

From Snakes to Crabs

Lessons learned rewriting PostHog's feature flags from Python to Rust



Dylan Martin

(former) feature flags team lead



PostHog

Dylan Martin

(former) feature flags team lead at PostHog



- Led the team that rewrote PostHog's feature flags in Rust
- "The last 10 percent takes 90% of the work"
- Static types aficionado; former functional programmer
- I write about this stuff at dylanamartin.com



Agenda

- **The rewrite**
why we replaced Python with Rust, and what the rewrite gained
- **The mystery**
the service degraded unexpectedly as load grew
- **The lessons**
how to use concurrency and parallelism safely in Rust

The Django service, ca. 2024

~500k

requests per minute

300

Pods to serve them

\$8.8k

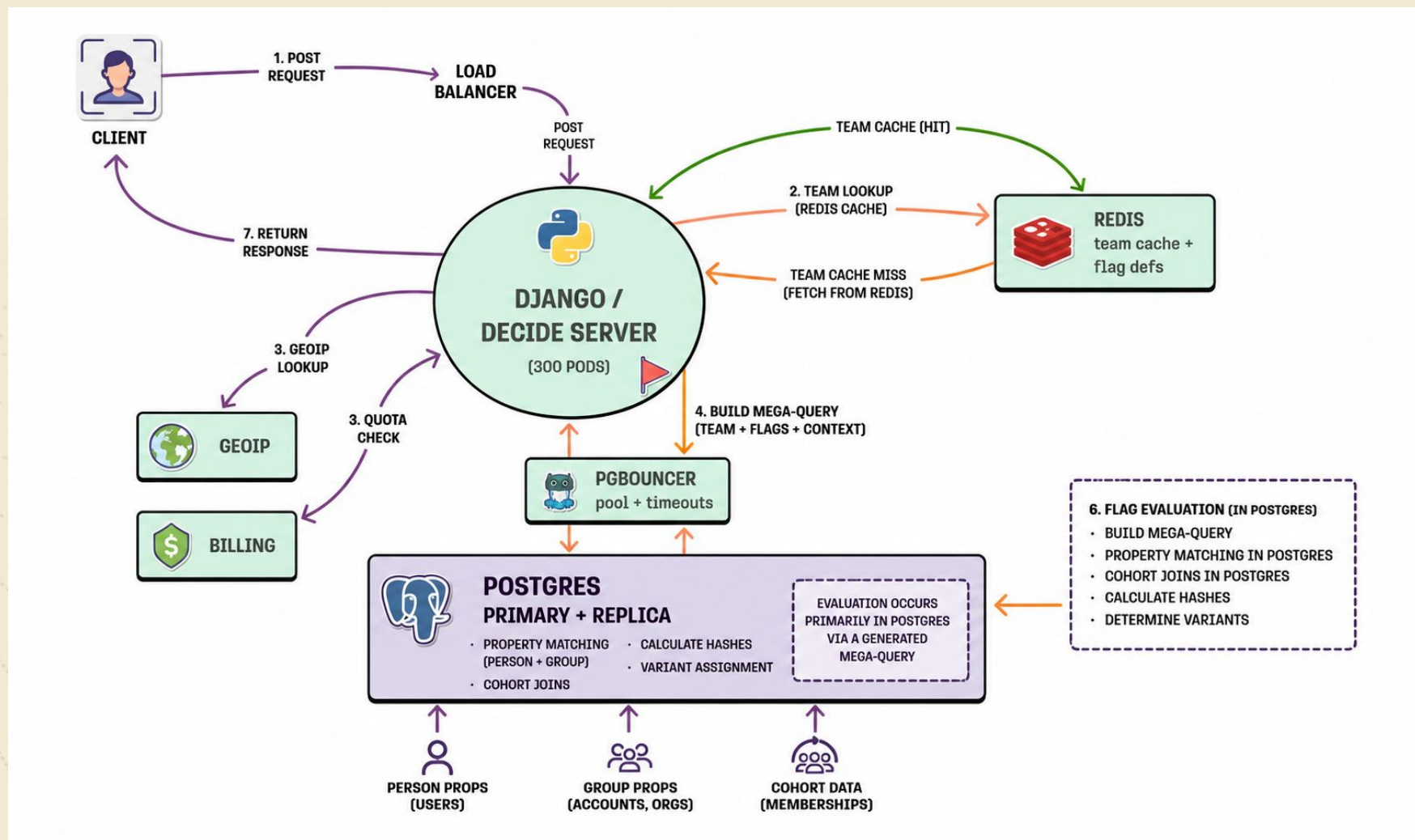
per month in compute

904ms

p99 latency

We weren't scaling well, our P90-P99 had unacceptable latency, and we didn't have a plan for our projected hypergrowth

The Django service, ca. 2024 (cont.)



Outside of CPU constraints, we had architecture problems

Evaluation lived in SQL

the database parsed flag conditions and transferred unnecessary data

Cohorts needed expensive joins

each evaluation computed membership with joins across many tables

Reliability hung on PgBouncer

no timeouts in code: pooler settings failed when the database was slow

We needed a rewrite.

No amount of tuning would fix the architecture.

How we selected Rust

FastAPI + async Python

easy migration, small performance gain, no timeout primitives

Node.js + HyperExpress

good benchmarks, but it depends on a C++ binding with a small community

Rust + axum

new code, but ~32k req/s vs ~1.5k for Django, a strict type system, and timeouts in code

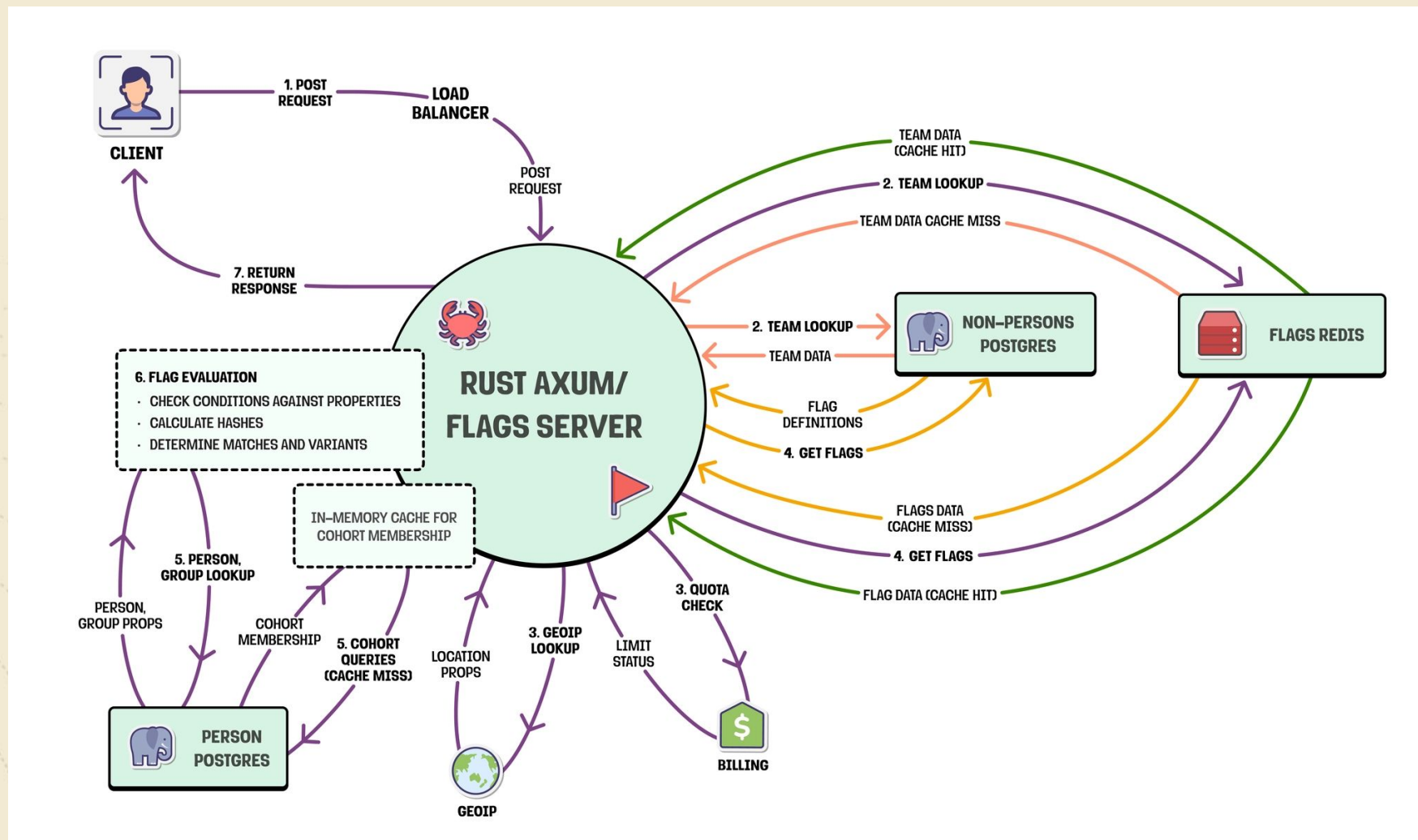
21×

benchmark throughput

What changed

- **Rewrote it in Rust**
a new axum service with strict types
- **Moved evaluation into memory**
one query for person properties, then evaluate every condition in parallel
- **Cached cohort membership**
an application cache makes cohort flags as fast as property flags
- **Removed PgBouncer**
the application connects directly and controls pooling and timeouts

The new architecture



Latency results

percentile	Django	Rust	speedup
p99	904ms	85.4ms	10.6x
p95	381ms	42.9ms	8.9x
p90	160ms	31.2ms	5.1x
p50	21.7ms	11.8ms	1.8x

(measured in production with the same traffic patterns)

Cost and reliability results

300 → 90

Pods for the same traffic

−68%

compute cost: \$8.8k → \$2.8k/mo

0

flag outages in the next 3 months*

*we published a blog post about this record, then had one outage. c'est la vie.

Then traffic doubled.

p99 latency spiked to 2.5 seconds. The application froze at random times.

First hypothesis: the database

The symptoms

connection pools maxed at 100%, and acquire times spiked in lockstep with p99

The data

Aurora showed fast, stable queries and normal CPU. No pressure at all.

Something was making the database *appear* slow

The clue: CPU throttling

More pods did not help

pod count did not change latency at all. The problem was inside every pod.

CFS throttling spiked with p99

mean CPU stayed low. Short, violent bursts of CPU demand, not sustained load.

The I/O runtime was frozen

the app was too frozen to even ask the database for a connection. The pool metrics were a downstream effect.

Two runtimes, one CPU budget

Tokio

waits for many things at once

async I/O: HTTP, database, cache. Starts one worker thread per CPU core.

Rayon

does many things at once

parallel computation on large data sets. Also starts one thread per CPU core.

In one container: two times too many threads. But oversubscription was only the accomplice.

The culprit

```
// inside the Tokio request handler let results:  
Vec<_> = flags_to_evaluate  
.par_iter() // blocks the I/O worker  
...
```

the I/O worker thread is blocked while Rayon processes the flags

a blocked worker cannot poll futures: no HTTP, no database, no cache

when most workers block, the application freezes and returns 504 errors

Every request used Rayon with no backpressure. This caused thread contention.

The fix: spawn, don't block

```
let (tx, rx) = oneshot::channel(); rayon::spawn (move ||  
{  
    let processed = flags  
        .into_par_iter()...collect();  
    let _ = tx.send(processed);  
});  
let res = rx.await; // worker is not blocked
```

Budget the threads

set pool sizes from the CPU request, not the limit
(available_parallelism() reads the cgroup limit)

Add backpressure

a semaphore limits Rayon to N batches at one time

Do not parallelize small requests

requests with fewer than ~200 flags run sequentially. 85 to 90% of requests are below this limit

Second fix: the dependency graph

Before

the code built the graph from all flags in the system, then scanned it repeatedly to find the evaluation order: $O(n^2)$

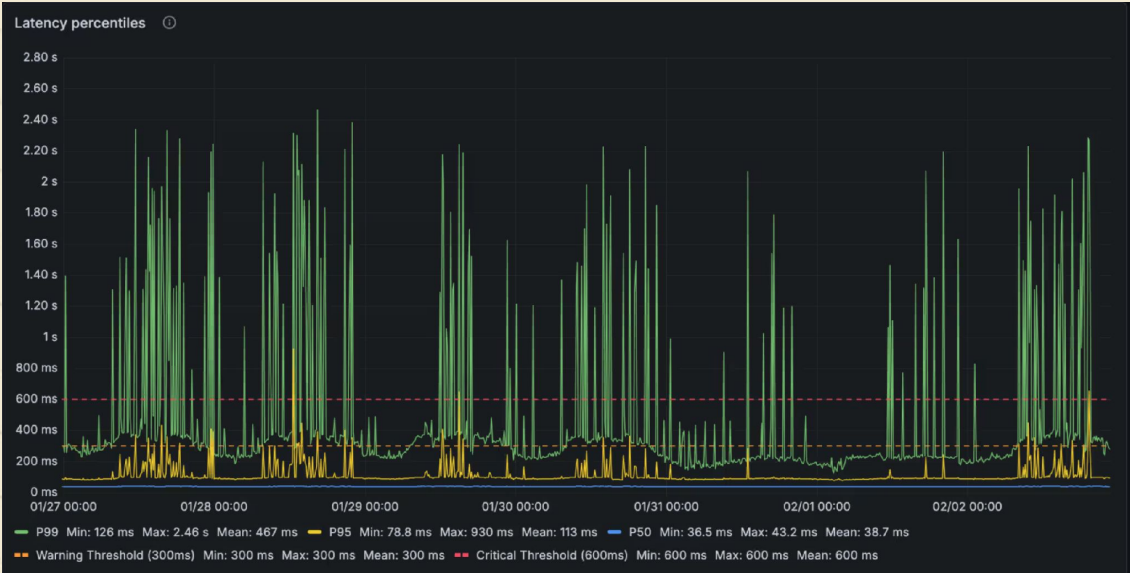
After

a BFS closure includes only the requested flags. Kahn's algorithm finds the order in one pass: $O(V+E)$

This uses less memory and makes the latency more stable.

p99, one week apart

Before: spikes to 2.5s



After: flat at ~94ms



The results

460 → 94ms

mean p99, with spikes below 200ms

~10%

pool utilization, acquire p99 ~4ms

~13k req/s

on a constant fleet of 50 pods

3am ≈ peak

p99 is the same at low traffic and at peak traffic

What we learned

- **Do not trust the symptoms**
the metrics pointed to the database, but the cause was a blocked I/O runtime
- **Do not parallelize small work**
most requests got no benefit from parallel dispatch. A threshold fixed this
- **Do not trust library defaults**
each thread pool claimed all cores in a limited container. Set the thread counts yourself
- **The best optimization is doing less**
fewer queries, fewer dependencies, fewer failure modes

P99 CONF

Thank you! If you want to connect...



Dylan Martin

dylan@posthog.com

dylanamartin.com

linkedin.com/in/dmarticus

@dmarticus on all socials



Further reading

posthog.com/blog/even-faster-more-reliable-flags

the rewrite: architecture, results, and the cutover plan

posthog.com/blog/untangling-rayon-and-tokio

the full debugging story, with the actual PRs

ryhl.io/blog/async-what-is-blocking

the definitive guide to the spawn-don't-block pattern