DEVWEEK NEW YORK · 2026

PostHog

From Snakes to Crabs

Rewriting PostHog's feature flag platform in Rust.

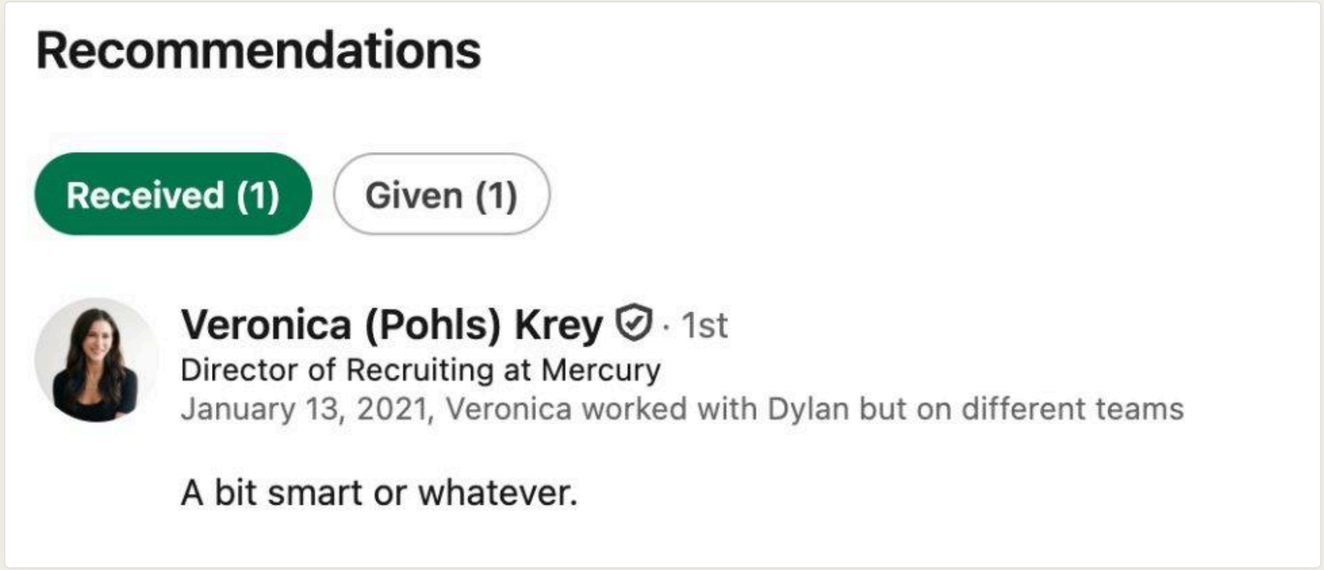
Dylan Martin

@dmarticus Tech lead, PostHog

WHO I AM

Dylan Martin

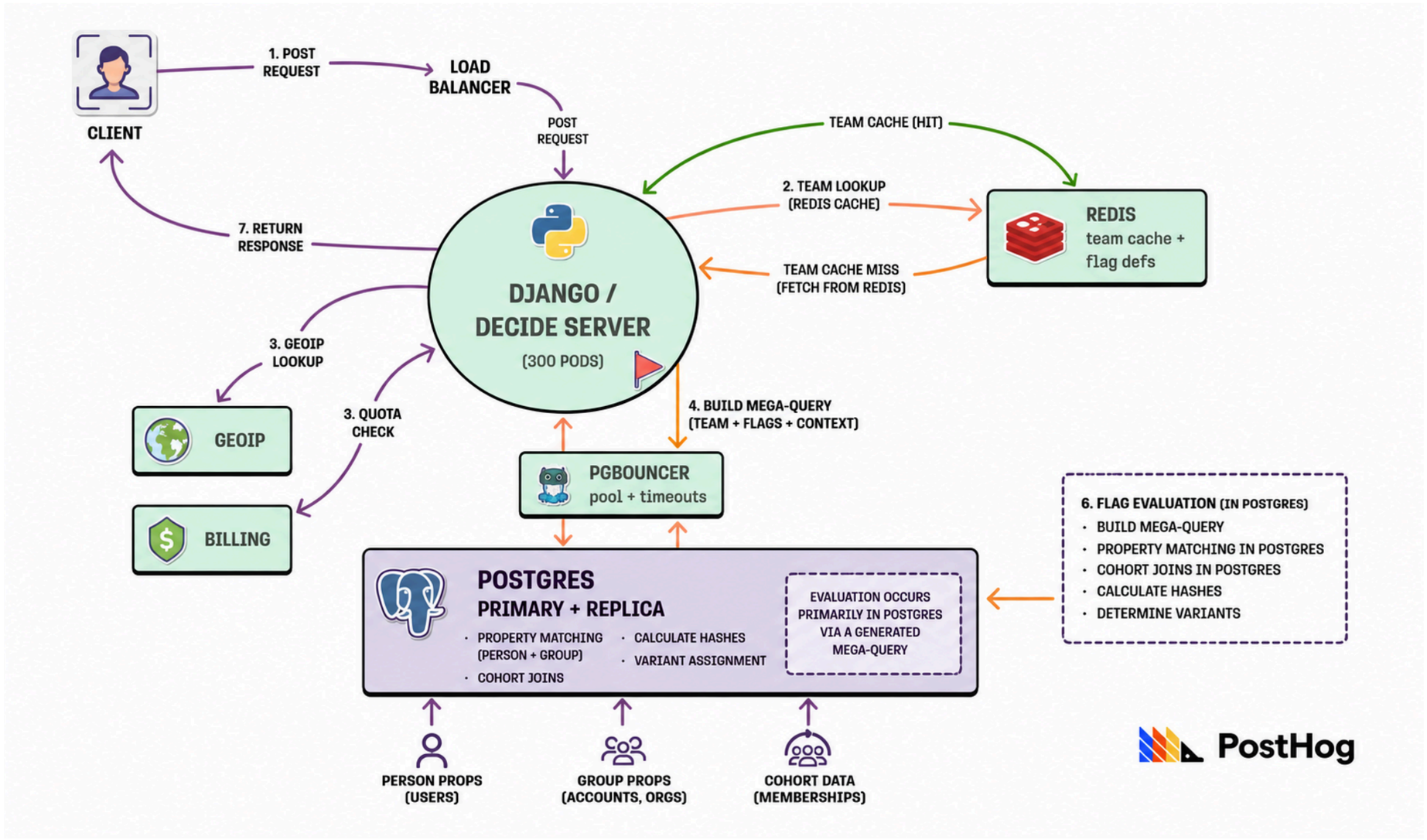
Former tech lead, Feature Flags @ PostHog



Feature flags fail differently.

MOST SERVICES

FEATURE FLAGS


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

WHERE WE'RE GOING

What I'm going to talk about.

01

The problem

What was structurally wrong with the old Python service.

02

Choosing Rust

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

03

Implementation

Tests were my best friend.

04

Rollout

Taking advantage of an existing service.

05

Results

What the rewrite actually bought us.

06

Reflection

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

SECTION 01

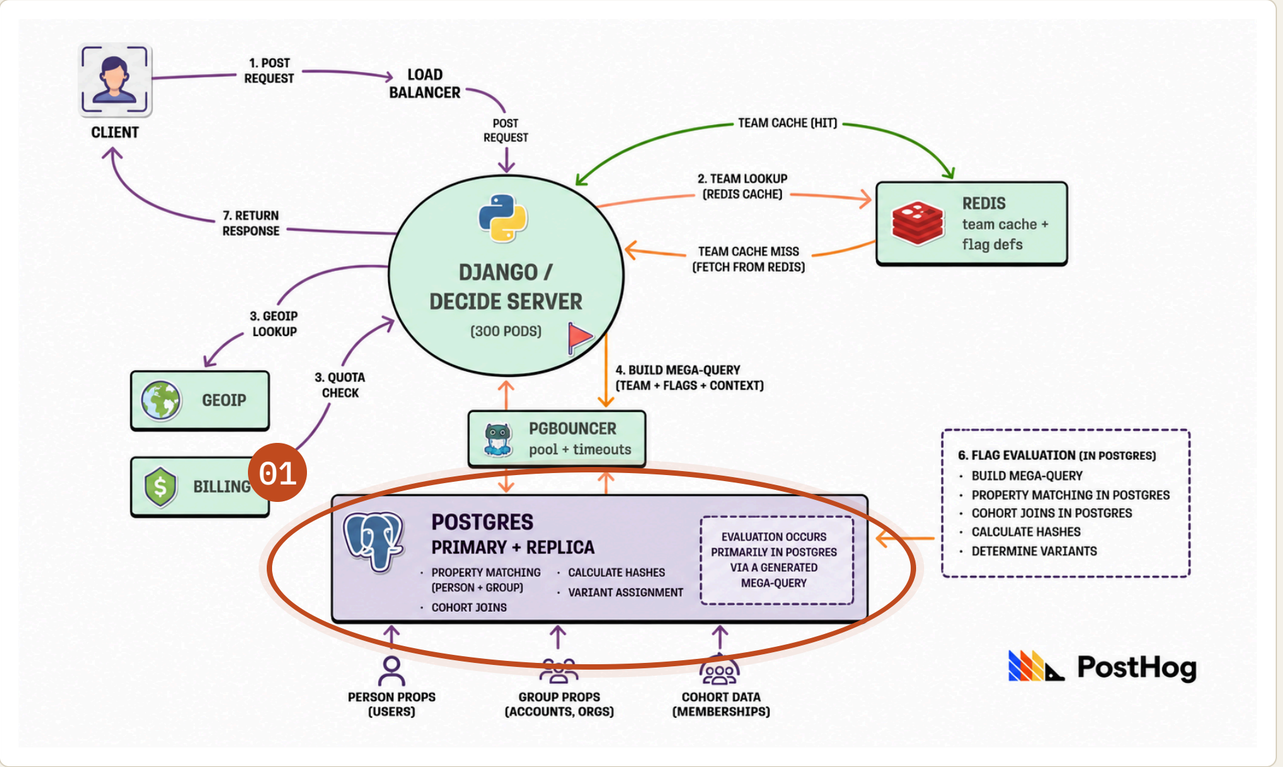
The problem.

What was structurally wrong with the old Python service.

Three things were wrong with the old service.

01

The ORM



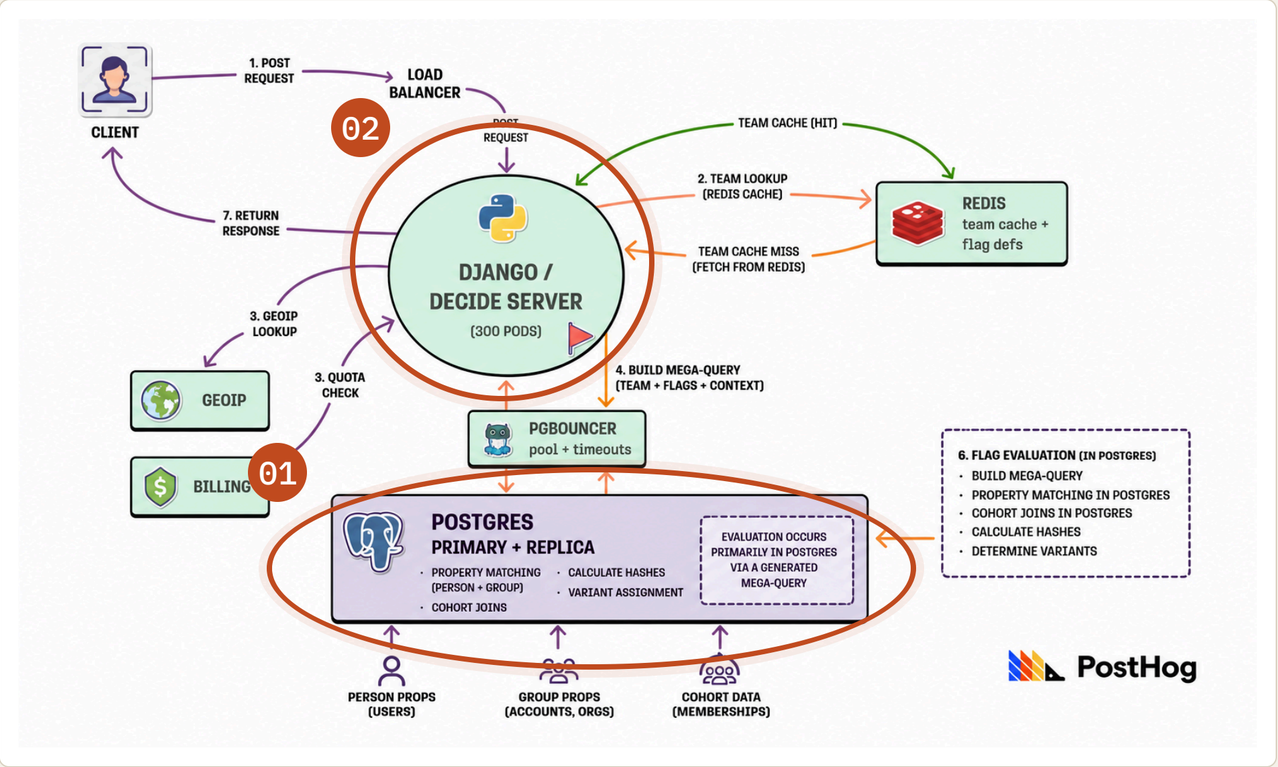
Three things were wrong with the old service.

01

The ORM

02

The runtime

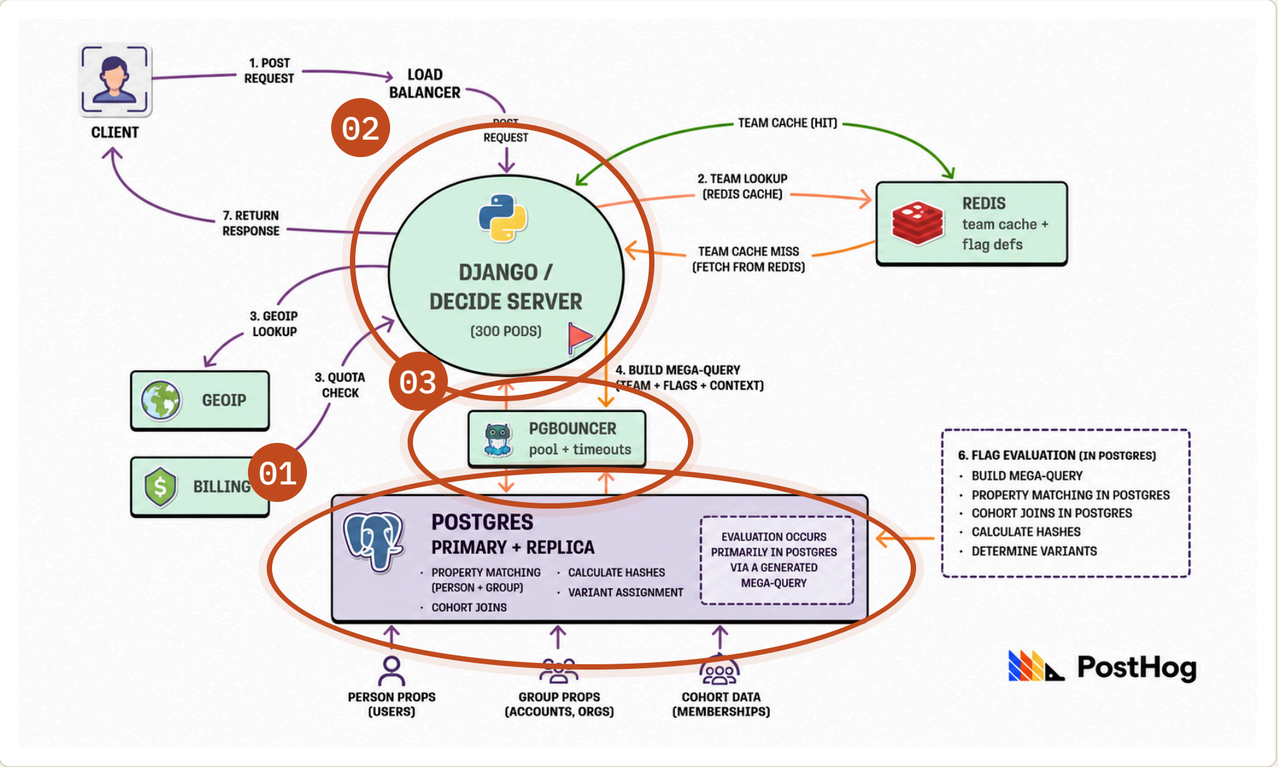


Three things were wrong with the old service.

01
The ORM

02
The runtime

03
Connection pooling



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;
```

JSONB matching can't use an index.

```
EXPLAIN — matching one flag's property:

Seq Scan on person  (cost=0.00..48000  rows=2000000)
  Filter: ((properties->>'plan') = 'enterprise')

-- A JSONB lookup can't use a btree. So Postgres reads
-- every row, on every request, for every flag.
```

The planner can't see through a cast.

```
-- Properties arrive as JSON text:
properties->>'age'   →   "25"   (text, not a number)

-- So every comparison casts, row by row:
(properties->>'age')::int > 21

-- The cast hides the column from the planner —
-- no stats, no index, consistently bad row estimates.
```

Query cost grows with every flag.

One inline expression per flag, all in a single generated query. **Parse, plan, execute in every request.**

Some operators have no SQL form.

Regex match	→	`hand-rolled SQL shim`
Version compare	→	`hand-rolled SQL shim`
Relative dates	→	`hand-rolled SQL shim`

One process per request.

REQUEST

One request comes in

→

WORKER PROCESS

Blocked · holds 50–150MB

idle while it waits

→

POSTGRES

The slow query

One core per process.

POD · 8 CORES

The GIL runs one core of Python at a time, per worker — adding cores to the box bought us nothing.

Postgres couldn't take the connections.

300 PODS

Thousands of workers

each wants a connection

→

PGBOUNCER

Multiplexes them down

→

POSTGRES

A few hundred, max

Timeouts lived in PgBouncer, not our code.

```
# pgbouncer.ini — where our deadlines actually lived:
query_timeout      = 15
query_wait_timeout = 10
# a separate system, clunky to tune

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

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

Decouple property fetching from flag computation and ideally get off Django's ORM.

02

A genuinely fast async runtime

Python's async works, but the runtime is still slow.

03

Connection pooling without PgBouncer

Share pools in-process, so timeouts can live in our code.

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

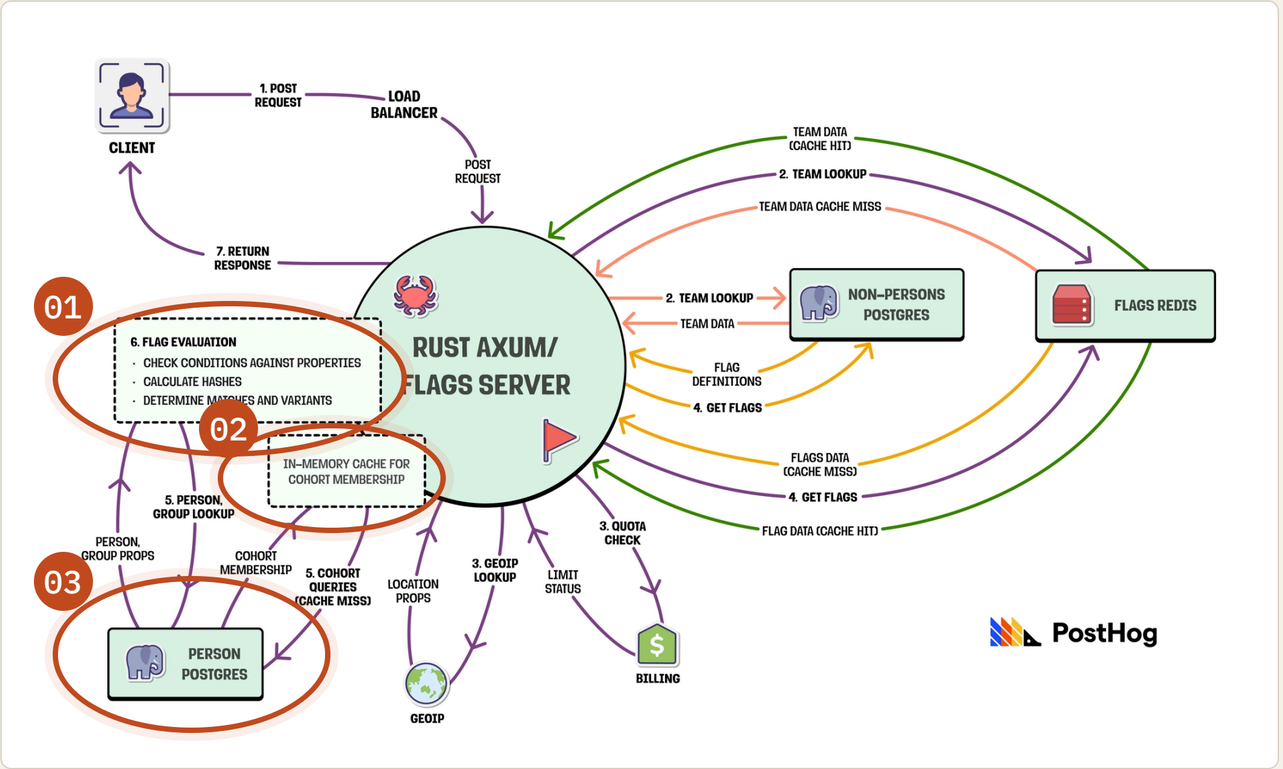
Cohorts cached in-memory

No re-joining membership on every request.

03

Direct pool management

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

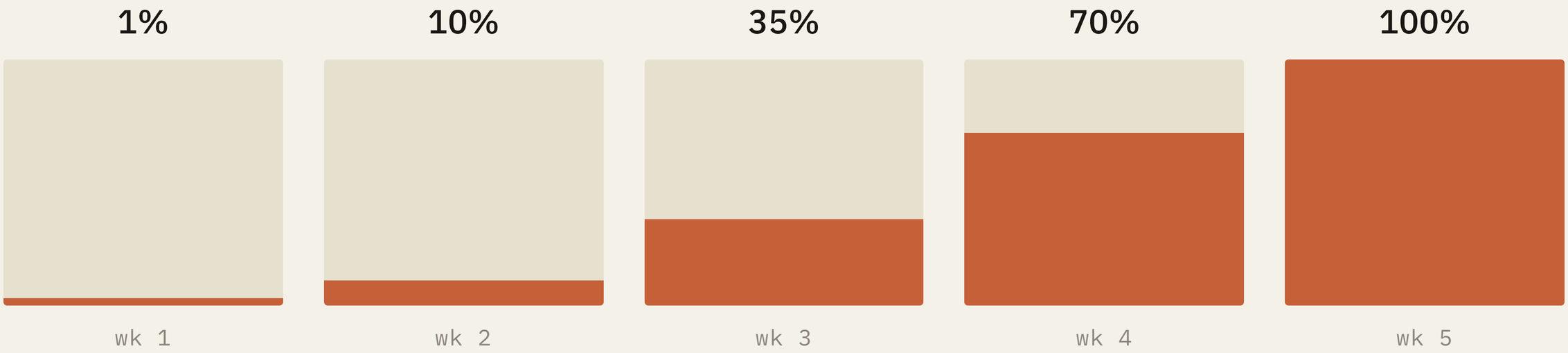


SECTION 04

Shipping it without breaking production.

Taking advantage of an existing service.

Roll out by API token. Slowly. Watch everything.



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%

Reliability.

THREE MONTHS POST-CUTOVER

Zero feature flag outages.

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.

Instrument the whole stack, not just the app.

What I'd do differently.

Property-based testing.

Load testing infrastructure.

Instrument the whole stack, not just the app.

Leverage AI more heavily.

Three takeaways.

01

Nailing the architecture up front was worth the effort.

02

Old tests are the best tool for preserving assumptions.

03

A gradual rollout on existing traffic is a good substitute for a load test.

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