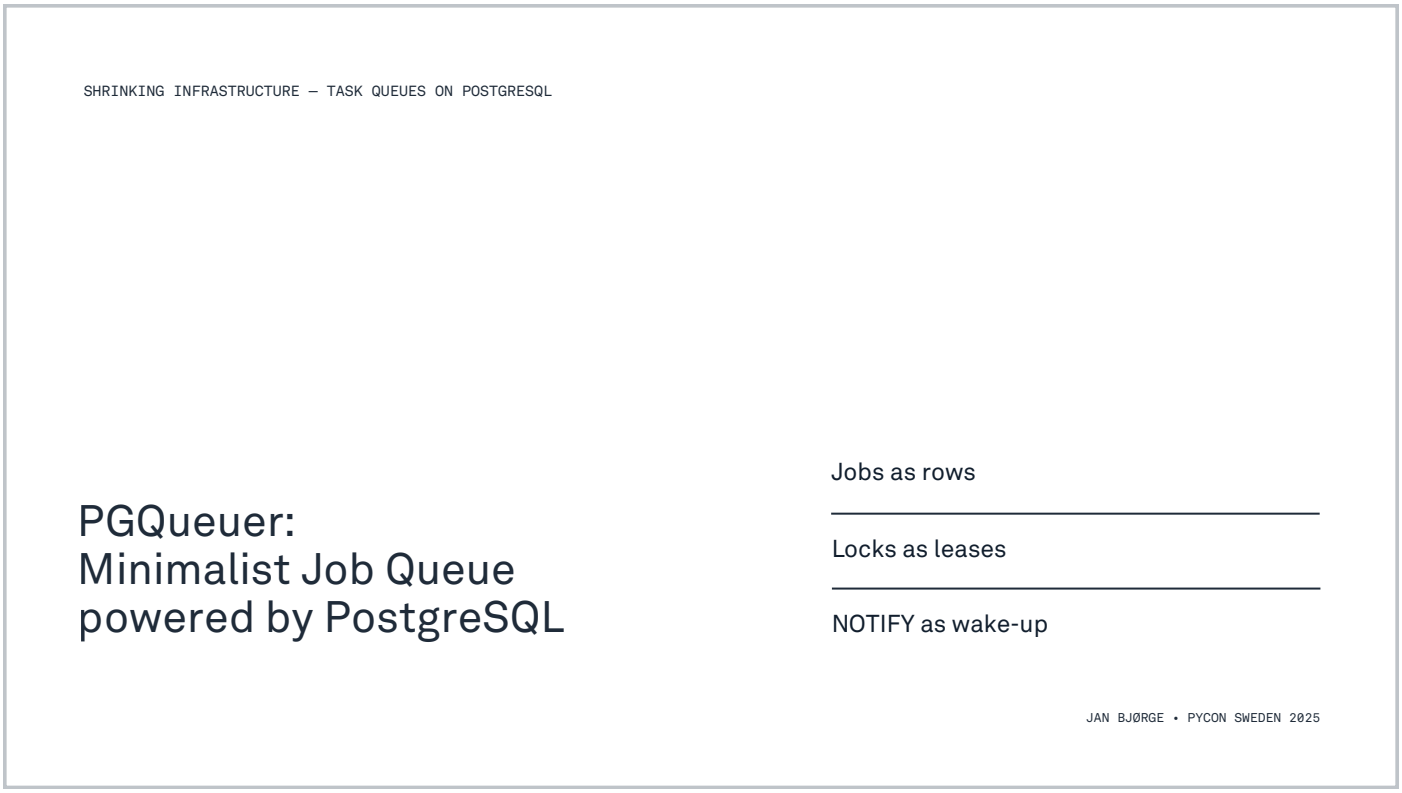


Hi everyone, I'm Jan Bjørge or just JB. I built PGQueuer — a lightweight job queue that runs entirely on Postgres.

If your app already uses Postgres, can it also run your background jobs?

PGQueuer shows that you can: it's reliable, fast, and requires no new infrastructure.

Over the next half hour, I'll show how it works, why it's robust, and when it makes sense to use it.

I work at Pio, where small robots deliver warehouse orders to a worker for picking and packing. Just like a vending machine! We like to say it's just like a vending machine

It's built on AutoStore technology, and our stack is mostly Python with some sprinkes of Rust.

If you like robotics and python we're hiring.

The pitch: what if you could run your entire job queue on the database you already have? No new infrastructure to operate.

That's the core idea behind PGQueuer. It's not revolutionary - it's pragmatic. Your Postgres instance handles your application data, your transactions, your consistency guarantees. Why shouldn't it also handle your background jobs?

Existing solutions existed, but seemed to require the installation of extensions and with un-clear path to success given a manged db (might be depending on given cloud provider).

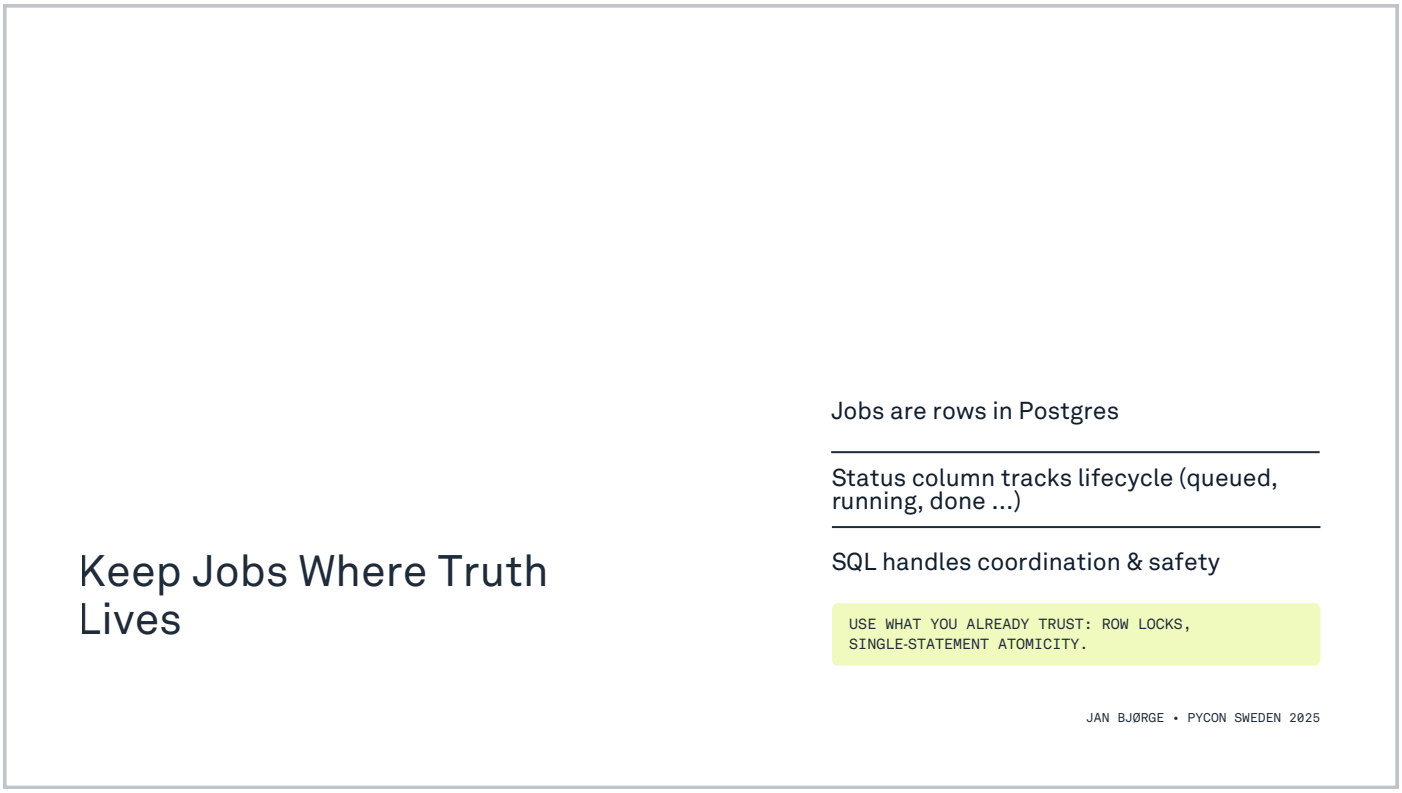
The concept is simple: jobs as rows and NOTIFY as your wake-up signal.

This started with a side project.
I had an app tracking eletrical price changes, and when it detected features i needed to trigger a background job.

Everyone said "just use Redis", but I kept asking: why? My data is already safely in Postgres and the cloud provider at the time did not offer a manged Redis, so i thout. How hard can it be?

I took a small bet: could I build a queue in a weekend? Turns out yes-and it was simpler than other options at the time.

That result became PGQueuer.



Lets keep jobs where your data already lives.
Each job is a row.
Lifecycle manged with a status column.
Workers coordinate using PgSQL with `FOR UPDATE SKIP LOCKED` for concurrency, `LISTEN/NOTIFY` for instant wake-ups, and WAL writes for durability - all via a single-statement - no long-lived transactions.

So when a worker dies mid-task, the database notices automatically. If the job's heartbeat stops updating, it detects staleness and moves the job back to queued. Another worker picks it up and retries. No external orchestration needed - Postgres has the information to recover.

Data Model

TWO TABLES POWER THE SYSTEM:

Four ordinary tables keep everything transparent:

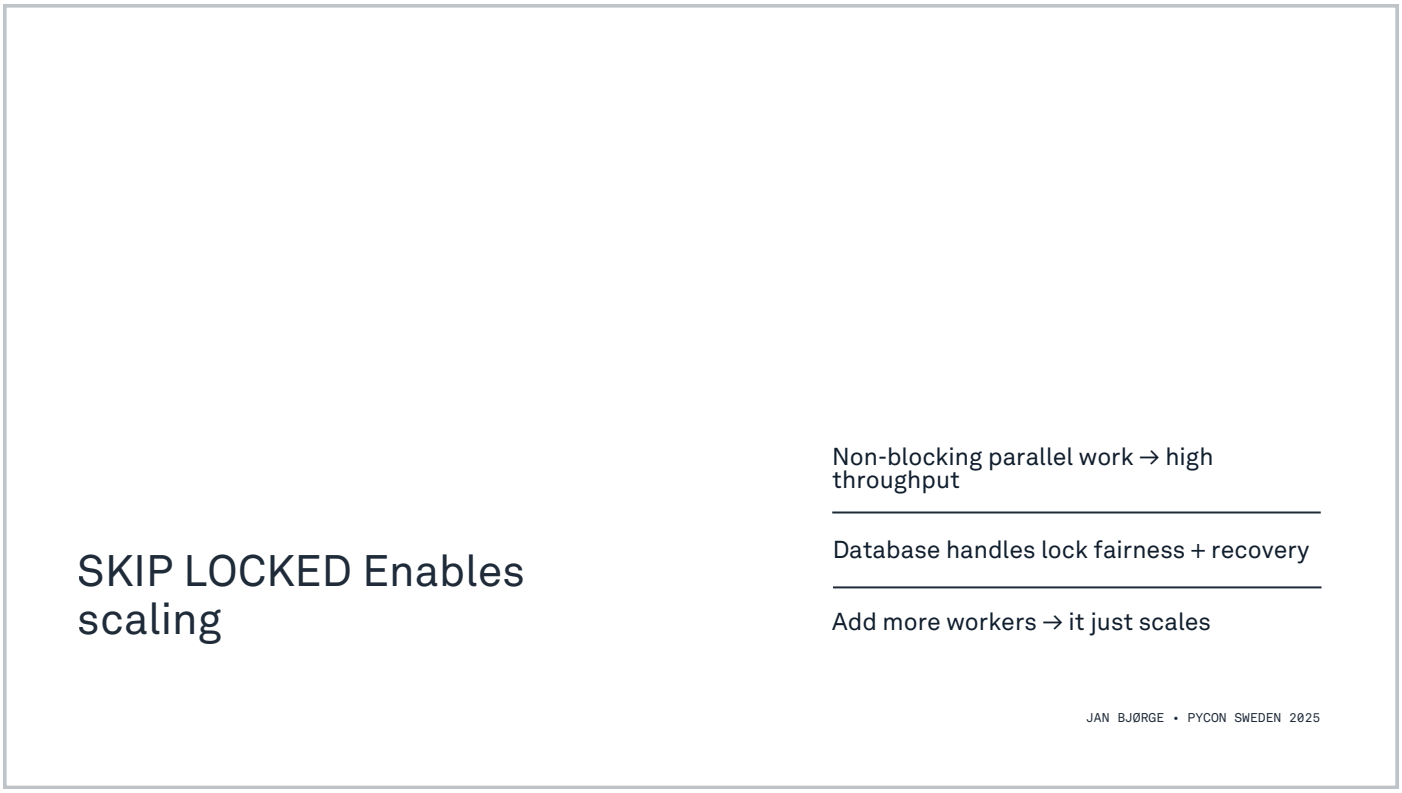
Table	Purpose
pgqueuer.jobs	Active + queued jobs
pgqueuer.jobs_log	Per-run trail (status, traceback, latency)
pgqueuer.statistics	Aggregated counters for dashboard + alerts
pgqueuer.schedules	Cron-style recurring jobs + their lease state

BOTH ARE ORDINARY TABLES → QUERY WITH SQL, NO BLACK BOX.

JAN BJØRGE • PYCON SWEDEN 2025

Native SQL keeps everything transparent.
You can run "SELECT COUNT(*) FROM WHERE status='queued'" and instantly see backlog, or even get a backlog per entrypoint by adding a group by.

Need other dequeue logic?Enhance the SQL to fit your needs, ex. ignore jobs that are older then 10 seconds?



Its super power is the usage of PgSQLs `SKIP LOCKED`.

It removes the need for external coordination - each worker safely skips busy rows. That’s how it scales horizontally with almost no code.

SAFE CONCURRENCY

Goal: multiple workers → no double processing.

```
WITH next AS (
  SELECT id
  FROM pgqueuer.jobs
  WHERE status='queued'
  ORDER BY priority DESC, id ASC
  FOR UPDATE SKIP LOCKED
  LIMIT $1
)
UPDATE pgqueuer.jobs j
SET status='picked', heartbeat=NOW(),
WHERE id=NOW() IN (SELECT id FROM next)
RETURNING *;
```

Each lock acts as a lease until commit.

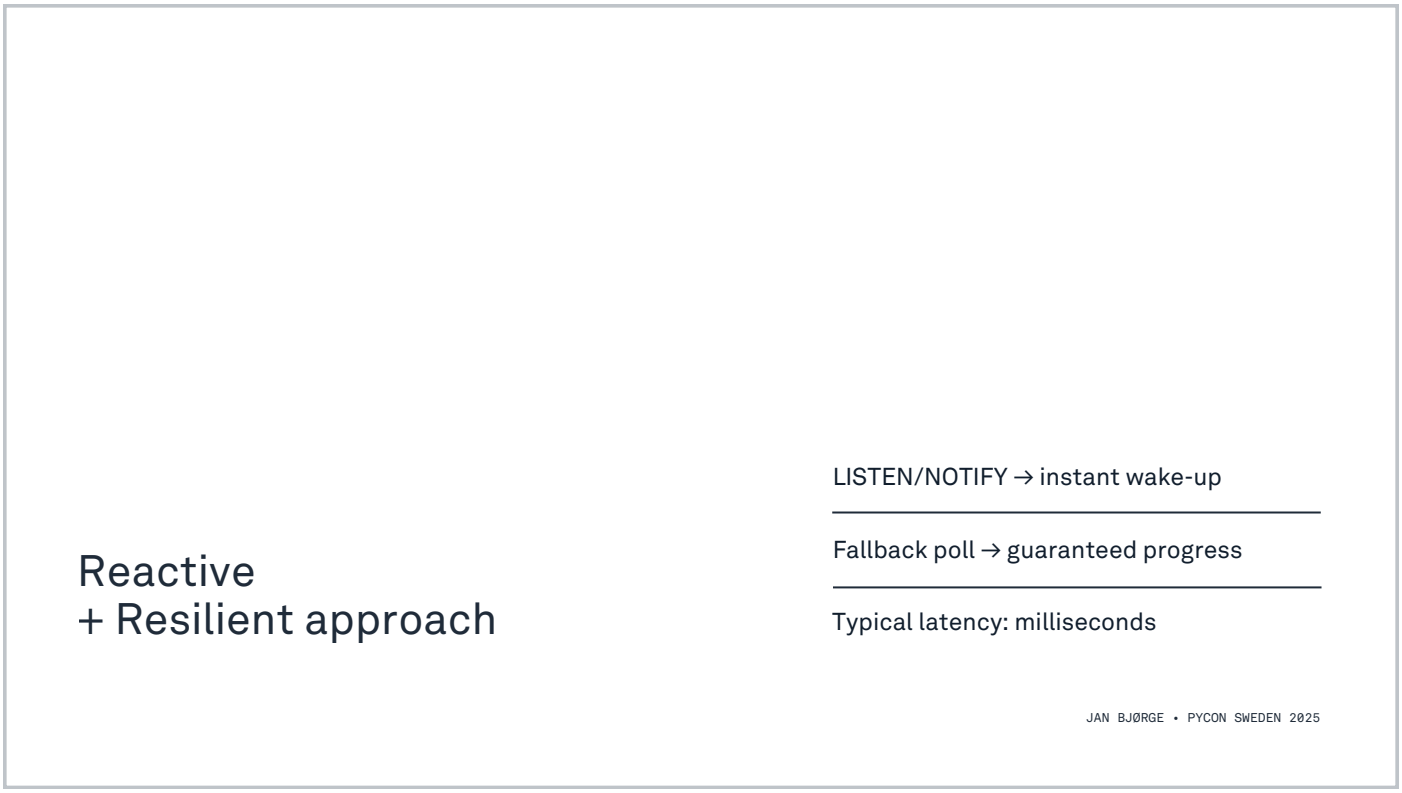
JAN BJØRGE • PYCON SWEDEN 2025

What SKIP LOCKED prevents. Imagine two workers hitting the database at the exact same millisecond, both querying for the next job.

Without `SKIP LOCKED`, they might both lock the same row, and one has to wait — causing worker starvation under load.

But with `FOR UPDATE SKIP LOCKED`, worker A locks row 42 and claims it; worker B sees row 42 is locked and instantly moves to row 43. Both make progress. Now multiply that by 50 workers - that's how it scales without contention.

And if a worker crashes mid-task? If the claim didn't commit, Postgres releases locks and the job remains available; no partial state persists.
Postgres does the hard lifting



Workers need to wake up when new jobs arrive, but we need this to be both fast and reliable. So we use a two-tier approach that combines the best of both worlds.

First tier is LISTEN/NOTIFY, which is Postgres's built-in pub/sub mechanism. When a job is enqueued, Postgres sends a NOTIFY signal on a channel. All workers that are listening on that channel receive the notification nearly instantly we're talking milliseconds. They wake up from their sleep, check for new jobs, and grab them. This is the happy path and it's very efficient. The connection stays open, there's no polling overhead, just a clean event-driven wake-up.

But NOTIFY is best-effort. If a worker is offline, it misses the signal. If the network connection resets or there's a blip in connectivity, the notification might not get through. So we have a second tier: a fallback poll loop. Workers periodically check the database for new jobs, typically a few seconds but you can configure this. If they missed a NOTIFY for any reason, they'll still pick up the job on the next poll cycle.

This dual approach gives you the speed of event-driven architecture without sacrificing reliability. In the happy path, latency is milliseconds. If something goes wrong with notifications, you lose at most a few seconds. You never lose jobs.

Triggers

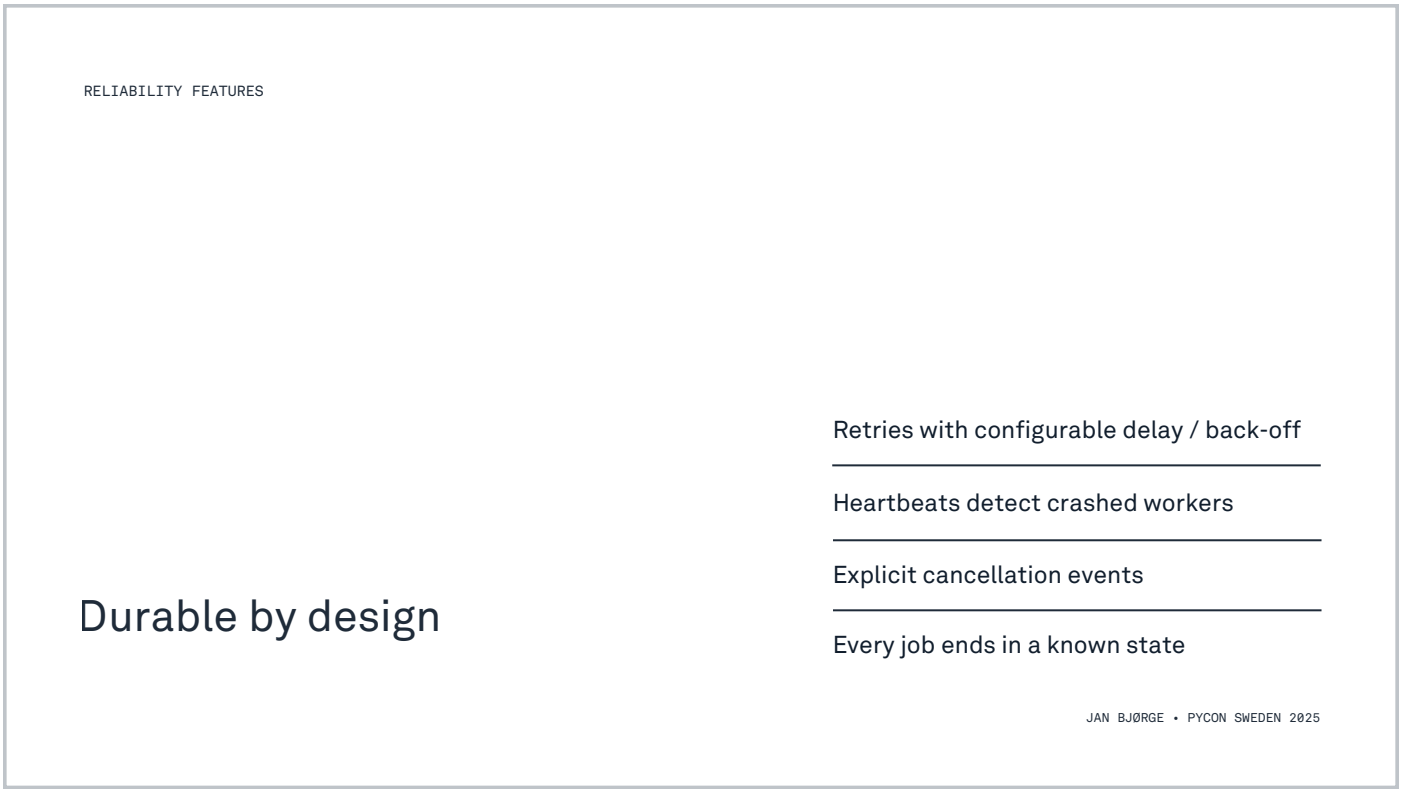
```sql
CREATE FUNCTION fn_pgqueuer_changed() RETURNS TRIGGER AS $$
    DECLARE
        to_emit BOOLEAN := false;  -- Flag to decide whether to emit a notification
    BEGIN
        -- Check operation type and set the emit flag accordingly
        IF TG_OP = 'UPDATE' AND OLD IS DISTINCT FROM NEW THEN
            to_emit := true;
        ELSIF TG_OP = 'DELETE' THEN
            to_emit := true;
        ELSIF TG_OP = 'INSERT' THEN
            to_emit := true;
        ELSIF TG_OP = 'TRUNCATE' THEN
            to_emit := true;
        END IF;

        -- Perform notification if the emit flag is set
        IF to_emit THEN
            PERFORM pg_notify(
                'ch_pgqueuer',
                json_build_object(
                    'channel', 'ch_pgqueuer',
                    'operation', lower(TG_OP),
                    'sent_at', NOW(),
                    'table', TG_TABLE_NAME,
                    'type', 'table_changed_event'
                )::text
            );
        END IF;

    END;
    $$ LANGUAGE plpgsql;

    CREATE TRIGGER tg_pgqueuer_changed
    AFTER INSERT OR UPDATE OR DELETE OR TRUNCATE ON pgqueuer
    EXECUTE FUNCTION fn_pgqueuer_changed();
```

JAN BJØRGE • PYCON SWEDEN 2025

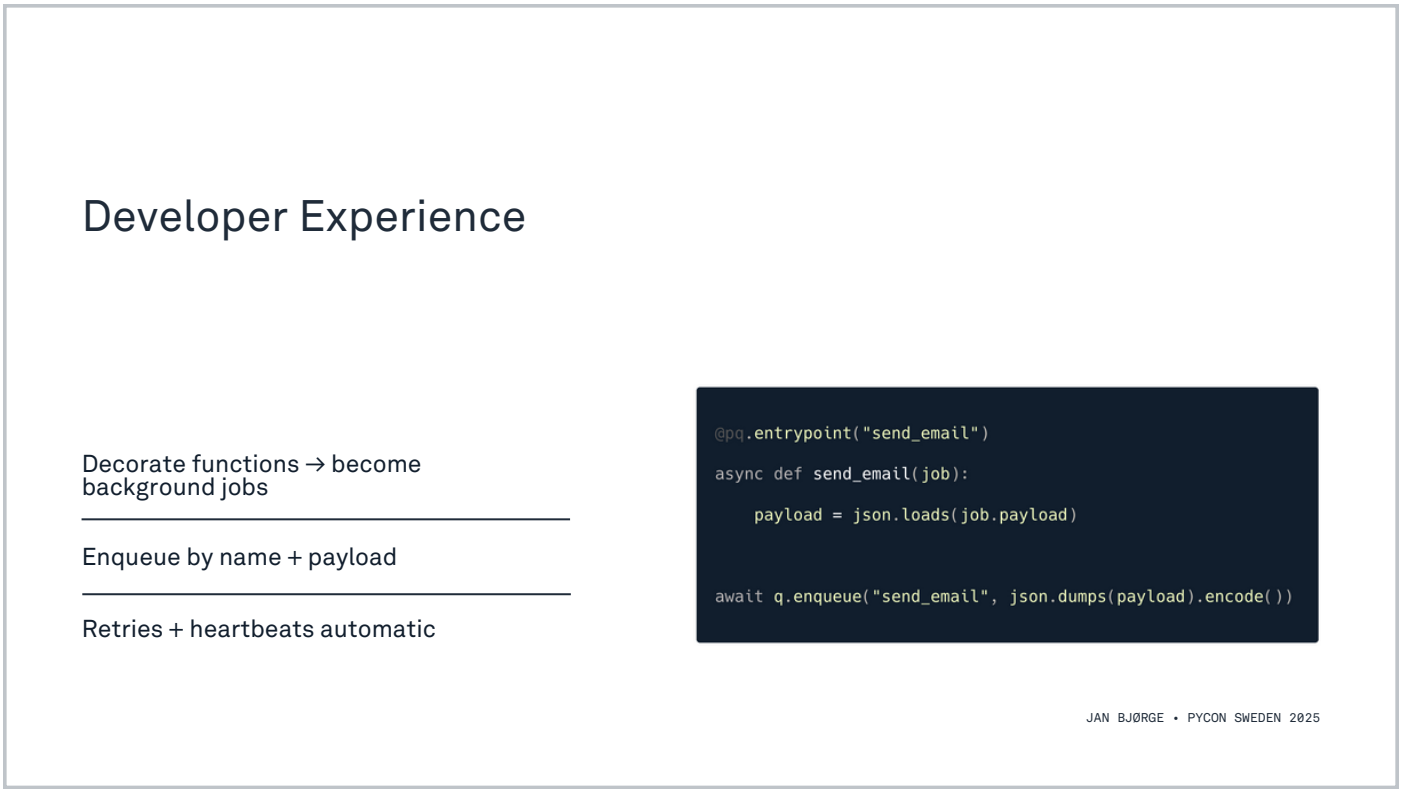
Now, how does the NOTIFY mechanism actually fire? Behind the scenes, a database trigger does the work. Every time a row is modified, this trigger fires automatically. The trigger function checks the operation type—was this an INSERT, UPDATE, DELETE, or TRUNCATE - and constructs a JSON payload describing what happened.

Then it calls pg_notify to broadcast that event on a channel called 'ch_pgqueuer'. All workers listening on that channel instantly receive the notification with the payload. They wake up and check for new work. The trigger is completely automatic—it runs whenever the jobs table changes, whether it's one job being enqueued or a batch of 10,000. No application code needs to explicitly call NOTIFY; the database handles it.

This is why NOTIFY is so reactive: the trigger fires as part of the same transaction that inserts or modifies the job. There's no delay, no polling, no batching lag. The moment a job hits the table, every listening worker gets the signal. That's how we achieve millisecond latency. .

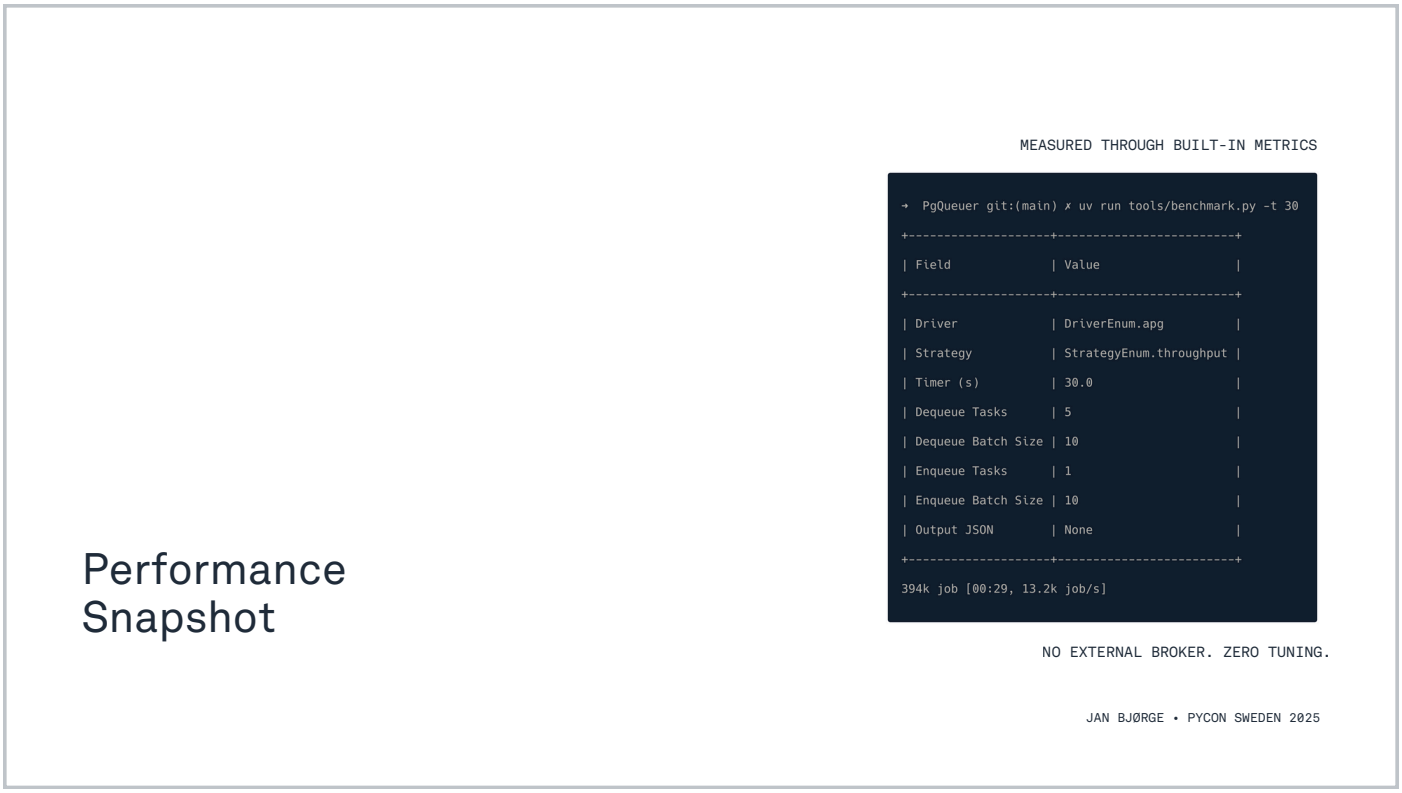


If a worker crashes, heartbeat timestamps detect it and the queue retries the job automatically. You can configure retry policies for specific jobs if you need them maybe retry up to 5 times with back-off. You can explicitly cancel jobs and workers stop gracefully. Every job ends in a known state: completed, failed, or cancelled. No mystery states. You query the database and know exactly where every job is.



To create a background job handler, you use the entrypoint decorator on an async function. You give it a name—that name becomes how you reference it when enqueueing work. Your function receives a Job object which contains the payload and metadata. It's just a regular async function.

When you want to queue work, you call the enqueue function with the job name and a byte payload. You encode however you want—JSON, pickle, doesn't matter. The framework doesn't care about the format. You can enqueue single jobs or batch multiple jobs in one call for efficiency. From the caller's perspective, it's a async function call that returns immediately.



These are numbers from our benchmark suite. We have two benchmark strategies: the throughput strategy and the drain strategy. What you're seeing here is the throughput strategy, which is designed to measure performance under static load.

Here's how it works: the benchmark runs for 30 seconds with both producers and consumers working simultaneously. On the consumer side, we have 5 independent workes running in parallel, each pulling batches of 10 jobs at a time from the queue. On the producer side, we have 1 task continuously enqueuing new jobs in batches of 10 as well.

The result you see: ~400k jobs processed in 30 seconds at ~13k jobs per second.

PGQueuer shines when your application already runs on Postgres and you need durable background jobs — things like sending emails, processing reports, cleaning data, or scheduling tasks.

It's built for independent jobs that don't depend on each other's timing. If your workflow needs strict ordering or event-driven chains, you're better off using a system designed for orchestration.

But for most day-to-day background work, keeping everything inside Postgres means fewer moving parts and simpler operations.

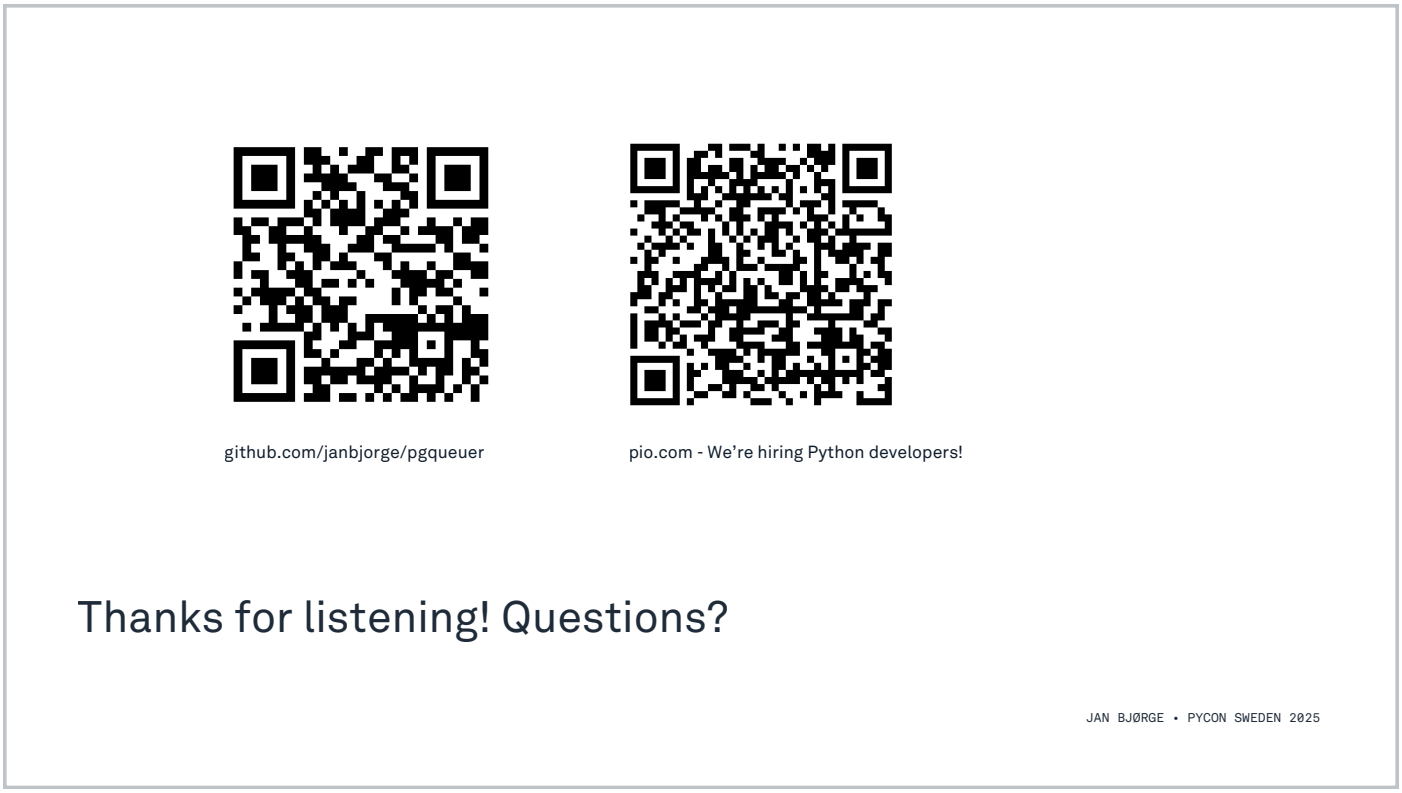
The biggest surprise wasn’t performance - it was reliability.
Postgres already handles concurrency, consistency, and crash recovery gracefully.
Sometimes the fastest way forward is to use what you already have and cut systems instead of adding them.

Demo

One-Line Anchor

> “Jobs as rows; locks as leases; NOTIFY for immediacy, polling for certainty.”

JAN BJØRGE • PYCON SWEDEN 2025



Thanks for joining!
If you try PGQueuer, let me know what works and what doesn't — feedback makes it better.
And if you're into Python + automation + robots, come talk to me as well.

Enjoy the rest of PyCon!