POSTHOG × ONE | RUST MEETUP

PostHog

From Snakes to Crabs

Rewriting PostHog's feature flag platform in Rust.

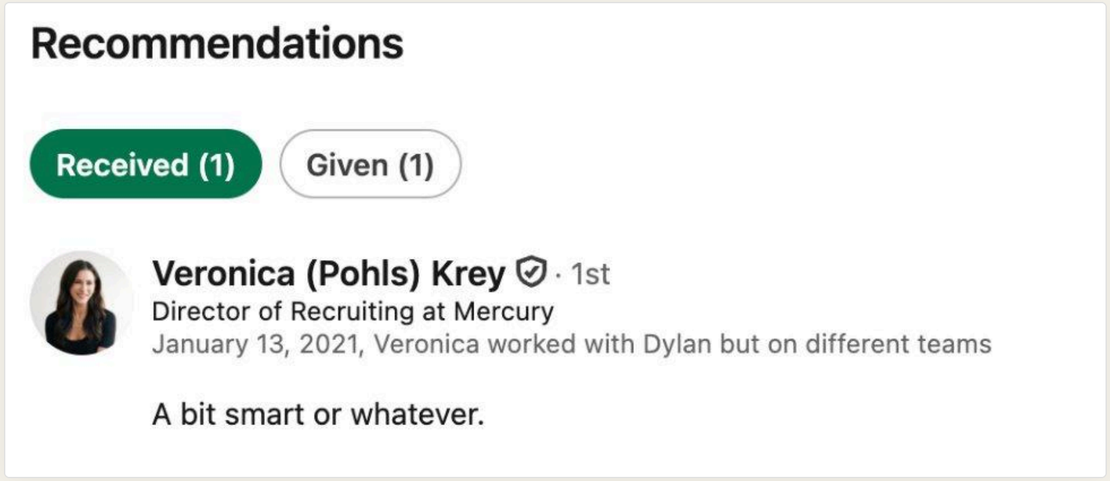
Dylan Martin

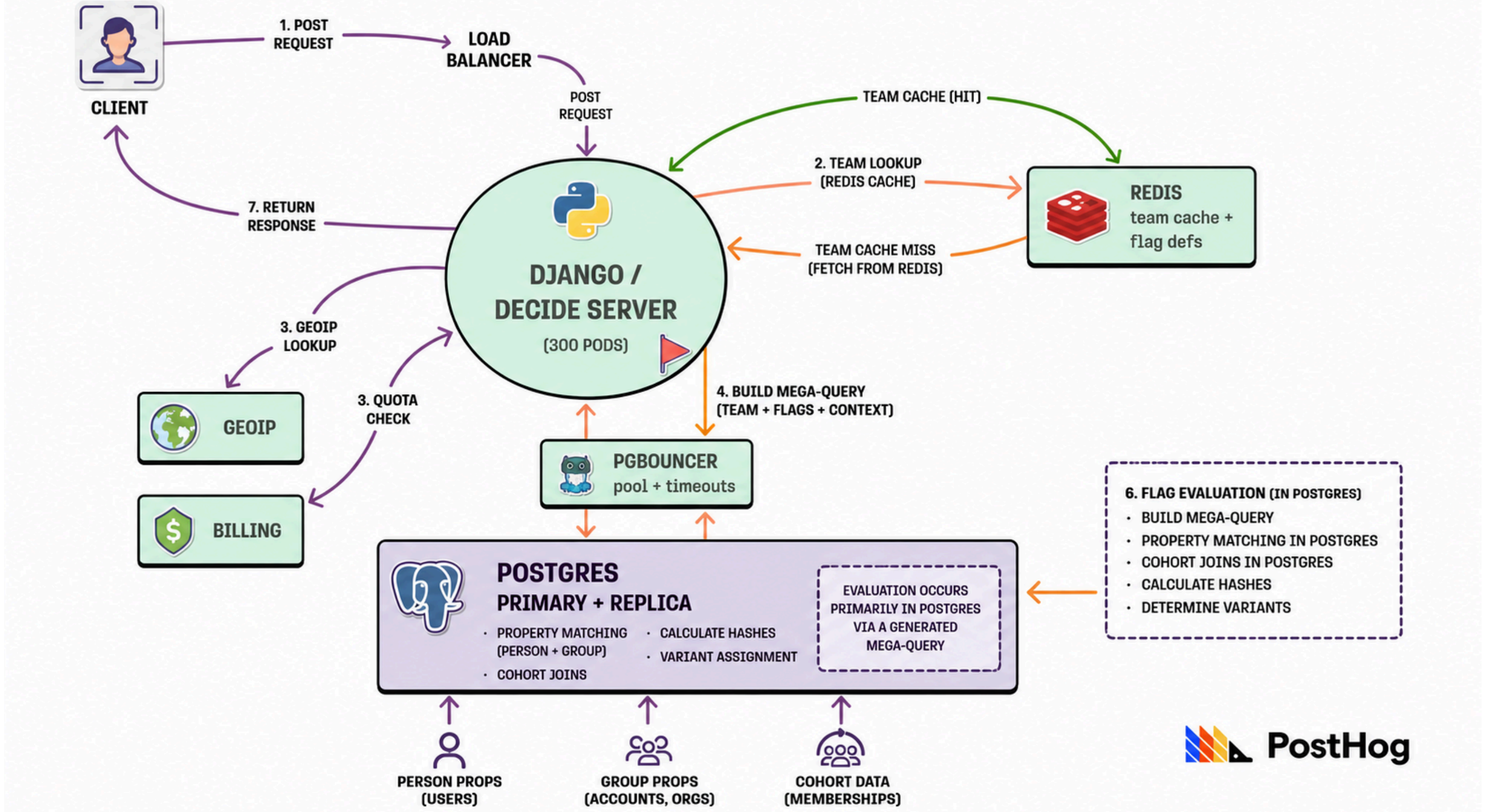
@dmarticus Tech lead, PostHog

WHO I AM

Dylan Martin

(Former) tech lead, Feature Flags @ PostHog




CLIENT
1. POST REQUEST
LOAD BALANCER
POST REQUEST
TEAM CACHE (HIT)
2. TEAM LOOKUP (REDIS CACHE)
REDIS
team cache + flag defs
TEAM CACHE MISS (FETCH FROM REDIS)
7. RETURN RESPONSE
DJANGO / DECIDE SERVER
(300 PODS)
3. GEOIP LOOKUP
GEOIP
3. QUOTA CHECK
BILLING
4. BUILD MEGA-QUERY (TEAM + FLAGS + CONTEXT)
PGBOUNCER
pool + timeouts
POSTGRES
PRIMARY + REPLICA
· PROPERTY MATCHING (PERSON + GROUP)
· COHORT JOINS
· CALCULATE HASHES
· VARIANT ASSIGNMENT
EVALUATION OCCURS PRIMARILY IN POSTGRES VIA A GENERATED MEGA-QUERY
6. FLAG EVALUATION (IN POSTGRES)
· BUILD MEGA-QUERY
· PROPERTY MATCHING IN POSTGRES
· COHORT JOINS IN POSTGRES
· CALCULATE HASHES
· DETERMINE VARIANTS
PERSON PROPS (USERS)
GROUP PROPS (ACCOUNTS, ORGS)
COHORT DATA (MEMBERSHIPS)
PostHog

SECTION 01

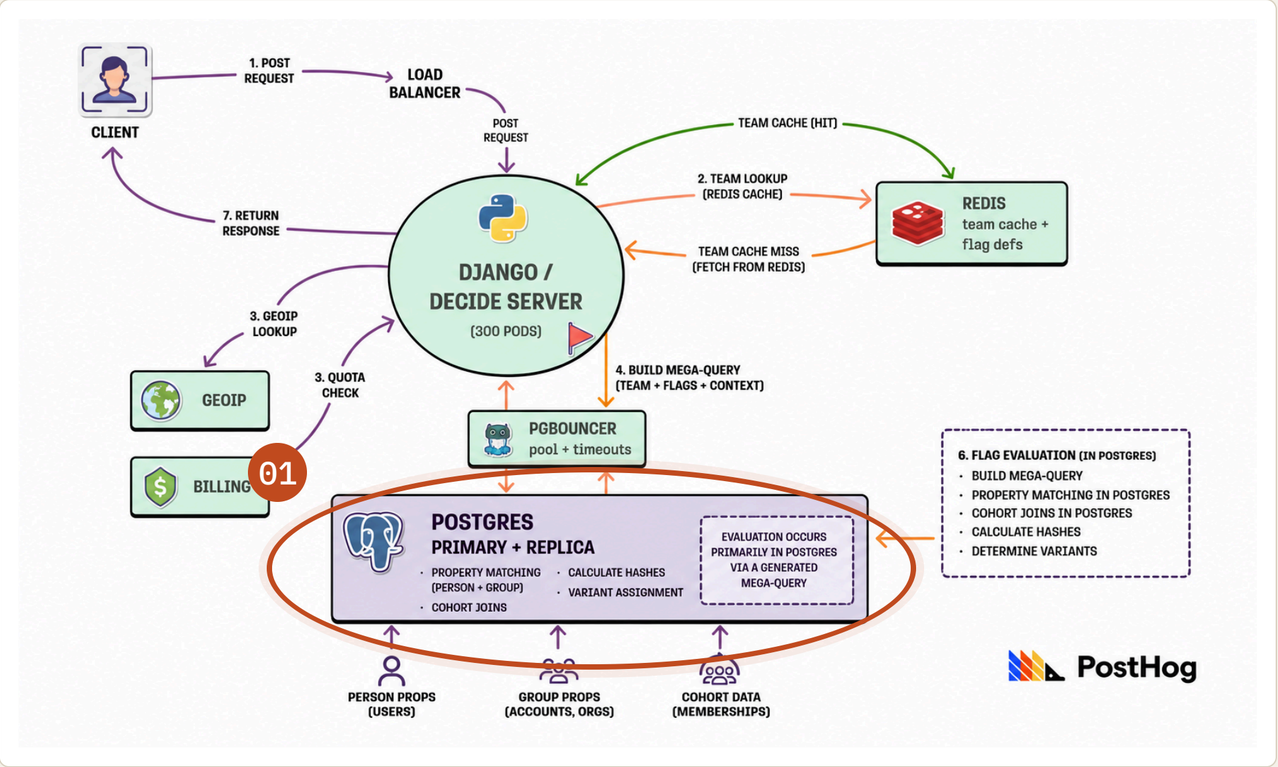
The problem.

What was structurally wrong with the old Python service.

Three things were wrong with the old service.

01

The ORM



Evaluating all flags was one big mega-query

```
-- ONE query per request. Conditions for EVERY flag,
-- evaluated inline by Postgres:
SELECT person.id, person.properties,
  (person.properties->>'email' LIKE '%@acme.com' AND
   (person.properties->>'plan')::text = 'enterprise') AS flag_a,
  (cohort.id = ANY($2)) AS flag_b,
  -- ... one expression per flag, for every flag
FROM person
  LEFT JOIN cohortpeople ON cohortpeople.person_id = person.id
  LEFT JOIN cohort ON cohort.id = cohortpeople.cohort_id
WHERE person.team_id = $1 AND person.id = $3;
```

Why that one query didn't scale.

01 JSONB matching can't use an index

02 Query cost grows with every flag

03 Operators with no SQL form

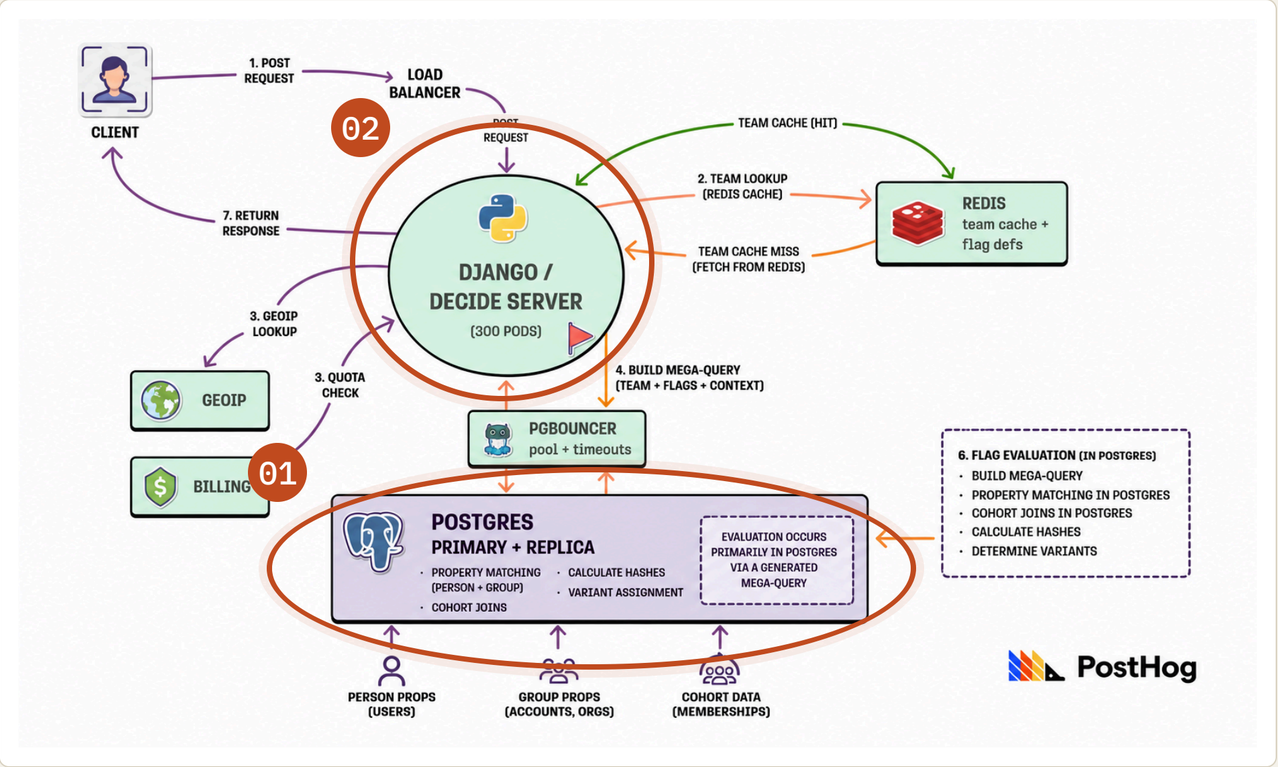
Three things were wrong with the old service.

01

The ORM

02

The runtime



Python couldn't use the box.

POD · 8 CORES

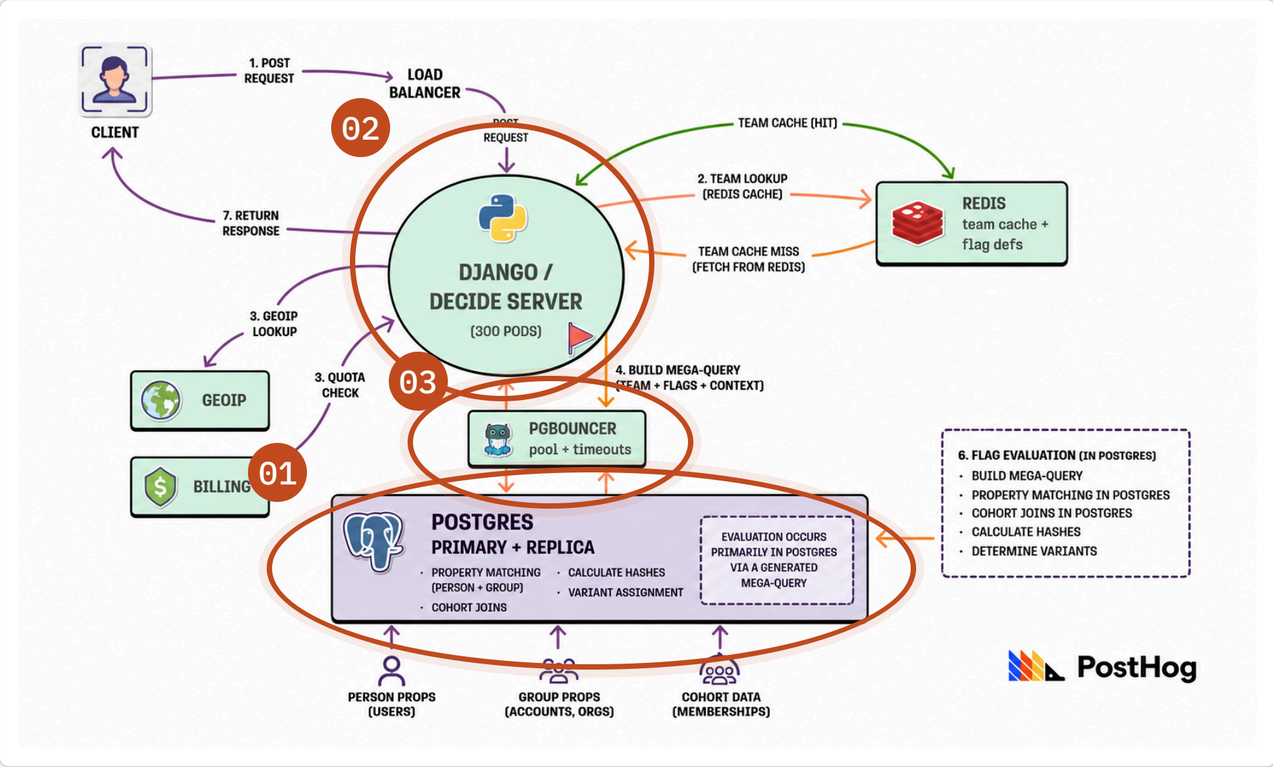
Sync workers block a whole process on the DB, and the GIL pins Python to one core per worker

Three things were wrong with the old service.

01
The ORM

02
The runtime

03
Connection pooling



Timeouts trapped in PgBouncer.

```
# Postgres caps out in the low hundreds, so thousands of
# workers funnel through PgBouncer. Our deadlines live there too:
query_timeout      = 15
query_wait_timeout = 10

# in the Django request handler:
# …no real per-request deadline we could reach.
```

A separate system, clunky to tune

SECTION 02

Choosing Rust.

What the replacement had to do, and why Rust cleared the bar.

What the replacement had to solve.

01 Evaluate flags outside the database

02 A genuinely fast async runtime

03 Connection pooling without PgBouncer

Why Rust won.

	Async Python FASTAPI	Node HYPEREXPRESS	Rust TOKIO
Eval off the ORM	No	Yes	Yes
Fast async runtime	Async, still slow	Async, slower than Rust	Fast and async
Pool without PgBouncer	No	Needs a proxy	Native, via sqlx
Team can ship it	Yes	Yes	Yes
VERDICT	SAME PROBLEMS, DEFERRED	FORCED INTO A PROXY	WON OR TIED ON EVERYTHING

SECTION 03

Build it without losing what we knew.

Tests were my best friend.

One module at a time.

01
Stand up the new endpoint

→

02
Rewrite a module

↓

03
Port its tests

←

04
Add integration tests

↑

↺

Then, test it with real data.

LOCAL BUILD

All prod flags + a test-bed of users

users built to match across every flag

OLD

Django, local

NEW

Rust, local

DIFF

Fix edge cases

↺ repeat

What changed in the new service.

01

Evaluation in app memory

Flag matching moved out of Postgres SQL — fetched raw, matched in parallel.

02

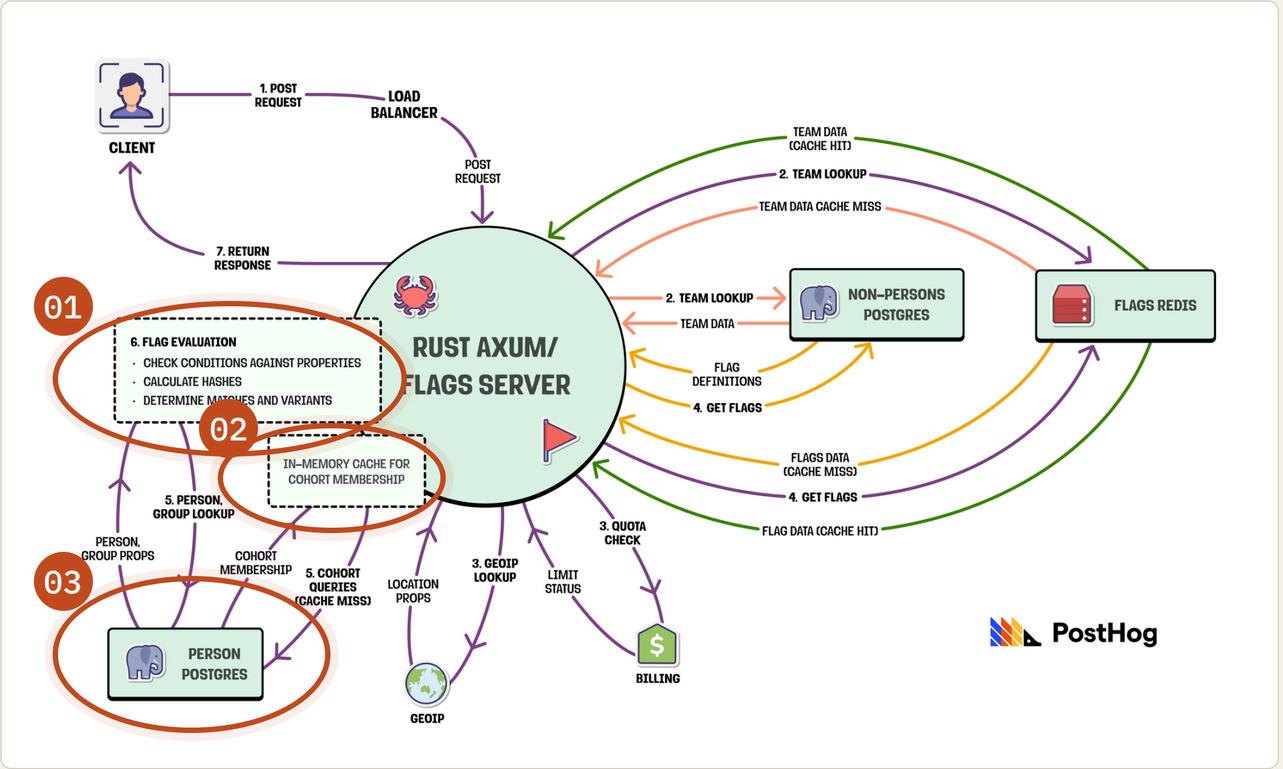
Cohorts cached in-memory

No re-joining membership on every request.

03

Direct pool management

Pools managed through sqlx at the app level. PgBouncer gone.



Pooling moves in-process.

```
// sqlx owns the pool — no PgBouncer in the path.
let pool = PgPoolOptions::new()
    .max_connections(20)
    .acquire_timeout(Duration::from_millis(50))
    .connect(&url).await?;

// And the deadline lives in our code, per request:
let row = timeout(Duration::from_millis(100), q.fetch_one(&pool)).await?;
```

One pool per pod, bounded and shared across tasks. **PgBouncer gone, timeouts back where we can reach them.**

Match every flag in parallel.

```
// Fetch raw properties once, then fan the matching
// out across cores — the thing the GIL never allowed.
let results: Vec<FlagMatch> = flags
    .par_iter()                          // rayon
    .map(|flag| flag.evaluate(&props))
    .collect();
```

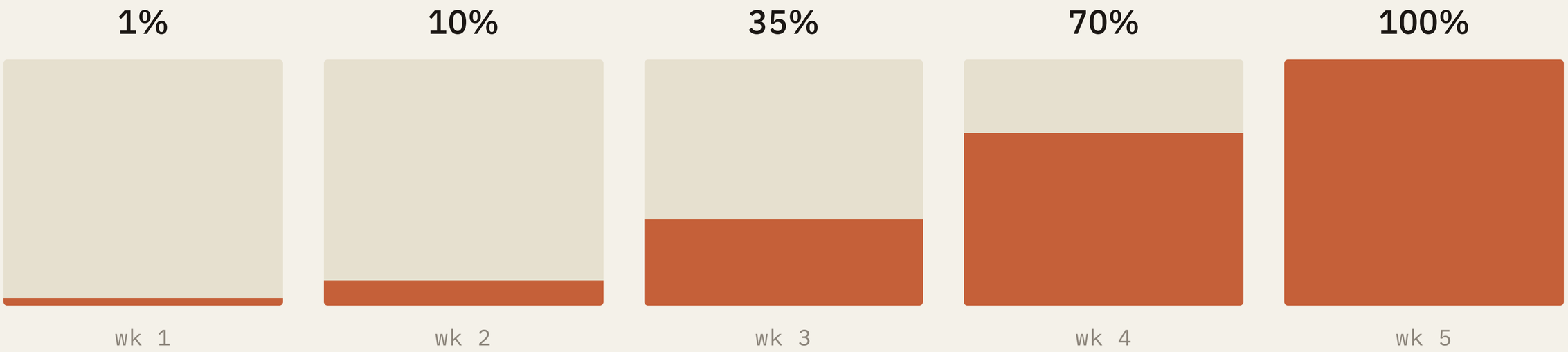
Postgres stops doing application work; the CPU-bound match runs on **all the cores, in memory**.

SECTION 04

Shipping it without breaking production.

Taking advantage of an existing service.

Roll out by hashing the API token.



SECTION 05

The results.

What the rewrite actually bought us.

Latency.

P99

~~904ms~~

85ms

10.6× faster · 90.5% reduction

Response time across the rollout. Cutover ≈ 21:15.

p95 381 → 43 ms p90 160 → 31 ms p50 22 → 12 ms

Cost.

PODS	300	→	90	−70%
COMPUTE SPEND	$8.8k	→	$2.8k	−68%
TRAFFIC	500k rpm	→	730k rpm	+46%

Bonus: here's now

P99 TODAY

25ms

Live dashboard · May 2026.

SECTION 06

Reflection.

What I'd do differently, and what to take home.

What I'd do differently.

Property-based testing.

What I'd do differently.

Property-based testing.

Load testing infrastructure.

What I'd do differently.

Property-based testing.

Load testing infrastructure.

Leverage AI more heavily.

END

PostHog

Thank you.
Questions?

READ MORE

Part 1 · posthog.com/blog/even-faster-more-reliable-flags

Part 2 · posthog.com/blog/untangling-tokio-and-rayon

FIND ME

dylanamartin.com

github.com/dmarticus

linkedin.com/in/dmarticus