PG PostgreSQL · W wal2json · Dz Debezium · K Apache Kafka · D Drasi CNCF Sandbox

Building Event-Driven Architectures Directly on PostgreSQL

CDC, Continuous Queries, and Real-Time Filtering with wal2json, Debezium + Kafka, and Drasi

Diaa Radwan
Global Blackbelt, Microsoft

1 Live Benchmarks
Latency, throughput, and DB overhead measured across 3 CDC approaches

2 Working Code
Docker Compose to Azure PG Flex
Python consumers, Locust traffic gen

3 Practical Guidance
WAL safety, replication slots, and when to use which approach

Azure PostgreSQL Flexible Server · Docker Compose · Python · Prometheus + Grafana

The Problem — Detecting Database Changes

Your app needs to react when data changes. How do you detect it?

THREE COMMON APPROACHES (AND THEIR PROBLEMS)

1 Polling — Slow

```
SELECT * FROM orders
WHERE updated_at > :last
```

Delayed detection
seconds to minutes of lag

Wasted DB queries
when nothing changed

Misses changes
between poll intervals

Cannot catch DELETEs

Most common approach —
90% of apps start here

2 Database Triggers — Risky

```
CREATE TRIGGER notify
AFTER INSERT ON orders
```

Blocks writes
runs inside the transaction

Hard to debug
logic hidden in DB layer

Side effects on every write
performance degrades at scale

Fire-and-forget

Tight coupling between
schema and business logic

3 Dual-Write — Dangerous

```
db.save(order)
kafka.send("paid", order)
```

DB succeeds, Kafka fails
data loss

Kafka succeeds, DB fails
phantom events

No atomicity guarantee
two systems, no coordination

Every service must implement

#1 cause of data inconsistency
in microservice architectures

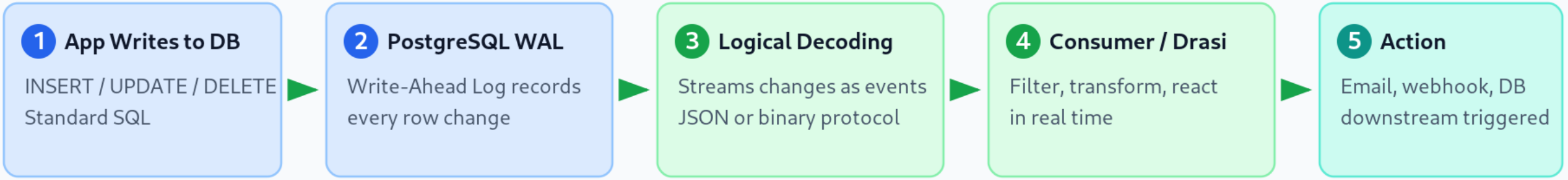
There has to be a better way. What if PostgreSQL could tell you when data changes?

Next: Change Data Capture — let PostgreSQL do the work

Change Data Capture — The Solution

PostgreSQL already records every row change in the WAL. We just need to read it.

THE CDC PIPELINE



KEY BENEFITS

Zero app code changes

Any write to the database automatically becomes an event

Captures everything

INSERTs, UPDATEs, DELETEs — nothing missed

Guaranteed delivery

The WAL is durable — if the row committed, the event exists

No dual-write risk

One source of truth. One write path. Consistent.

That's what we're building today. Three approaches. Same database. Same workload. Different tradeoffs.

How Logical Replication Works

From application write to consumer — five steps

THE PIPELINE

1 App Writes

INSERT / UPDATE / DELETE on the orders table.

Standard SQL writes.
No code changes needed.

2 WAL Captures

Write-Ahead Log records every row change with before + after values.

3 Logical Decoding

Plugin converts binary WAL into JSON (wal2json) or protocol (pgoutput).

4 Publication Filter

Only tables you choose. FOR TABLE orders filters at the PG level.

5 Slot → Consumer

Replication slot tracks position (LSN). Consumer reads stream in order.

Key Concept: The Replication Slot

Each consumer gets its own slot. The slot tracks where the consumer is in the WAL stream.

PostgreSQL retains WAL segments until the consumer confirms it has processed them.

This is powerful — but also dangerous. We'll come back to that.

Azure

Azure PostgreSQL Flexible Server

wal_level = logical is a dynamic parameter — no restart needed.
Up to 64 replication slots. Default max_replication_slots = 10.

Three Things to Configure

wal_level, a publication, and a replication slot

1 wal_level = logical

```
-- postgresql.conf
wal_level =                logical
max_replication_slots = 10
max_wal_senders = 10

-- Verify:
SHOW wal_level;
-- Must return 'logical'
```

Requires restart on self-managed PG

Azure PG Flex: dynamic — no restart

2 Publication

```
-- Choose tables to publish
CREATE PUBLICATION
    app_cdc_pub
FOR TABLE orders;

-- Or publish ALL tables:
CREATE PUBLICATION
all_changes
FOR ALL TABLES;
```

Think of it as a filter
at the PostgreSQL level

3 Replication Slot

```
-- wal2json example
SELECT
pg_create_logical_
replication_slot(
'    wal2json_slot    ',
'    wal2json  ');

-- pgoutput (Drasi/Debezium)
... 'drasi_slot','pgoutput'
```

Each consumer gets its own slot.
Slot retains WAL until consumed.

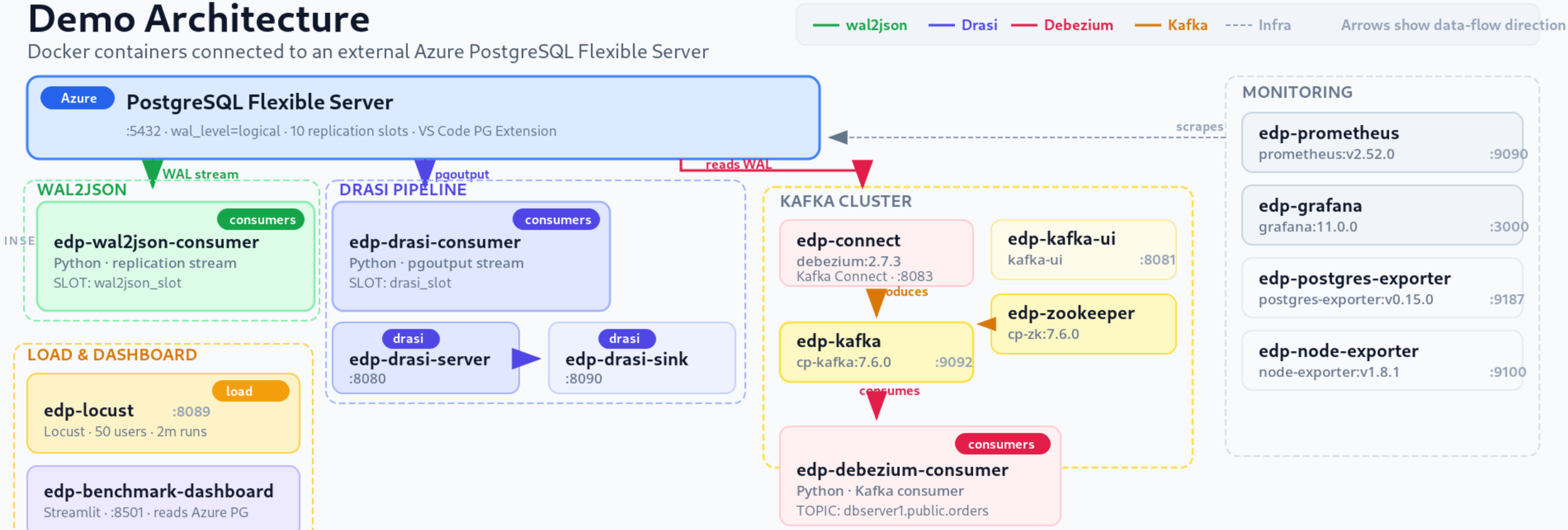
Summary

wal_level = logical tells PG to include row data. A Publication selects tables. A Slot tracks consumer progress.

Next: the demo architecture — three CDC pipelines running in parallel

Demo Architecture

Docker containers connected to an external Azure PostgreSQL Flexible Server



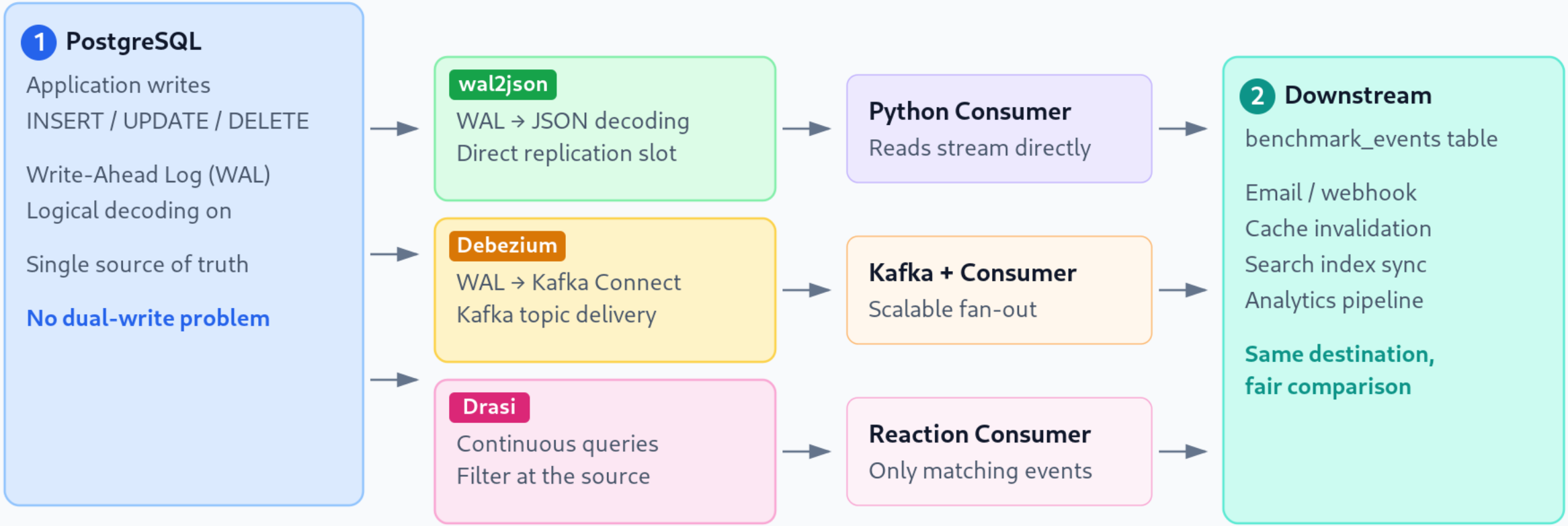
Summary

14 Docker containers + 1 external Azure PostgreSQL Flexible Server

- **1 WAL2JSON**
  Direct WAL consumer via wal2json plugin — lowest latency, single slot per consumer
  edp-wal2json-consumer
- **4 CDC Consumers**
  Drasi pipeline (consumer + server + sink) with continuous queries + Debezium consumer
  edp-drasi-consumer · edp-drasi-server · edp-drasi-sink · edp-debezium-consumer
- **2 App**
  edp-locust (traffic generator) · edp-benchmark-dashboard (Streamlit results)
- **4 Kafka**
  Streaming backbone — broker, schema, connect, and UI for Debezium CDC
  zookeeper · kafka · kafka-ui · connect
- **4 Monitoring**
  Metrics collection, dashboards, and PostgreSQL / host exporters
  prometheus · grafana · postgres-exporter · node-exporter

Three CDC Approaches — PostgreSQL

Same database, same workload, different tradeoffs. All writing to benchmark_events.



Same load. Same data. Different architectures. Let's run them and compare.

wal2json — CDC Pipeline Flow

PG WAL → wal2json plugin → Python consumer → DB write

1 PostgreSQL WAL

Write-Ahead Log records every row-level change

Logical decoding enabled wal_level = logical

Already happening for every write

2 wal2json Plugin

Decodes binary WAL into JSON format

Custom output plugin installed on server

Replication slot consumer

3 Python Consumer

Reads replication stream directly from PostgreSQL

Parses JSON change events ~150 lines of Python

No middleware needed

4 DB Write

INSERT into benchmark_events

Timestamps + latency metrics recorded

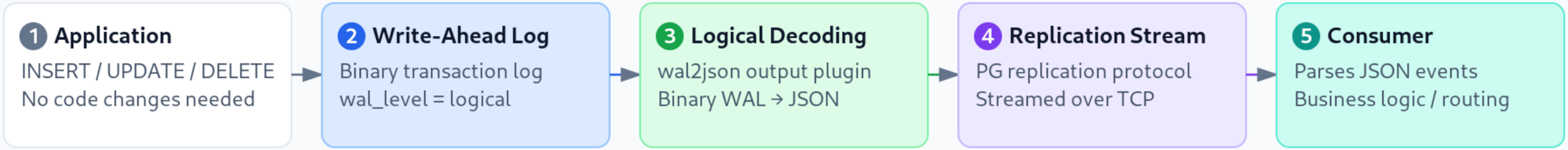
15,200 rows captured

- **~1.5 ms** average latency
- **15,200** events captured
- **Direct connection** — no middleware overhead

wal2json — How It Works

PostgreSQL native CDC via logical decoding — zero external infrastructure

The Mechanism



Sample wal2json output:


```
{"change":[{"kind":"insert","table":"orders","columnnames":["id","customer","amount"],
"columnvalues":["1001","acme-corp","249.99"]}]}
```


Benefits

- **Built into PostgreSQL**
- **Zero application changes**
- **Guaranteed delivery** — WAL is durable
- **JSON output** — easy to parse
- **Sub-millisecond latency**
- **No dual-write problem**

Simplest CDC path — ~150 lines of Python

Trade-offs

- **One slot = one consumer** — no fan-out
- **WAL bloat risk** — slow consumer blocks cleanup
- **No retries or DLQ** — reliability is on you
- **Tight DB coupling** — direct PG connection
- **No schema registry**
- **Manual monitoring** — track lag yourself

5 consumers = 5 slots = 5× WAL cost

wal2json — Real-World Use Cases

Where wal2json shines: single-consumer, low-infrastructure CDC patterns

WHEN TO USE WAL2JSON

Event Sourcing

Capture domain events from database writes without dual-write in microservice architectures.

One slot per aggregate stream

Cache Invalidation

Detect row changes and flush Redis, Memcached, or CDN caches automatically.

Sub-ms detection, instant flush

Search Index Sync

Stream changes to Elasticsearch or OpenSearch. Keep search indexes up-to-date in real time.

No stale search results

Audit Logging

Capture every mutation for compliance without modifying application code or triggers.

Zero-touch compliance

Data Replication

Replicate specific tables to another database, data warehouse, or cross-region standby.

Selective table mirroring

Real-Time ETL

Stream row changes to a data lake or warehouse as they happen — no batch windows.

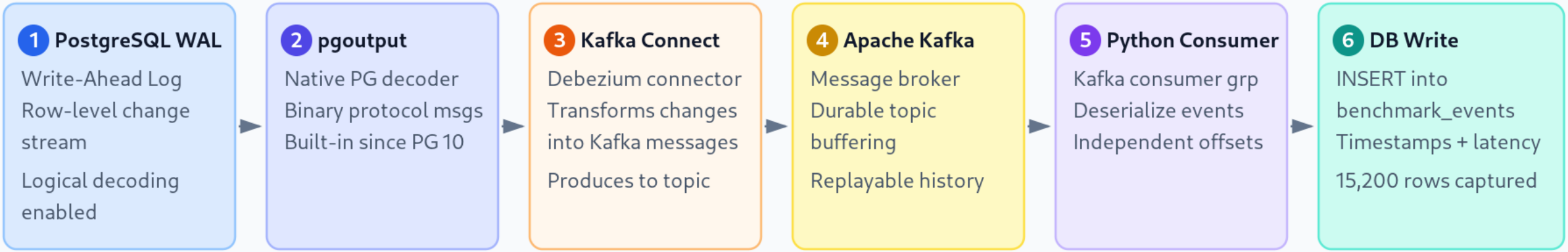
Continuous data pipelines

Best fit: Single consumer, all events, simplest possible setup. Add Kafka or Drasi later if needed.

Next: Let's add Kafka to the mix — Debezium pipeline

Debezium — CDC Pipeline Flow

PG WAL → pgoutput → Kafka Connect → Kafka → Python consumer → DB write



- **~560 ms** average latency
- **15,200** events captured
- **Kafka hop** — broker write + consumer poll latency

Debezium + Kafka — How It Works

Enterprise-grade CDC via Kafka Connect — scalable fan-out with durable event streaming

Sample Kafka Message

Debezium Kafka message (simplified):

{"before":null,"after":{"id":1001,"customer":"acme-corp","amount":249.99},
"source":{"table":"orders","lsn":234881024},"op":"c","ts_ms":1713800000000}

Benefits and Trade-offs

Benefits

- **Fan-out to many consumers**
  Multiple consumer groups, independent offsets
- **Durable event log**
  Kafka retains events — replay anytime
- **Schema evolution**
  Schema Registry tracks before/after schemas
- **Exactly-once semantics**
  With Kafka transactions + idempotent writes
- **Rich ecosystem**
  100+ connectors for sources and sinks

Trade-offs

- **Infrastructure complexity**
  Kafka + ZooKeeper/KRaft + Connect cluster
- **Higher latency — ~560ms**
  Extra hop through Kafka adds significant delay
- **Operational overhead**
  Broker tuning, topic management, monitoring
- **Resource-heavy**
  JVM-based, significant memory and CPU
- **WAL bloat still possible**
  If connector falls behind, same PG risk

Best fit: Multiple consumers need the same events. You already have Kafka. Event replay is required.

Debezium + Kafka — Real-World Use Cases

Where Debezium shines: multi-consumer, enterprise-scale event streaming

WHEN TO USE DEBEZIUM + KAFKA

Microservice Eventing

Decouple services with Kafka topics per entity. Each service consumes independently at own pace.

Fan-out via consumer groups

CQRS / Read Models

Build optimized read-side projections from CDC events — search, dashboards, views.

Separate read and write paths

Cross-Region Replication

Stream database changes across regions via Kafka MirrorMaker for geo-distributed systems.

Active-active multi-region

Real-Time Data Mesh

Publish domain data products as Kafka topics. Teams self-serve via consumer groups.

Data as a product, owned by teams

Outbox Pattern

Write events to an outbox table. Debezium captures the outbox and publishes to Kafka reliably.

Transactional guarantees + events

Best fit: Already have Kafka + need fan-out + event replay. Enterprise middleware stack.

Next: Drasi — what if you only care about specific data patterns?

Drasi — CDC Pipeline Flow

PG WAL → pgoutput → continuous query engine → filtered events → DB write

1 PostgreSQL WAL

Write-Ahead Log records every row-level change

Logical decoding enabled wal_level = logical

Same WAL as wal2json

→

2 pgoutput Plugin

Native PG protocol decoder — built-in

since PostgreSQL 10 Structured binary msgs

Same protocol as Debezium

→

3 Continuous Query

Drasi evaluates every change against queries

Only matching events are emitted downstream

THE KEY DIFFERENCE

→

4 DB Write

INSERT into benchmark_events

Both filtered and unfiltered rows logged

20,269 rows captured

- **~1.5 ms** average latency
- **20,269** events (filtered + unfiltered)
- **Server-side query** — continuous filtering

Drasi — How It Works with PostgreSQL

Query-driven CDC — continuous queries that react only when results change

Sample Continuous Query

Drasi continuous query — only fires when a high-value paid order appears:

```
MATCH (o:Order) WHERE o.priority = 'high' AND o.amount > 500 AND o.status = 'PAID'
RETURN o.id, o.customer, o.amount -- only matching rows trigger a Reaction
```

One Slot → Multiple Queries → Multiple Reactions

Drasi uses 1 replication slot. Runs 2+ continuous queries against the single stream.
Each query can trigger a different reaction (webhook, gRPC, HTTP). Fan-out without extra slots.

Benefits and Trade-offs

Benefits

- **Filter at the source**
  Only matching events leave the pipeline
- **Declarative queries**
  Define what matters, not how to detect it
- **1 slot, many queries**
  Fan-out without extra replication slots
- **Runtime rule changes**
  Add new queries mid-stream, no restarts

Cypher + SQL — graph and relational queries

Trade-offs

- **Newer project**
  Smaller community than Debezium/Kafka
- **No durable event log**
  No Kafka-style replay or retention by default
- **Kubernetes-native**
  Designed for K8s, not ideal for bare-metal
- **Complex query limits**
  Very complex queries may impact performance

CNCF Sandbox — active development

Best fit: Specific data patterns. High-priority alerts. Rules that change at runtime. K8s-native environments.

Drasi — Real-World Use Cases

Where Drasi shines: targeted reactions to specific data patterns

WHEN TO USE DRASI

High-Priority Order Alerts

React only when orders match priority + amount thresholds. Skip all routine transactions.

~70% fewer downstream events

Threshold-Based Alerts

Trigger when inventory drops below reorder point, account hits limit, or SLA is breached.

Continuous monitoring, no polling

Fraud Detection

Continuous queries on transaction patterns — flag anomalies in real time without batch jobs.

Millisecond anomaly detection

Compliance Monitoring

Watch for regulatory violations as data changes — GDPR retention, PCI scope, audit gaps.

Real-time compliance checks

IoT Event Processing

React to sensor data patterns stored in PostgreSQL — equipment failure prediction, alerting.

Graph queries across relations

Key advantage: Rules change at runtime. No code changes. No restarts. Add queries mid-stream.

Next: Benchmark results — all three approaches compared side by side

Benchmark Results

2-minute Locust run • 50 concurrent users • Azure PG Flexible Server

~1.5 ms

wal2json avg latency

~560 ms

Debezium + Kafka avg latency

~1.5 ms

Drasi avg latency

Feature Comparison

Metric	wal2json	Debezium + Kafka	Drasi
Avg Latency	~1.5 ms	~560 ms	~1.5 ms
Events Captured	15,200	15,200	20,269 (incl. filtered)
Replication Slots	1 per consumer	1 (Kafka fans out)	1 (queries fan out)
Fan-Out	Extra slot per consumer	Kafka consumer groups	Multiple queries on 1 slot
Filtering	Client-side	SMTs or client-side	Server-side continuous query
Operational Complexity	Low — Python script	High — Kafka cluster	Medium — K8s operator
Durable Event Log	No	Kafka retention	No

Key Takeaway: wal2json is the simplest. Kafka adds durability + fan-out at the cost of latency.
Drasi adds intelligence + fan-out at the source. No single winner — it depends on your needs.

Test Config: Azure PG Flexible Server • 2 vCores • 50 users • 2-min Locust run • Monitoring via Prometheus/Grafana
All three read the same WAL. The difference is where processing happens and how events are distributed.

WAL Danger Zone

When replication slots stall, WAL files accumulate and disk fills up

64 GB — **of WAL accumulated in production**

because one consumer was paused for 2 hours. The DB went read-only.

WAL Bloat Timeline — 3 Danger Phases

Phase 1 — Healthy

Consumer is running

Consumer reads WAL events.

Slot LSN advances normally.

Old WAL segments recycled.

WAL size: ~256 MB (normal)

Phase 2 — Consumer Paused

Slot holds WAL position

PostgreSQL cannot recycle

WAL segments past the slot.

Files accumulate on disk.

WAL size: growing... 4 GB → 16 GB

Phase 3 — Disk Full

Database goes READ-ONLY

No more writes accepted.

Application errors cascade.

Emergency intervention needed.

WAL size: 64 GB → disk full

Root cause: Replication Slot + Inactive Consumer

A replication slot tells PostgreSQL "keep all WAL from this point forward." If the consumer disconnects or falls behind, WAL accumulates indefinitely. This affects ALL three CDC approaches — not just one.

This is the #1 operational risk of logical replication.

Next slide: defenses and monitoring

WAL Defenses

Three layers of protection against replication slot WAL bloat

Defense 1 — Cap WAL Size

```
-- postgresql.conf or ALTER SYSTEM
max_slot_wal_keep_size = '10GB'
-- PG 13+: auto-invalidates stale slots
```

PostgreSQL will invalidate any slot
that holds more than 10 GB of WAL.
Consumer must re-sync after invalidation.

Best first line of defense

Defense 2 — Monitor Slot Lag

```
SELECT slot_name,
pg_wal_lsn_diff(pg_current_wal_lsn(),
confirmed_flush_lsn) AS lag_bytes
```

Query pg_replication_slots regularly.
Alert when lag exceeds threshold.
Prometheus + Grafana in our demo.

Alert before it's too late

Defense 3 — Drop Stale Slots

```
-- Emergency: drop the stalled slot
SELECT pg_drop_replication_slot(
'stale_slot_name');
```

Immediately frees WAL disk space.
Consumer loses its position.
Must re-create slot + re-sync.

Nuclear option — use when disk is critical

Operations Checklist

- **Set max_slot_wal_keep_size (PG 13+)**
- **Monitor slot lag with Prometheus**
- **Alert on lag > 1 GB (or your threshold)**
- **Automate stale slot cleanup**
- **Document runbook for WAL emergencies**
- **Test consumer restart recovery**

Azure PostgreSQL Flexible Server

- wal_level = logical via Server Parameters
- Set max_slot_wal_keep_size in portal
- Azure Alerts on storage % used
- Metrics: pg_replication_slots in Azure Monitor

No server restart needed for wal_level change

Remember: WAL safety applies to ALL three approaches. Every replication slot is a risk. Monitor them all.

Decision Guide

Which CDC approach fits your use case?

wal2json — Use When…

- **Single consumer pattern**
- One service reads the change stream
- **Lowest latency matters**
- Sub-2ms change capture needed
- **Minimal infrastructure**
- No Kafka, no K8s, just Python + PG
- **Prototypes and small teams**
- Get started in minutes

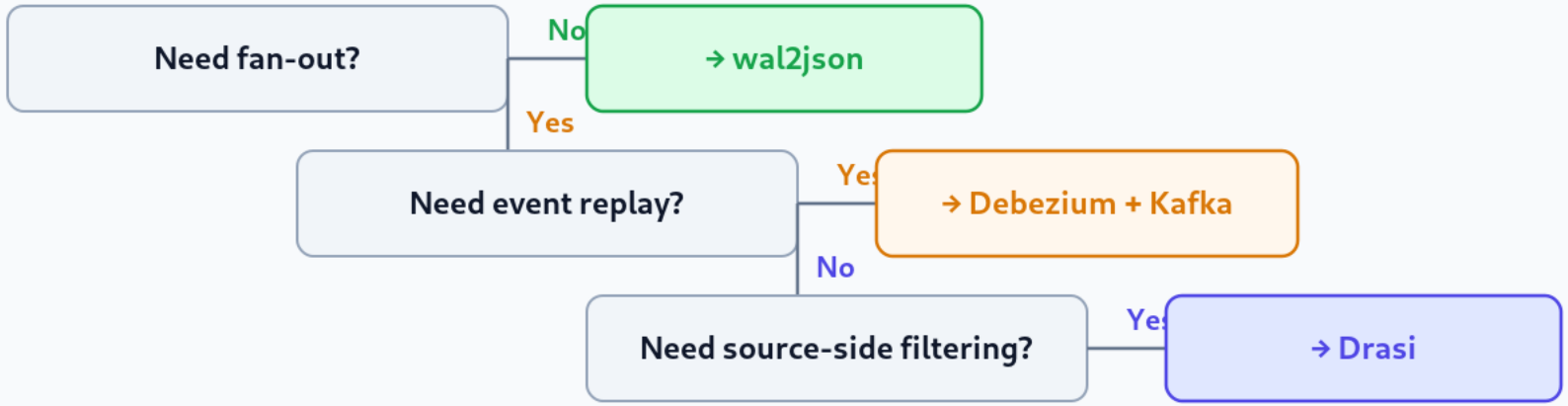
Debezium + Kafka — Use When…

- **Multiple consumers needed**
- Fan-out to many services via Kafka
- **Event replay required**
- Durable log with configurable retention
- **Schema evolution matters**
- Avro, JSON Schema, Protobuf support
- **Existing Kafka ecosystem**
- Already running Kafka? Great fit.

Drasi — Use When…

- **Specific data patterns matter**
- React only to matching changes
- **Rules change at runtime**
- Add/modify queries without restart
- **Reduced downstream noise**
- ~70% fewer events downstream
- **Kubernetes-native environment**
- CNCF Sandbox project, K8s operator

Quick Decision Flow



Thank You!

All code is open source:

github.com/diaa/event-driven-with-postgresql

Questions? Let's connect!

PostgreSQL • Debezium • Kafka • Drasi