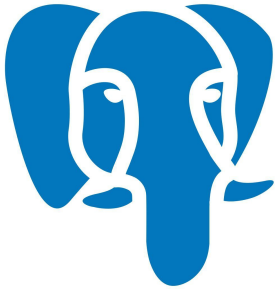




Plans: Free, Pro, Pro Max, Max Pro

Goal

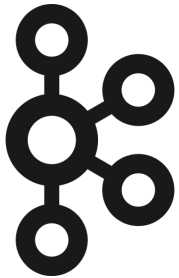
For every upgrade and downgrade that occurs, expose the event without querying the subscriptions table again.



PostgreSQL 16



io.debezium.connector.postgresql.PostgresConnector 2.7.4.Final



Kafka 4.1.x

Demo run on [Lenses.io](https://docs.lenses.io/latest) (https://docs.lenses.io/latest)

Kafka Cluster Basics

Apache Kafka: distributed event streaming platform for building real-time data pipelines and streaming applications.

Topic: an ordered log of events (records) stored durably in Kafka. Topics are Kafka's fundamental unit of organization - similar to a folder in a filesystem or a database table.

Producers write events to topics, **consumers** read events from topics.

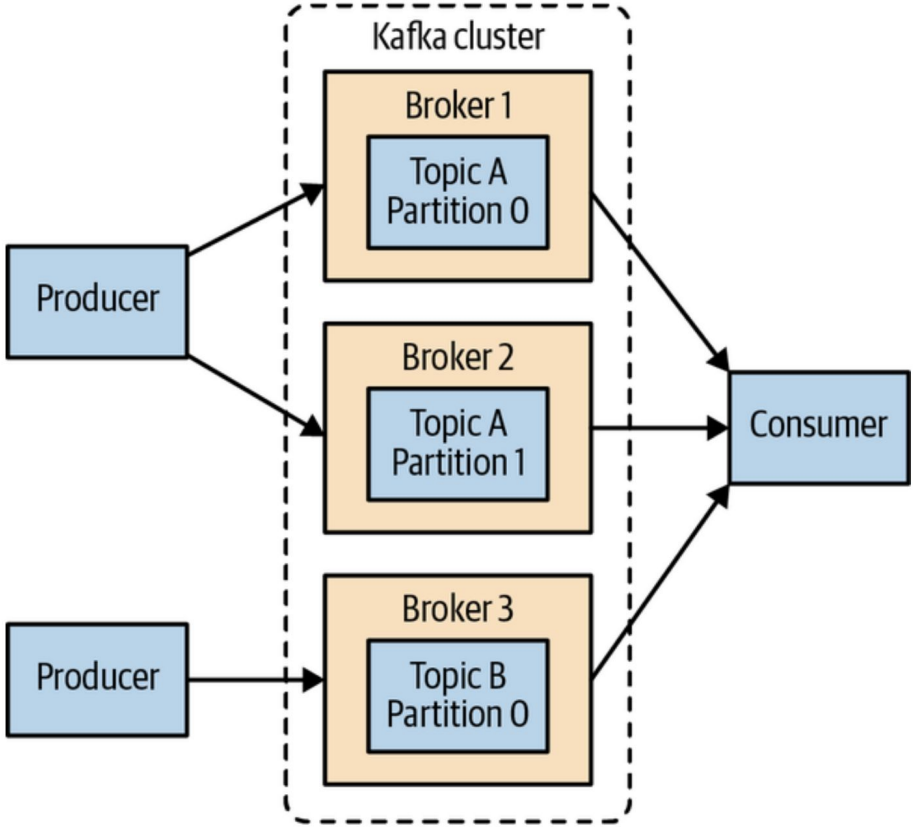


Figure 2-2. A simple Kafka cluster

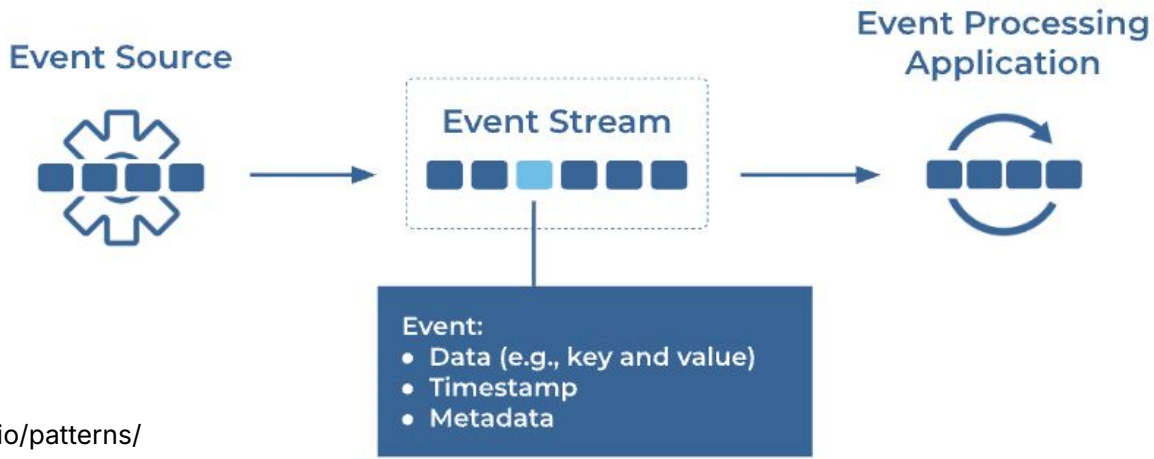
Event

Events represent facts and are used by decoupled applications, services, and systems to exchange data across an Event Streaming Platform.

Problem

How do I represent a fact about something that has happened?

Solution



https://developer.confluent.io/patterns/

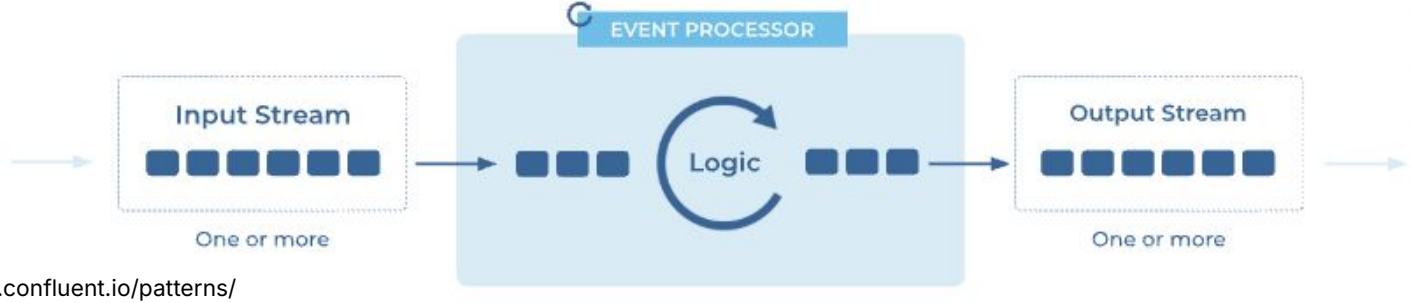
Event Processor

Once our data -- such as financial transactions, tracking information for shipments, IoT sensor measurements, etc. -- is set in motion as streams of events on an Event Streaming Platform, we want to put it to use and create value from it. How do we do this?

Problem

How can we process Events in an Event Streaming Platform?

Solution



https://developer.confluent.io/patterns/

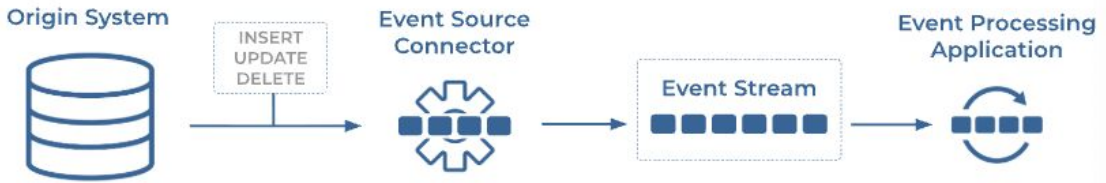
Event Source Connector

Event Processing Applications may want to consume data from existing data systems, which are not themselves Event Sources.

Problem

How can we connect cloud services and traditional systems, like relational databases, to an Event Streaming Platform, converting their data at rest to data in motion with Events?

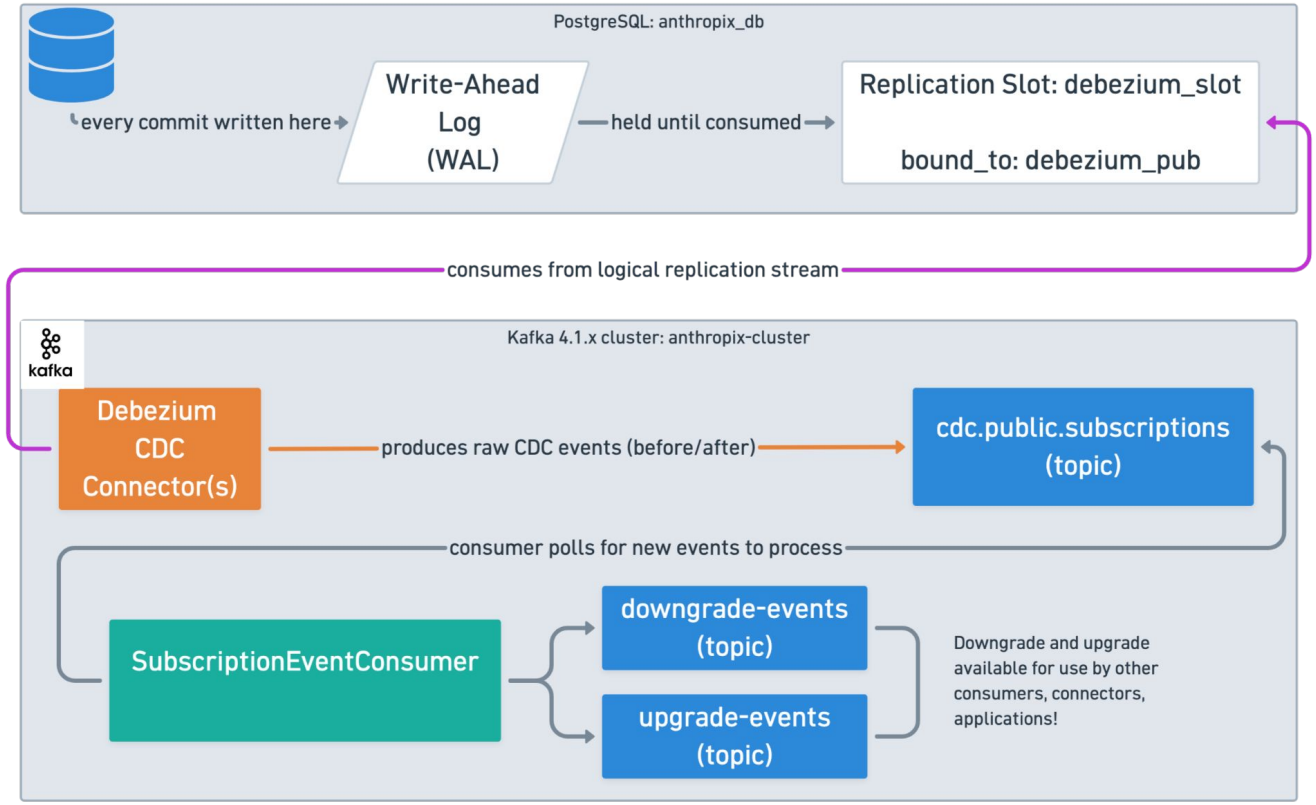
Solution



Generally speaking, we need to find a way to extract data as Events from the origin system. For relational databases, for example, a common technique is to use Change Data Capture, where changes to database tables—such as INSERTs, UPDATES, DELETEs—are captured as Events, which can then be ingested into another system. The components that perform this extraction and ingestion of Events are typically called "connectors". The connectors turn the origin system into an Event Source, then generate Events from that data, and finally sends these Events to the Event Streaming Platform.

https://developer.confluent.io/patterns/

Component Diagram: Postgres CDC Streaming to Kafka



Made with Whimsical

```
                                        Table "public.subscriptions"
       Column        |           Type           | Collation | Nullable |                  Default
---------------------+--------------------------+-----------+----------+-------------------------------------------
 id                  | integer                  |           | not null | nextval('subscriptions_id_seq'::regclass)
 user_id             | integer                  |           | not null |
 plan                | subscription_plan        |           | not null | 'free'::subscription_plan
 status              | subscription_status      |           | not null | 'trialing'::subscription_status
 token_limit         | integer                  |           | not null |
 tokens_used         | integer                  |           | not null | 0
 price_usd_cents     | integer                  |           | not null |
 trial_ends_at       | timestamp with time zone |           |          |
 current_period_start | timestamp with time zone |           | not null | now()
 current_period_end  | timestamp with time zone |           | not null | now() + '30 days'::interval
 created_at          | timestamp with time zone |           | not null | now()
 updated_at          | timestamp with time zone |           | not null | now()
Indexes:
    "subscriptions_pkey" PRIMARY KEY, btree (id)
```

```
cdc_source=# SELECT enum_range(NULL::subscription_plan);
enum_range
{free,pro,pro_max,max_pro}
```

Component Diagram: Postgres CDC Streaming to Kafka



A row is updated somewhere in our database.

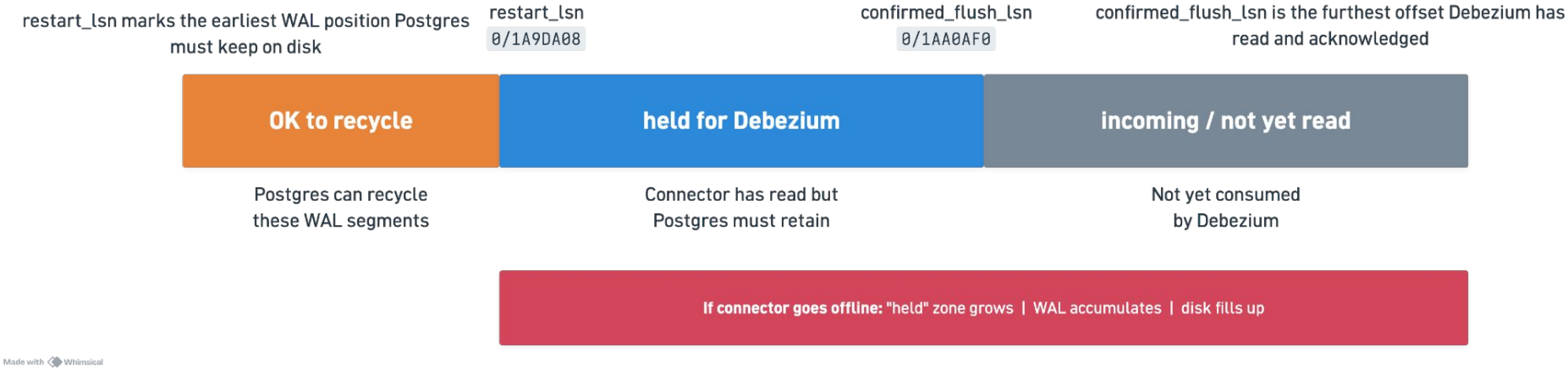
That change is appended to the WAL

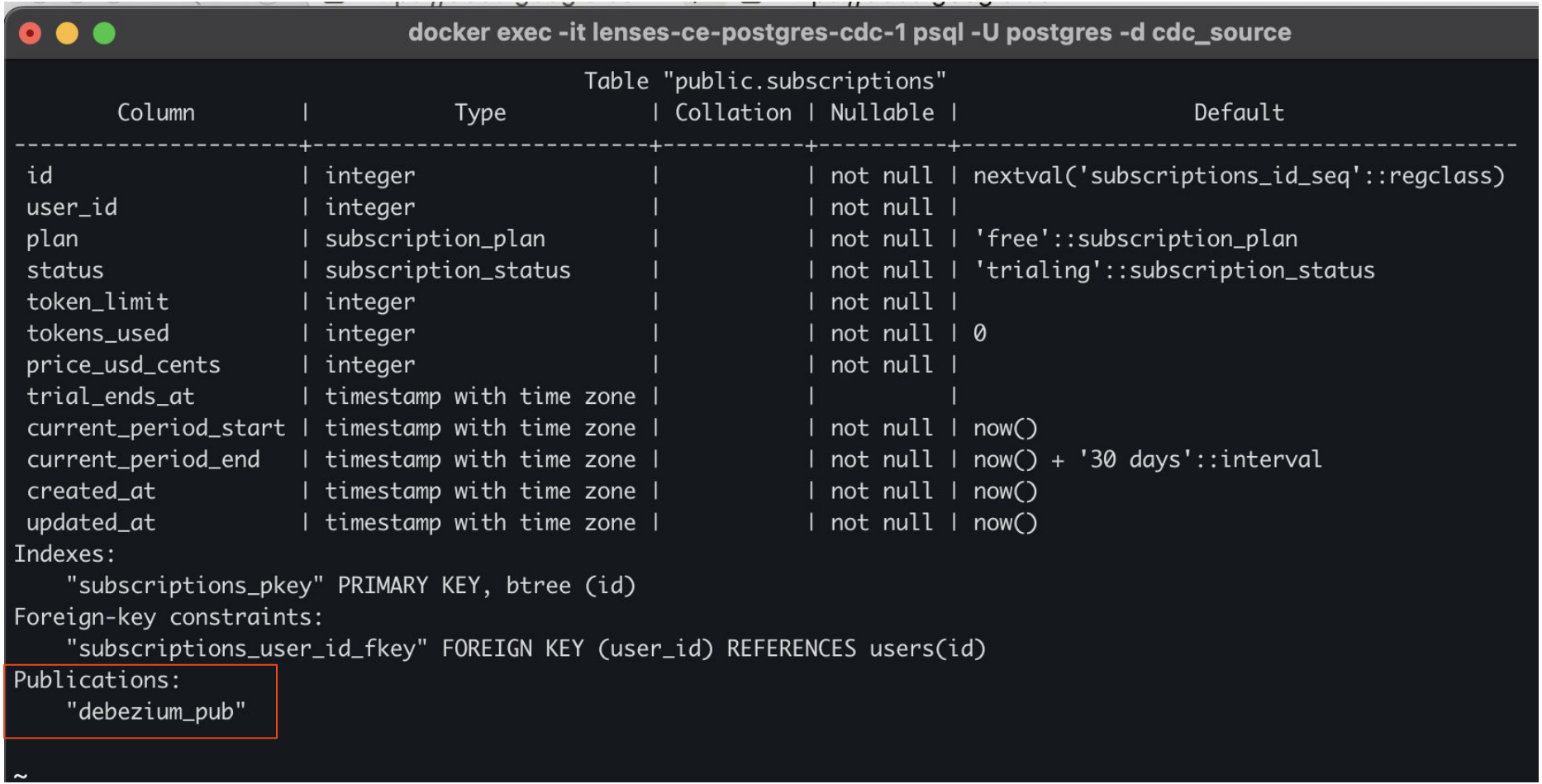
WAL: a sequential, append-only file that Postgres writes every change to *before* applying it to the actual data files.

We've configured a logical replication slot that tracks that last change in the WAL relating to the subscriptions table.

Made with Whimsical

Replication Slot Positions




docker exec -it lenses-ce-postgres-cdc-1 psql -U postgres -d cdc_source
Table "public.subscriptions"
Column | Type | Collation | Nullable | Default
id | integer | | not null | nextval('subscriptions_id_seq'::regclass)
user_id | integer | | not null |
plan | subscription_plan | | not null | 'free'::subscription_plan
status | subscription_status | | not null | 'trialing'::subscription_status
token_limit | integer | | not null |
tokens_used | integer | | not null | 0
price_usd_cents | integer | | not null |
trial_ends_at | timestamp with time zone | | |
current_period_start | timestamp with time zone | | not null | now()
current_period_end | timestamp with time zone | | not null | now() + '30 days'::interval
created_at | timestamp with time zone | | not null | now()
updated_at | timestamp with time zone | | not null | now()
Indexes:
"subscriptions_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
"subscriptions_user_id_fkey" FOREIGN KEY (user_id) REFERENCES users(id)
Publications:
"debezium_pub"
~

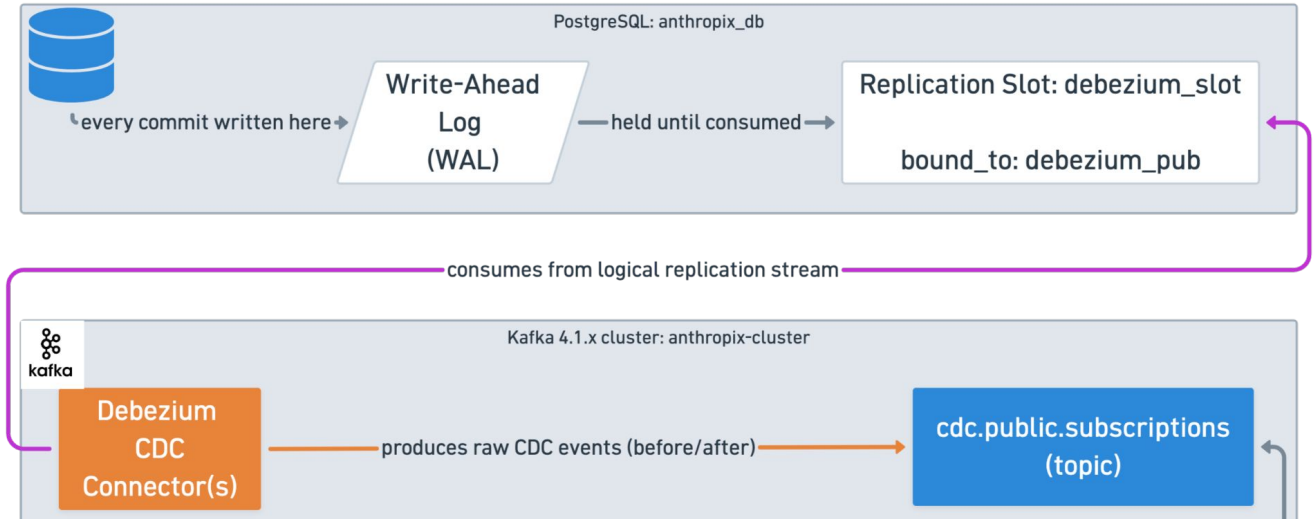
Replication Slot

```
cdc_source=# SELECT slot_name, plugin, slot_type, active, restart_lsn, confirmed_flush_lsn
FROM pg_replication_slots;
   slot_name   |  plugin  | slot_type | active | restart_lsn | confirmed_flush_lsn
---------------+----------+-----------+--------+-------------+--------------------
 debezium_slot | pgoutput | logical   | t      | 0/1A9DA08   | 0/1AA0AF0
(1 row)
```

plugin: Decodes WAL binary to structure change stream.

slot_type: use logical, not physical - physical is for streaming to replicas

Component Diagram: Postgres CDC Streaming to Kafka



Made with Whimsical

See what the connector sees (translated to JSON)

```
lsn | xid | data
0/1ACEC70 | 3749 |{
    "change": [
        {
            "kind": "insert",
            "schema": "public",
            "table": "subscriptions",
            "columnnames": [
                    "id", "user_id", "plan", "status", "token_limit", "tokens_used", "price_usd_cents",
                    "trial_ends_at", "current_period_start", "current_period_end", "created_at", "updated_at"
                ],
            "columntypes": [
                "integer", "integer", "subscription_plan", "subscription_status", "integer", "integer", "integer",
                "timestamp with time zone", "timestamp with time zone", "timestamp with time zone", "timestamp with time zone", "timestamp with time z
            ],
            "columnvalues": [
                619, 2, "pro", "trialing", 500000, 0, 0,
                "2026-06-17 22:26:18.758545+00", "2026-06-03 22:26:18.758545+00", "2026-07-03 22:26:18.758545+00", "2026-06-03 22:26:18.758545+00", "2
            ]
        }
    ]
}
```

CDC Event


```
{
  "op": "u",
  "ts_ms": 1748995200000,
  "source": {
    "version": "2.5.0.Final",
    "connector": "postgresql",
    "name": "cdc",
    "db": "cdc_source",
    "schema": "public",
    "table": "subscriptions",
    "txId": 742,
    "lsn": 24234567,
    "ts_ms": 1748995200000
  },
  "before": {
    "id": 1,
    "user_id": 1,
    "plan": "max_pro",
    "status": "active",
    "token_limit": 50000000,
    "tokens_used": 12400000,
    "price_usd_cents": 99900,
    "trial_ends_at": null,
    "current_period_start": 1746230400000,
    "current_period_end": 1748908800000,
    "created_at": 1746230400000,
    "updated_at": 1746230400000
  },
  "after": {
    "id": 1,
    "user_id": 1,
    "plan": "pro",
    "status": "active",
    "token_limit": 10000000,
    "tokens_used": 12400000,
    "price_usd_cents": 4900,
    "trial_ends_at": null,
    "current_period_start": 1748995200000,
    "current_period_end": 1751673600000,
    "created_at": 1746230400000,
    "updated_at": 1748995200000
  }
}
```

Provision debezium connector

```
{
  "name": "postgres-cdc",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "tasks.max": "1",

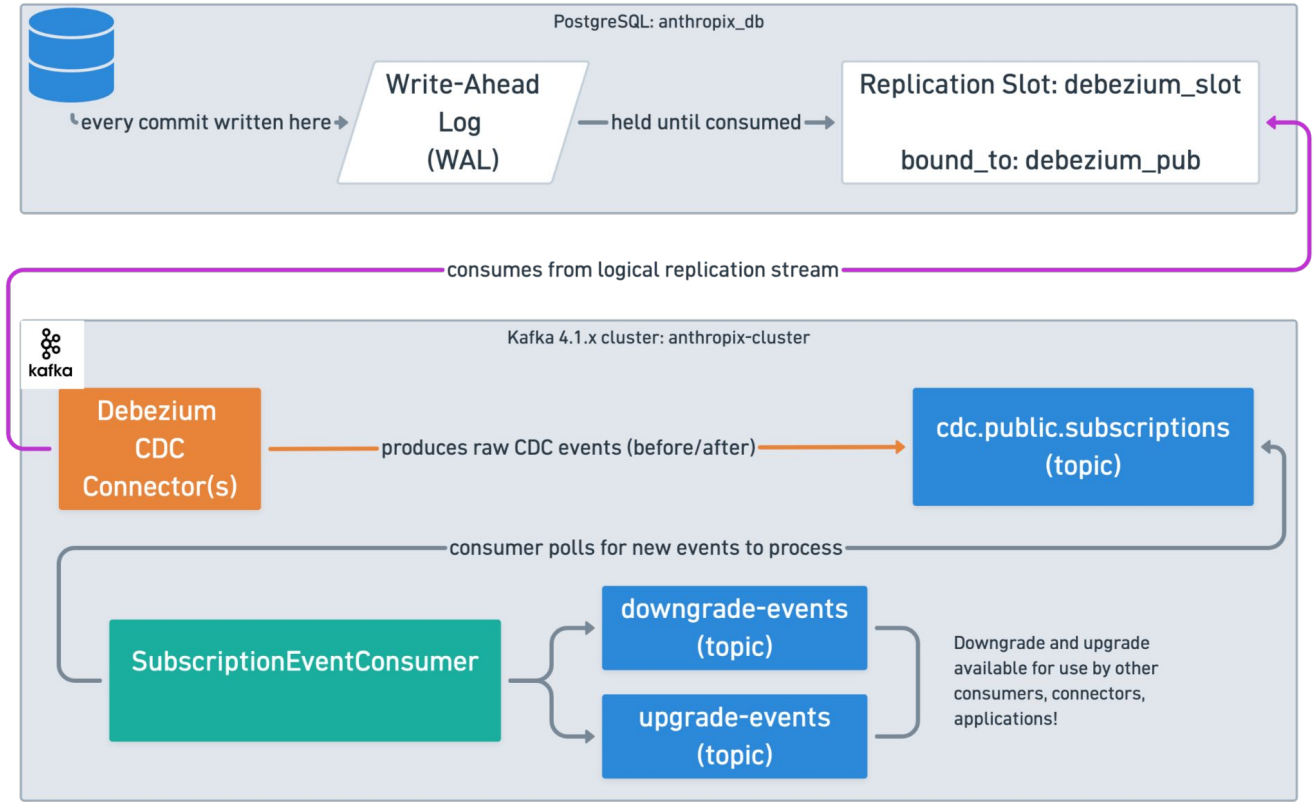
    "database.hostname": "postgres-cdc",
    "database.port": "5432",
    "database.user": "debezium",
    "database.password": "debezium",
    "database.dbname": "cdc_source",
    "database.server.name": "cdc",

    "plugin.name": "pgoutput",
    "slot.name": "debezium_slot",
    "publication.name": "debezium_pub",
    "publication.autocreate.mode": "disabled",
    "snapshot.mode": "never",

    "table.include.list": "public.users,public.subscriptions",
    "topic.prefix": "cdc",

    "key.converter": "org.apache.kafka.connect.json.JsonConverter",
    "key.converter.schemas.enable": "false",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false"
  }
}
```

Component Diagram: Postgres CDC Streaming to Kafka

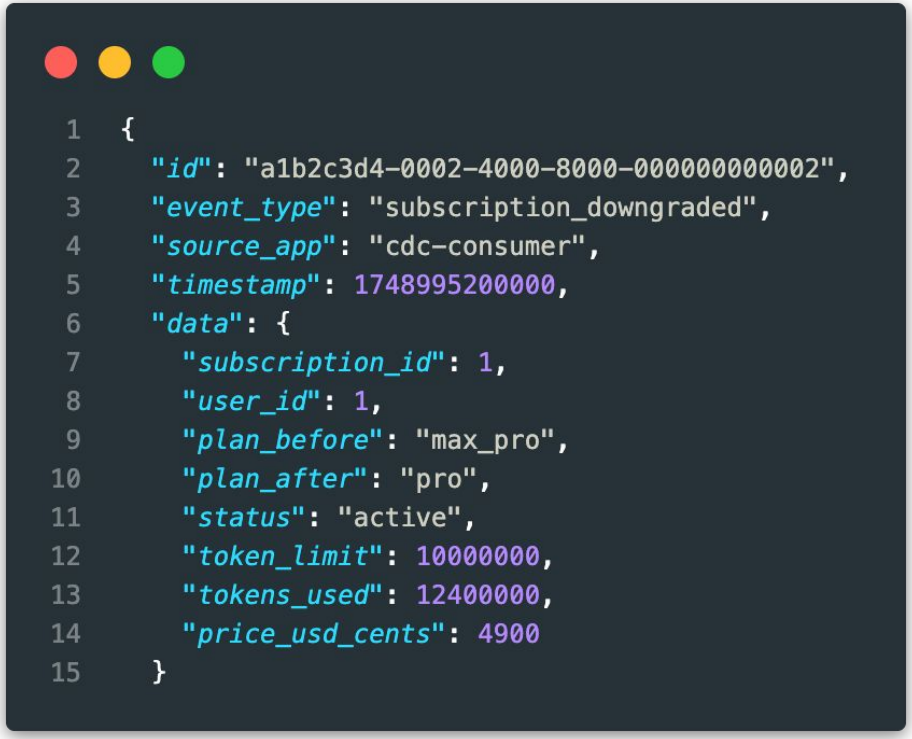


Made with Whimsical

Code time!



Map CDC Events into Business Logic Events



```json
{
  "id": "a1b2c3d4-0002-4000-8000-000000000002",
  "event_type": "subscription_downgraded",
  "source_app": "cdc-consumer",
  "timestamp": 1748995200000,
  "data": {
    "subscription_id": 1,
    "user_id": 1,
    "plan_before": "max_pro",
    "plan_after": "pro",
    "status": "active",
    "token_limit": 10000000,
    "tokens_used": 12400000,
    "price_usd_cents": 4900
  }
}
```

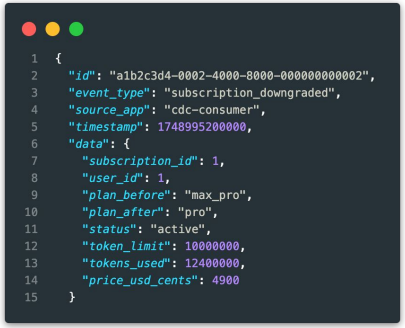
Demo time

https://www.youtube.com/watch?v=WctorG-20SI

Stream business logic events where your teams consume them

mixpanel

kafka Streams



```
{
  "id": "a1b2c3d4-0002-4000-8000-000000000002",
  "event_type": "subscription_downgraded",
  "source_app": "cdc-consumer",
  "timestamp": 1748995200000,
  "data": {
    "subscription_id": 1,
    "user_id": 1,
    "plan_before": "max_pro",
    "plan_after": "pro",
    "status": "active",
    "token_limit": 10000000,
    "tokens_used": 12400000,
    "price_usd_cents": 4900
  }
}
```

databricks

amazon S3



Gunnar Morling
Random Musings on All Things Software Engineering

Mastering Postgres Replication Slots: Preventing WAL Bloat and Other Production Issues

Posted at Jul 8, 2025 postgres cdc debezium replication

```
{
  ...
  "heartbeat.interval.ms" : "60000",
  "heartbeat.action.query" : "SELECT pg_logical_emit_message(false, 'heart
  ...
}
```

The connector executes this query every 60 seconds, writing a logical decoding message with the current timestamp to the WAL. Subsequently, it will retrieve the message via logical replication and thus allow the slot to advance. Note that the `EXECUTE` permission for this function must have been

```
max_slot_wal_keep_size=50GB
```

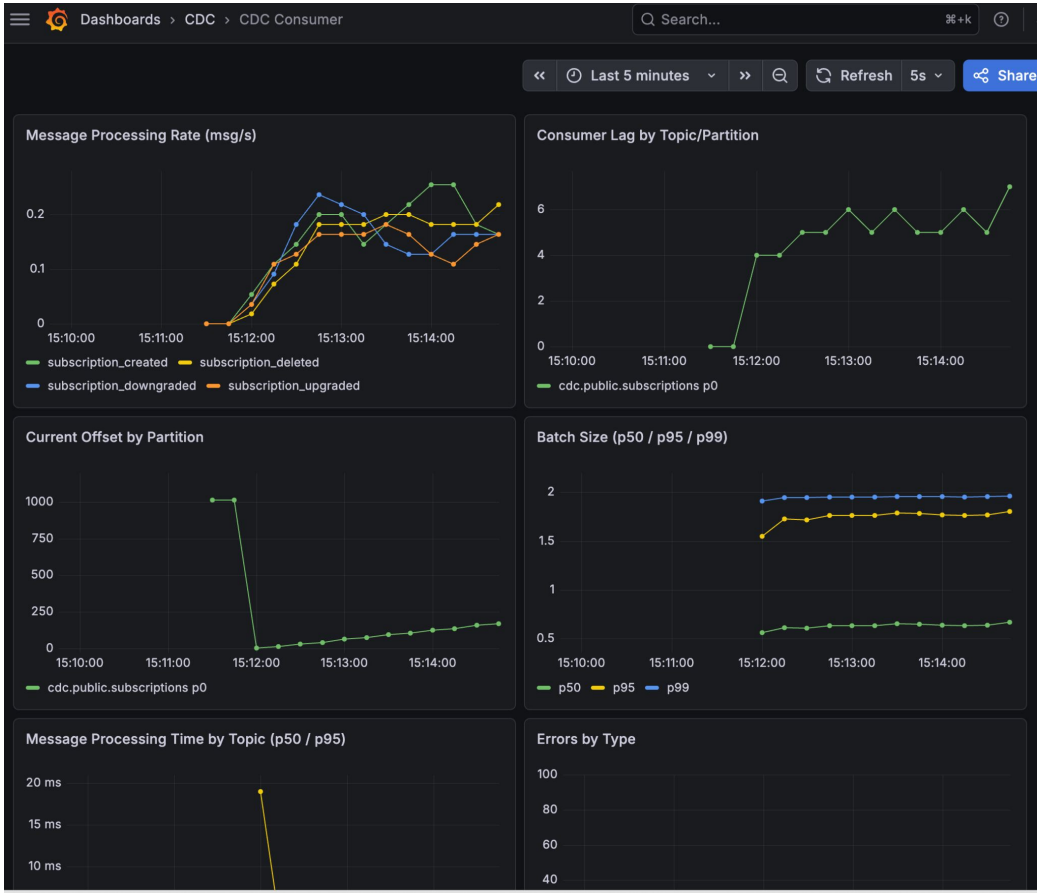
If the difference between a slot's restart LSN and the current LSN is larger than this limit, the database will invalidate the replication slot and drop older WAL segments. This renders the slot unusable, which means a new slot needs to be created, typically also taking a fresh initial snapshot of the data when using Debezium. While that's inconvenient for event consumers, it's definitely preferable to your operational database running out of disk space eventually.

https://www.morling.dev/blog/mastering-postgres-replication-slots/

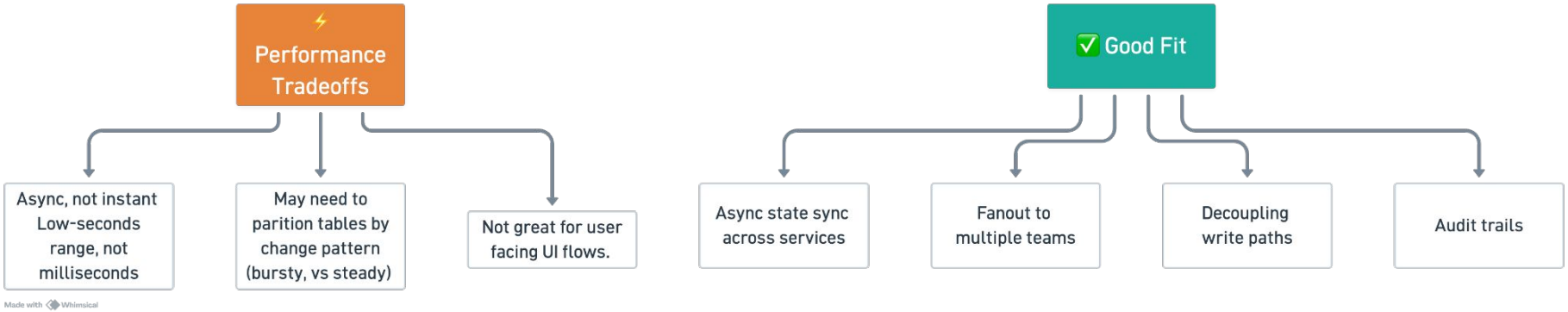


https://www.morling.dev/blog/mastering-postgres-replication-slots/

Local demo grafana



When should I try CDC?



Questions?