



`workspace_id % N`

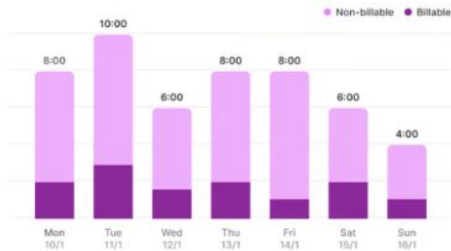
**War stories on building resilience through
application-level sharding**

José Neves - October 2026

The time tracker millions trust

Toggl gives teams the time data they need for better capacity, profitability, and project planning.

Weekly project report



Team activity

Jackie 25h 29m 45s Martha 23h 10m 14s Annie 22h 40m 12s

Show latest 20 time entries ▾

TEAM MEMBER	DESCRIPTION	DURATION
Mike	Documentation	3h 12m 50s
Nina	Hiring interviews	5h 25m 30s





📅 This week · W37



Member



Client



Project



Task



Tag



Description



+ Add filter



Reports might be outdated due to a temporary issue. We're working to fix it as soon as possible.

Total Hours

-

Billable Hours

-

Amount

-

Av

0.

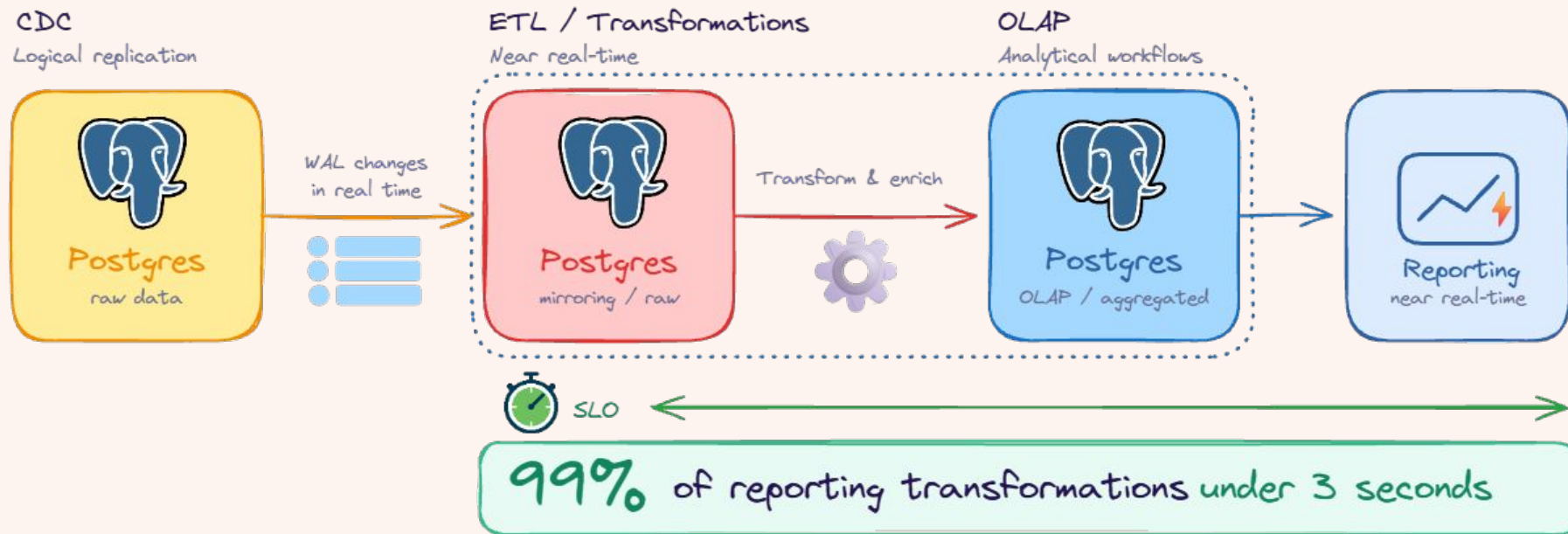
Duration by day

Stack by: Billable



Nothing to see here...

Near real-time requirement



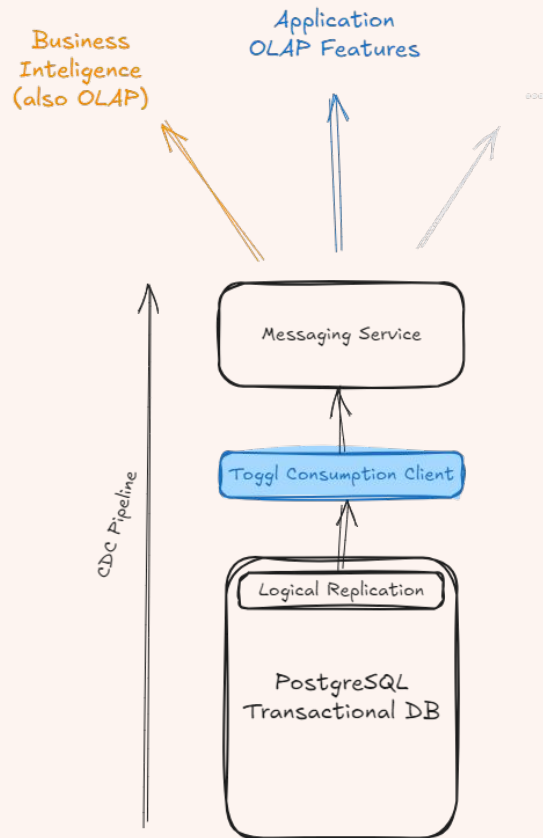
Our ETL Pipeline

More **control over batching**, and how the data is pre-massaged, over existing go-to solutions.

Built in **dedup** and **recovery processes**.

Better fit on our stack and design choices.

Extensibility, and integration capabilities.



Flaws Features

The **sequential nature** of the ETL pipeline meant that **any issue was everyone's issue**.

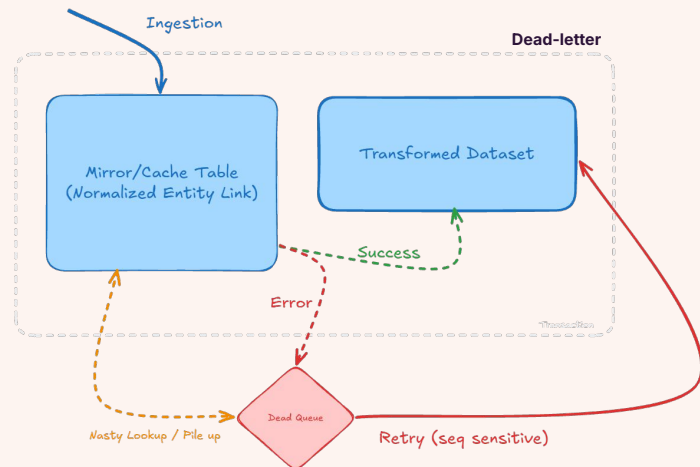
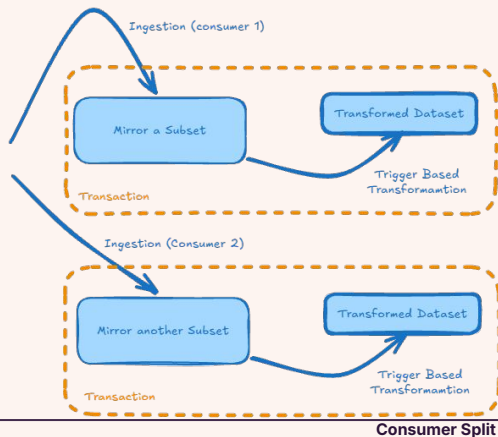
Having a multi-tenant database also meant that batches of logical replication events **were compiled indiscriminately**.

Success made the problem larger: a single event gets everyone's data propagation stale.

Flaws Features

Over-time Brainstormed Solutions

- Dead-letter queues
- Multiple transformation optimizations
- Splitting transformations by creating multiple consumers.



All trade latency, and compute for reliability.
And most are not... elegant.

Keeper Concept

Selective mirror of transactional data needed because logical replication events only carry data for the target row, in a given table.

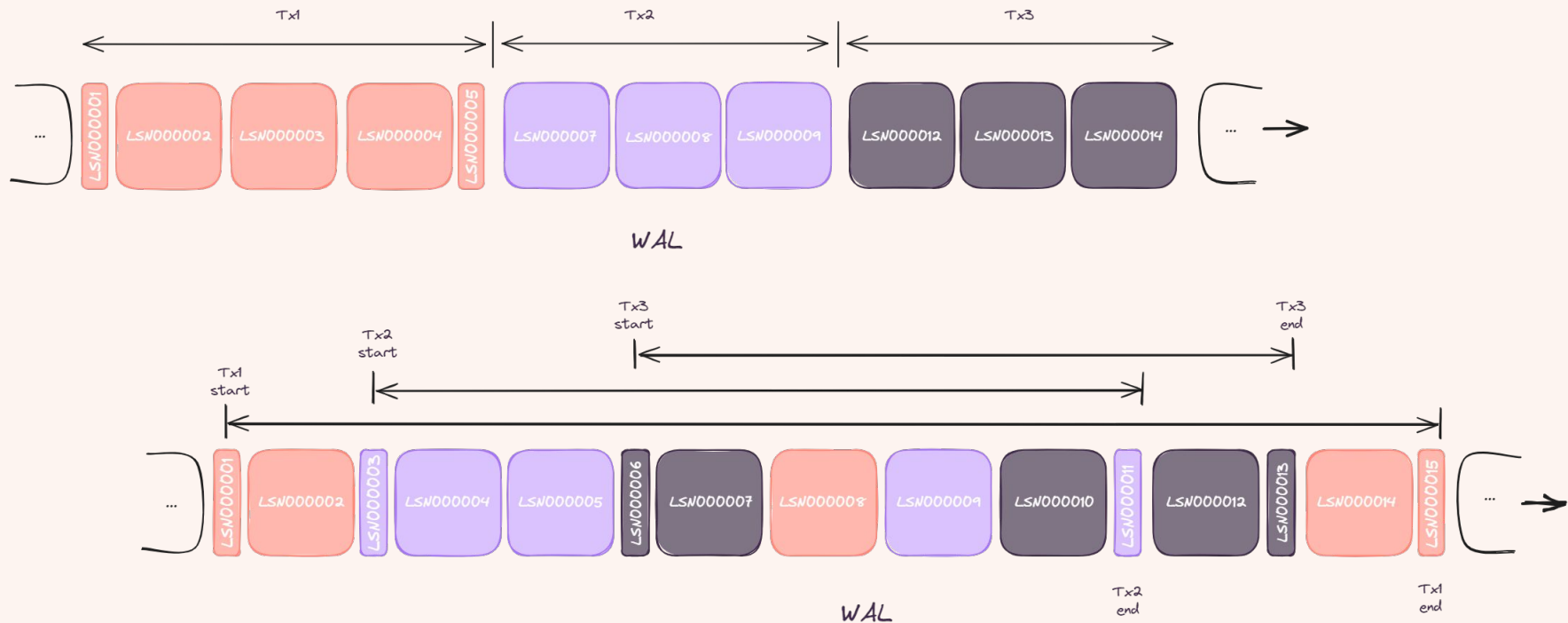
```
[  
  {"lsn": "0/0001", "entity": "user", "data": {...}},  
  {"lsn": "0/0002", "entity": "tasks", "data": {...}},  
]
```

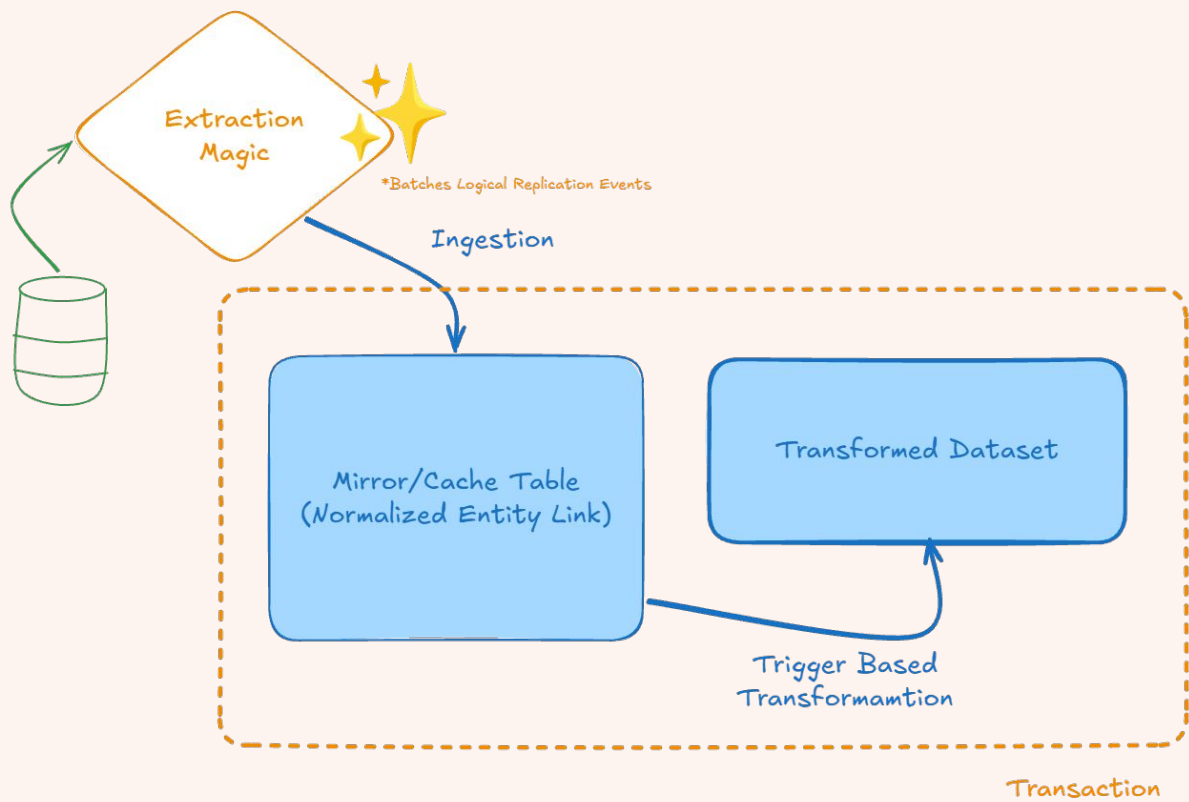
Our "keeper" mirror:

- Supports transformations **that require data that is not in the change event**
- Support contextual layer of filtering



Ordered Chaos



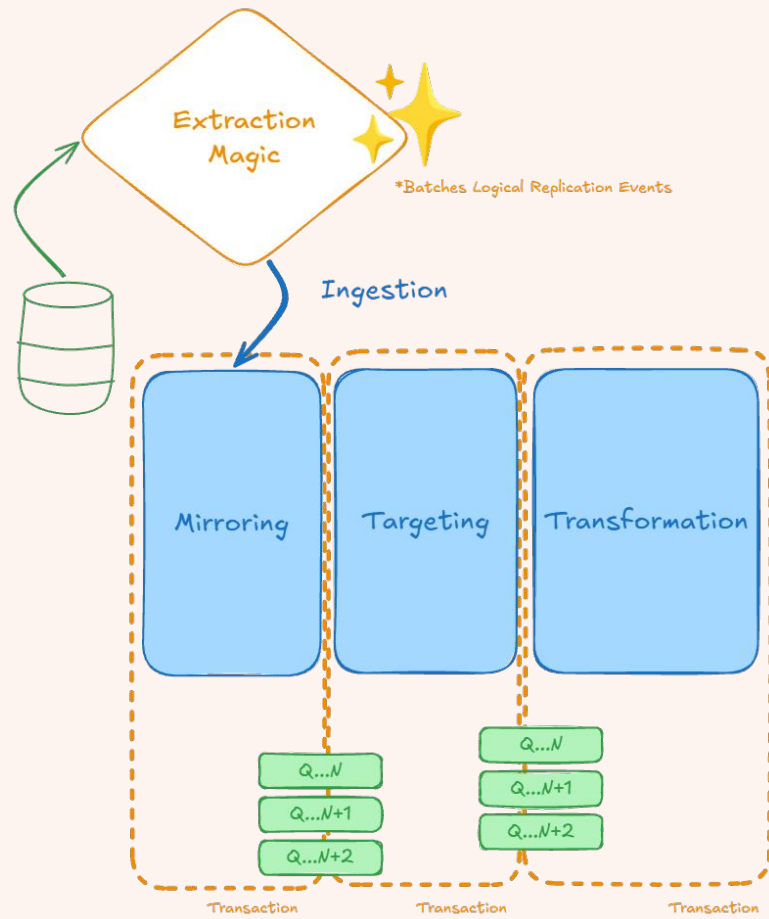


New Approach

Split the large transformation transaction, decoupling **mirroring**, **targeting**, and the **transformation** steps.

The split also meant that we needed to keep some sort of **volatile state between the different steps** - queues:

- **Targeting queue:** what changes are waiting for the targets to be computed.
- **Transformation queue:** what targets are waiting to be refreshed.



New Approach

Overwriting work in queues is fine: re-enqueue gets discarded by insert conflict constraints. **We care about updating - eventually - after last write, not once per write.**

Plus, queue management allows **low time operation retries.**

17% of targeted work never runs
A newer change squashed it before it ever executed.
74.7M targeted → 62.1M transformed

Claim is not FIFO. Workers pick the oldest workspace (`partition_key`), take an advisory lock then claim rows for that workspace only.

Planner hitting: avoid hash and merge joins.

New Approach

Postgres as a durable work queue

```
-- claim pattern (simplified)
WITH picked AS (
  SELECT id FROM horizonte_keeper.target_queue
  WHERE processing_at IS NULL AND attempt_count < 10
  ORDER BY created_at ASC, last_attempted_at ASC NULLS
  FIRST
  LIMIT $1
  FOR UPDATE SKIP LOCKED
)
UPDATE horizonte_keeper.target_queue q
SET processing_at = now(), last_attempted_at = now()
FROM picked WHERE q.id = picked.id
RETURNING ...;
```

- Hard-delete on success (queue is not an audit log).
- $\text{attempt_count} + \text{max } 10 \rightarrow$ **quarantine** poison workspaces instead of head-of-lines block.
- Stale recovery: $\text{processing_at} < \text{now}() - \text{interval reset under SKIP LOCKED}$.

New Approach

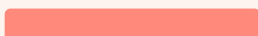
Deadlock prevention on concurrent (en)queue/delete

Naive concurrent workers upserting were deadlocking queues.

```
WITH partition_locks AS MATERIALIZED (  
  SELECT  
    pg_advisory_xact_lock(hashtext(hz_transformation_enqueue'), pk::int)  
  FROM UNNEST($8::BIGINT[]) AS pk -- MUST be sorted  
  ascending  
)  
INSERT INTO horizonte_keeper.transformation_queue  
(...)  
SELECT ... FROM UNNEST(...)  
ON CONFLICT (time_entry_id, partition_key) DOUPDATE  
SET ...
```

- Sort distinct `partition_keys` ascending before taking locks.
- Modulo worker assignment by partition to reduce contention.
- **Bulk, bulk, bulk, not row by row**
One statement over UNNEST arrays
- Partial covering index on the claim path: `INCLUDE (partition_key)` → index-only scans

New Approach

mirror · keeper upsert		28.4 ms
targeting · claim		30.9 ms
targeting · query		185 ms
targeting · enqueue		15.9 ms
transform · claim		42.6 ms
transform · query		363 ms

The new approach was working, but barely:

Latency and increased processing needs of the multiple hoops of the new process were **not a match for the efficiency** of the previous bulk transaction.

It was a known trade-off for a more robust process.

Sharding

~~Sealing~~ "future proofing it" was now a more pressing concern.

There are several ways to add sharding to Postgres: extensions like **Citus**, forks like **Postgres-XL**, **pg_shard**, or middleware in front of the database.

But these introduce a single point of failure when used out of the box.

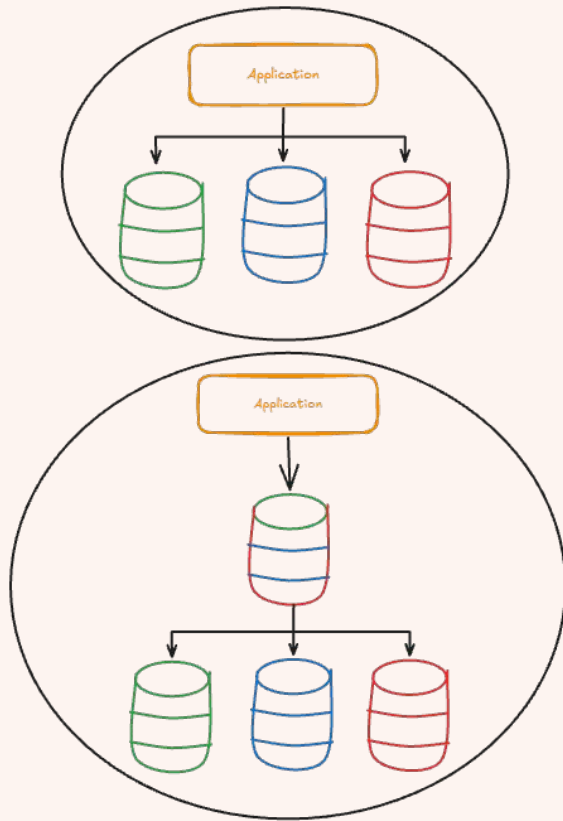


Sharding

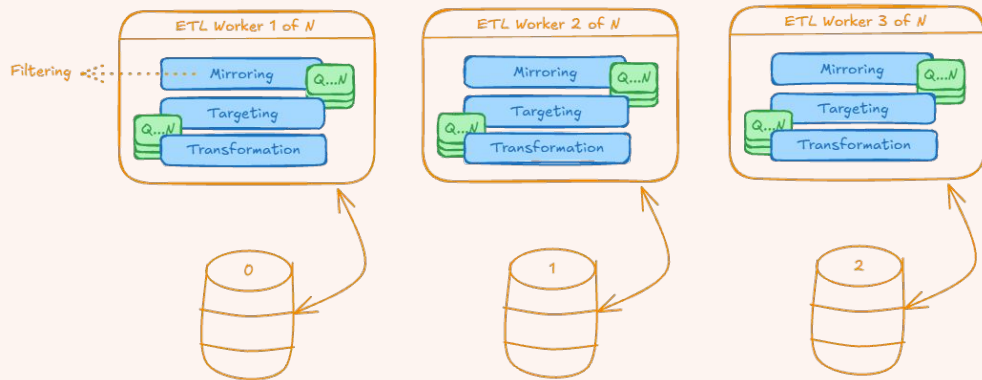
Sharding means horizontally partitioning data across multiple servers. And in our case that also meant distributing processing needs - which were our bottleneck, rather than IO pressure.

Sharding can be implemented in two ways:

- **Application-level** in which the application itself is aware of data partitioning.
- **PostgreSQL Extensions** that build in distributed data support.

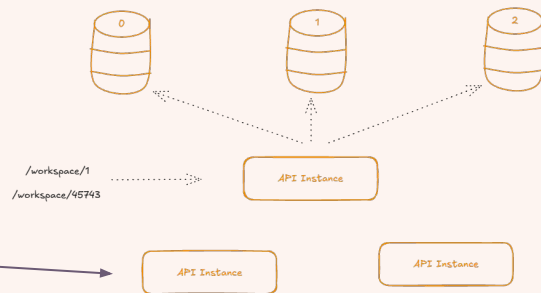


workspace_id % N



- Application N-aware, and targets the connection string index matching the remainder.

- **Identical schema**, same migrator on different targets.
- Shared mirrored tables for common data.



Maintenance and Visibility

Transform maintenance: one UPSERT query per dataset instead of a forest of entity-specific trigger transforms.

Full OTel per event was impractical - **a volume suicide.**

Instead we leveraged WAL LSN + stage timestamps through both queues, log structured fields, and use them to track claim, target, enqueue, and transform steps.

Which allows us to **fully trace every single event processing performance.**

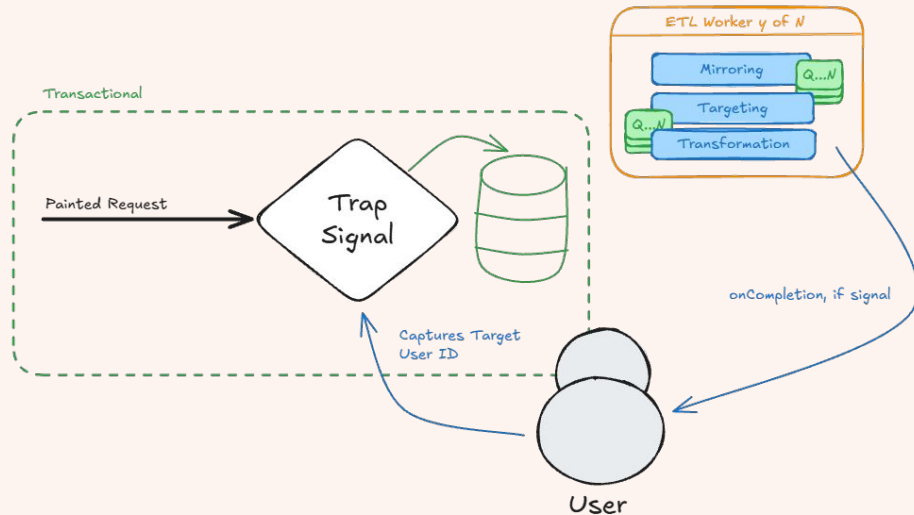


Sync UX

Eventual consistency is challenging to get right.

Flows need to **look sync, consistent**, and **delay or fail gracefully** in front of the user.

1. We don't pretend the pipeline is synchronous.
2. Opt-in signal on the write, not all flows need it.
3. **No fake strong consistency** - avoid `wait sleep(10)` at all costs.



Test Success

1. **Segregation:** poison entity and load tests.
2. **Observability:** queue lag gauges, ack rate/latency alerts.
3. **Operability:** generation/recovery reuses the same queues.
4. **Unified engine:** "OLAP as a package" is now used in multiple of our products.



Take home

Transformation living on transactional ack path is more efficient **but a less robust solution**: Durable enqueue first, heavy work after.

Postgres is an viable option for work queueing derived relational work - simplifies transactional implementation. Use `FOR UPDATE SKIP LOCKED` , unique upserts, advisory locks and lock order.

One full-row transform beats N partial updates - our overengineering threshold.

Application level sharding helped us future proof it - **grow N**.



José Neves  @rafalneves

 /rafalneves

Software Architect
strangeskills.com