PostHog

From Snakes to Crabs

Lessons learned rewriting PostHog feature flags from Python to Rust



Dylan Martin

(former) feature flags team lead



Dylan Martin

(former) feature flags team lead at PostHog



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

Agenda

The rewrite
why we rewrote, and what we gained

The mystery
the service degraded unexpectedly as load grew

The lessons
what we learned from the rewrite and the subsequent novel failure modes



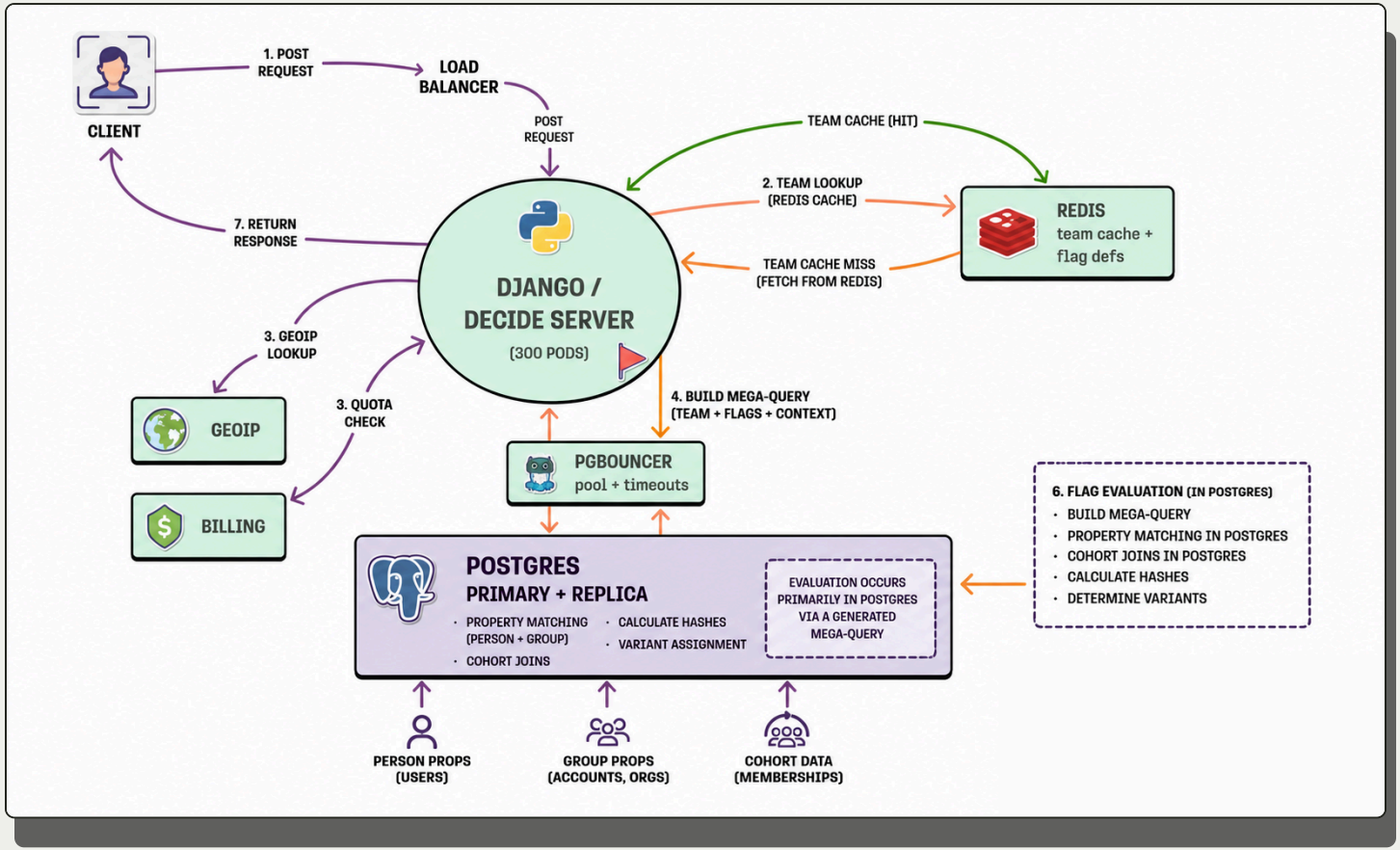
The Django service, ca. 2024

~500k	requests per minute
300	pods to serve them
$8.8k	per month in compute
904ms	p99 latency

The service was expensive, unwieldy, and slow.



The Django service ca. 2024 (cont)



Two kinds of problems

The architecture

how the service used Postgres

evaluation in SQL, cohort joins, and timeouts left to PgBouncer

The runtime

what Python and Django could do

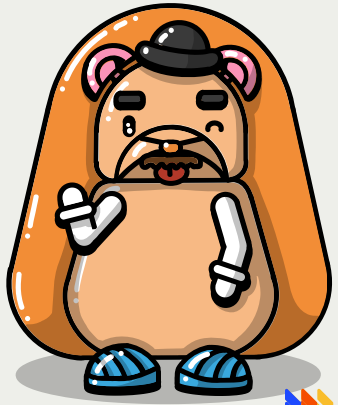
concurrency and CPU-bound work



Problem 1: the architecture

Evaluation lived in SQL

the database parsed flag conditions and transferred unnecessary data



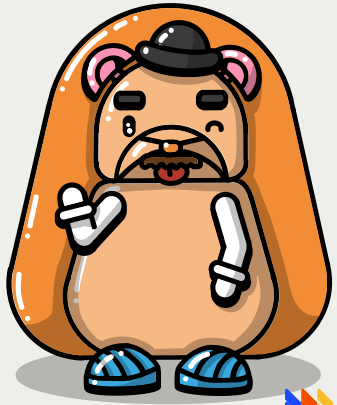
Problem 1: the architecture

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



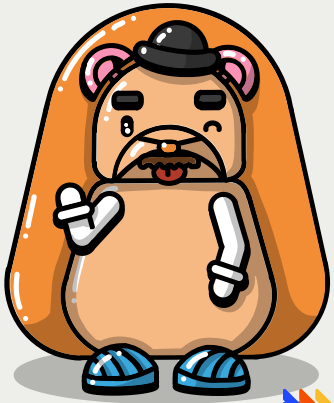
Problem 1: the architecture

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

Fixable in Django, but we'd still need 300 pods.



Problem 2: the runtime

Django was bad at concurrency

sync workers serve one request at a time, so more traffic meant more pods



Problem 2: the runtime

Django was bad at concurrency

sync workers serve one request at a time, so more traffic meant more pods

Python was slow at CPU-bound work

flag evaluation is hashing and condition matching, the work Python does slowest

We needed a different approach to shrink the fleet.

We needed a rewrite.

No amount of Kubernetes would fix the runtime.



How we selected Rust

benchmark throughput vs Django

FastAPI + async Python 7.5x faster

easy migration, but still Python, and no timeout primitives



How we selected Rust

benchmark throughput vs Django

FastAPI + async Python **7.5x faster**

easy migration, but still Python, and no timeout primitives

Node.js + HyperExpress **18x faster**

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



How we selected Rust

benchmark throughput vs Django

FastAPI + async Python 7.5x faster

easy migration, but still Python, and no timeout primitives

Node.js + HyperExpress 18x faster

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

Rust + axum 21x faster

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



So, we rewrote it.



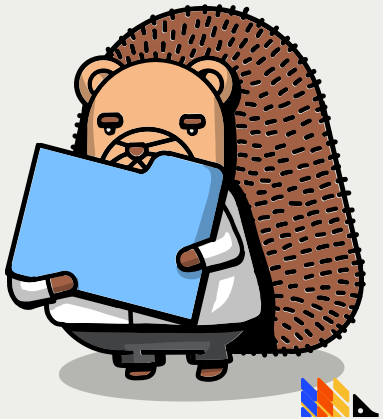
What changed

Rewrote it in Rust
a new axum service with strict types

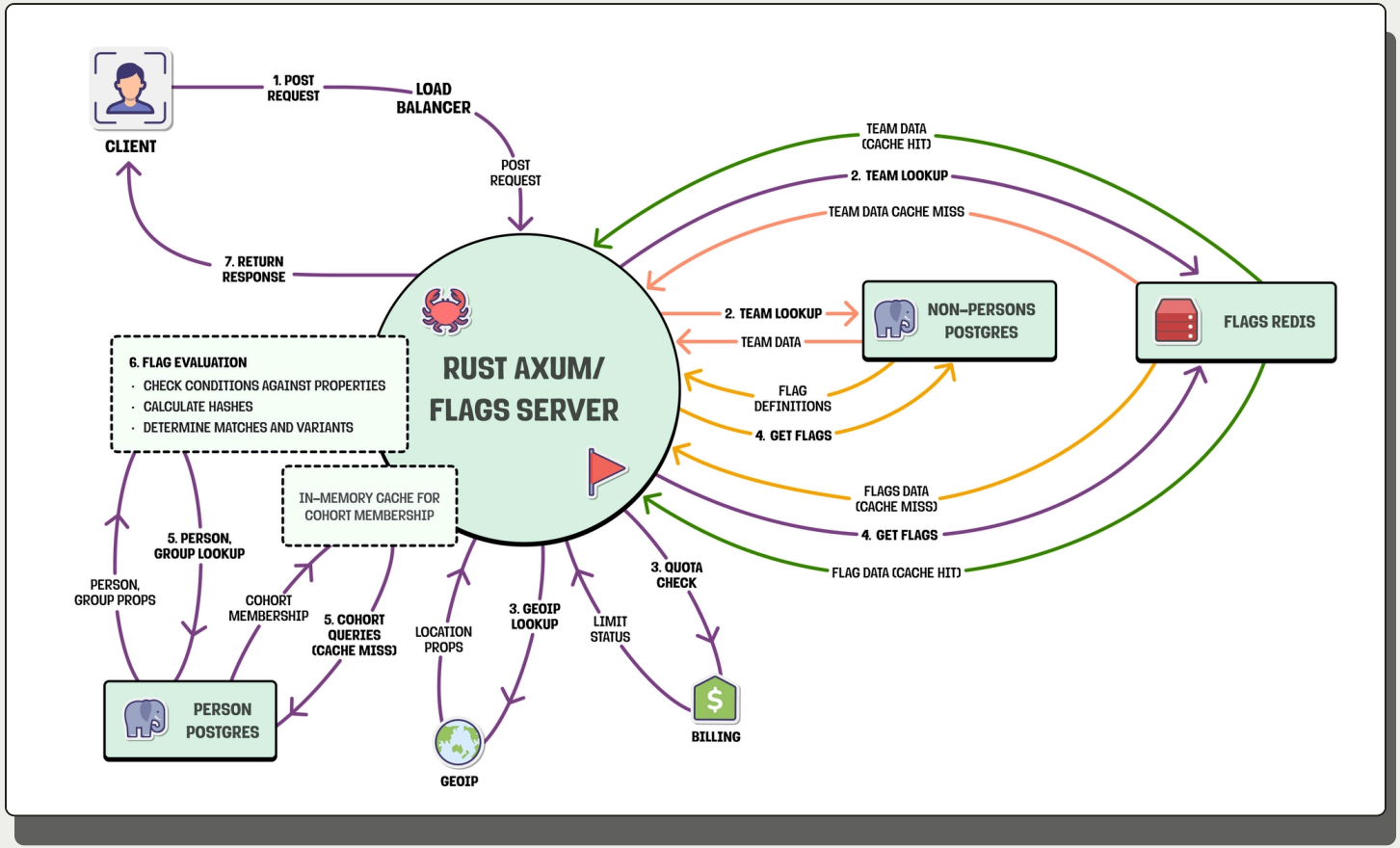
Moved evaluation into memory
one query for person properties, then evaluate every condition in parallel

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

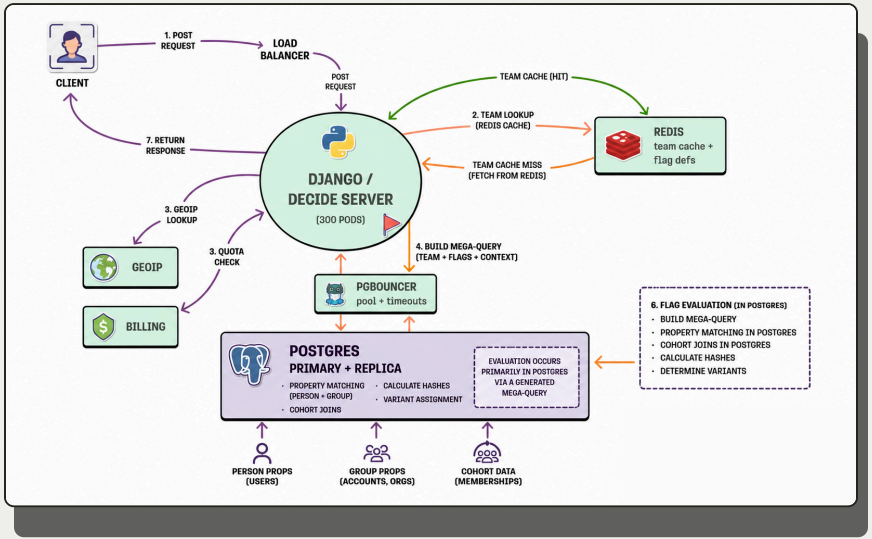


The new architecture



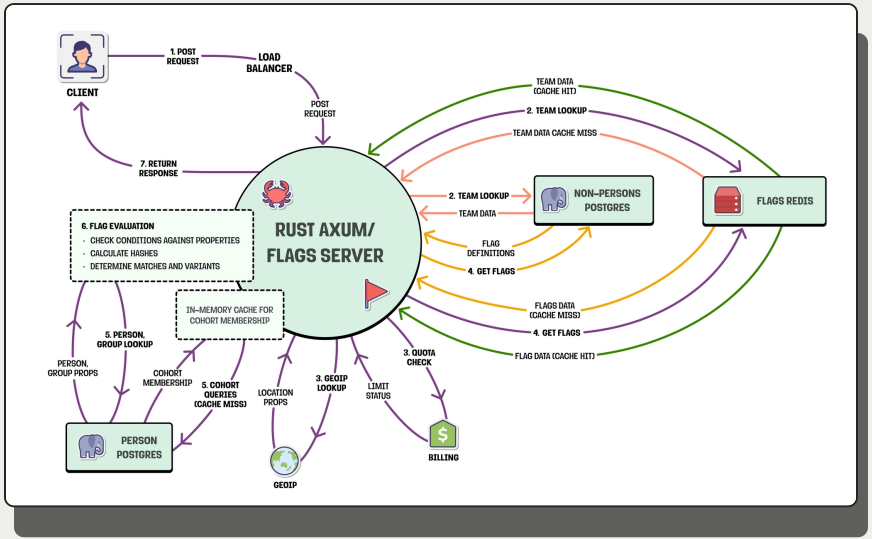
Before and after

Django



- Postgres evaluates every flag
- cohort joins on every request
- PgBouncer owns timeouts

Rust



- the service evaluates flags in memory
- cohort membership is cached
- the service owns pooling and timeouts

Latency results

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**

Cost and reliability results

300 → 90

pods for the same traffic

−68%

compute cost: $8.8k → $2.8k/mo

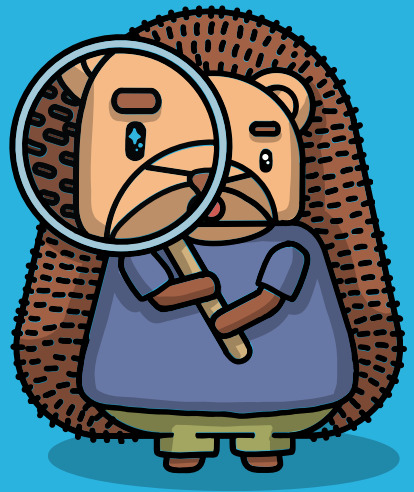
0

flag outages in the next 3 months*

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

And then things went sideways.

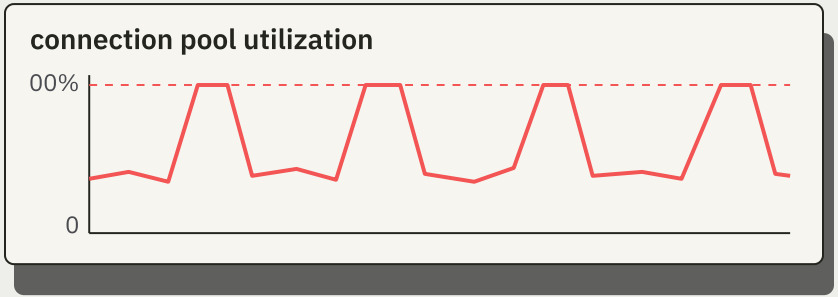
Traffic doubled. p99 spiked to 2.5 seconds, and the app froze at random times.



First hypothesis: the database

The symptoms

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



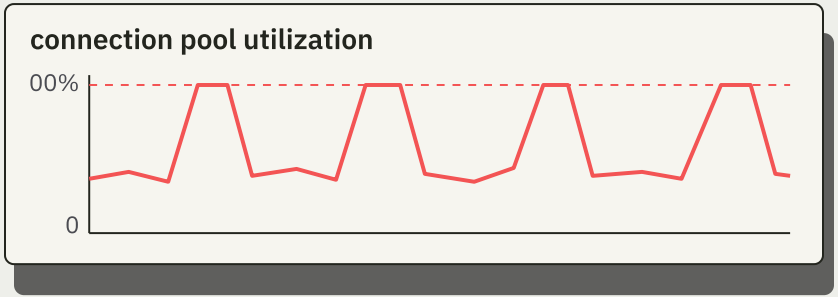
First hypothesis: the database

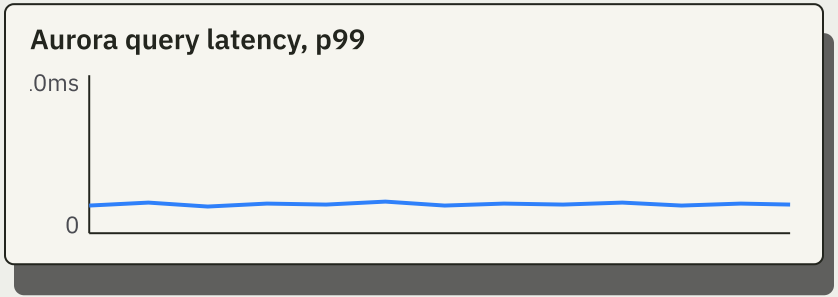
The symptoms

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.





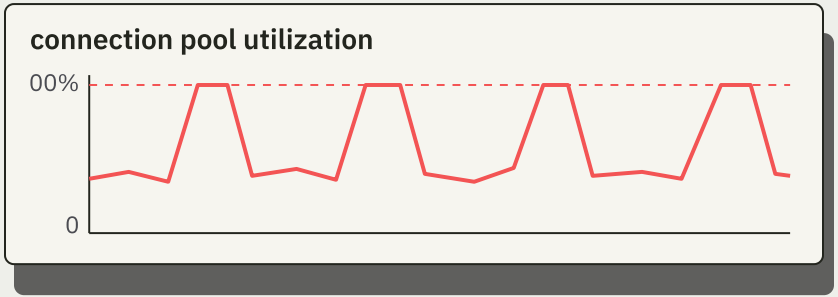
First hypothesis: the database

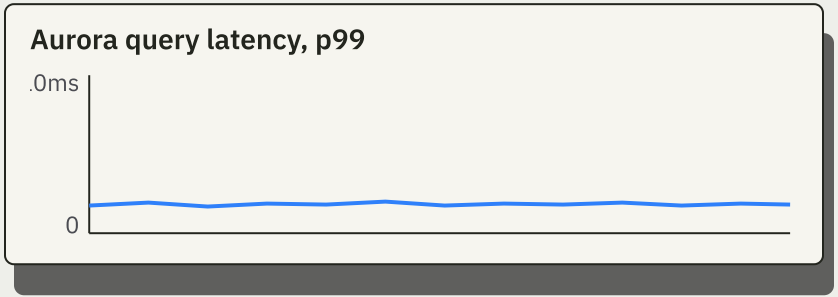
The symptoms

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.



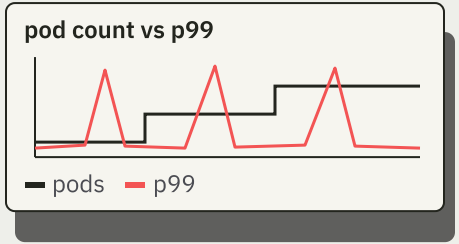


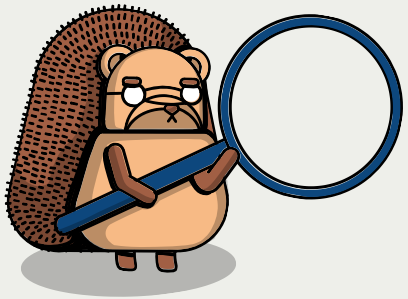
Something was making the database *appear* slow

The clue: CPU throttling

More pods did not help

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

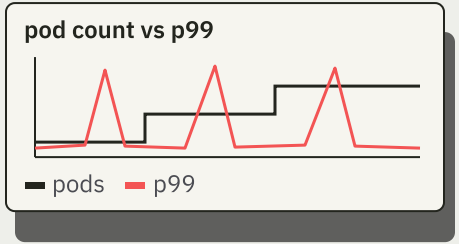




The clue: CPU throttling

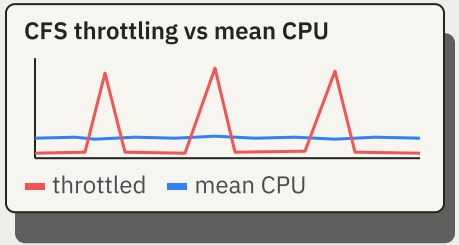
More pods did not help

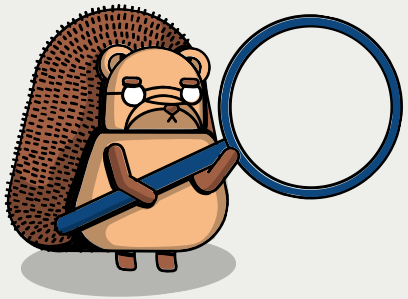
pod count did not change latency at all. The problem was inside every pod.



CFS throttling spiked with p99

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

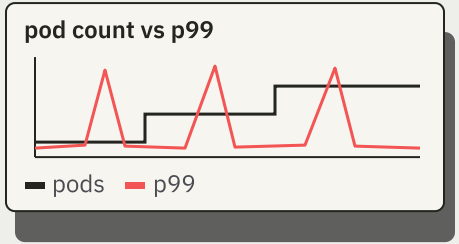




The clue: CPU throttling

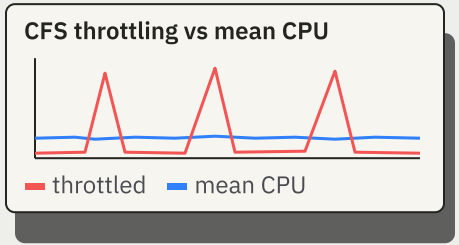
More pods did not help

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



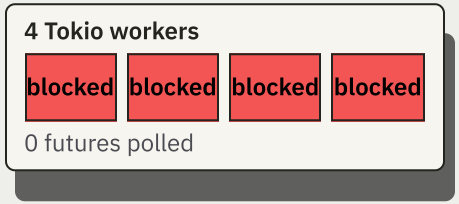
CFS throttling spiked with p99

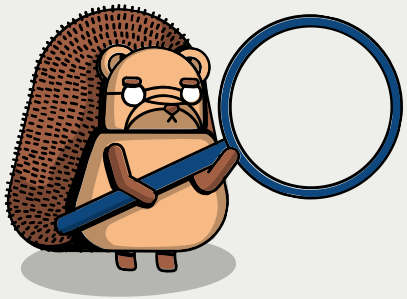
mean CPU stayed low. Short, violent bursts of CPU demand, not sustained load.



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

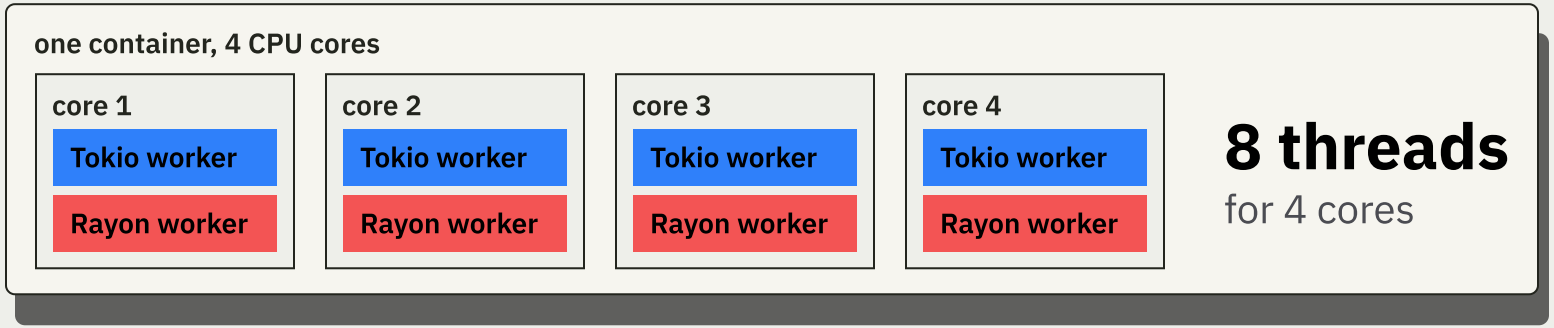
waits for many things at once

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

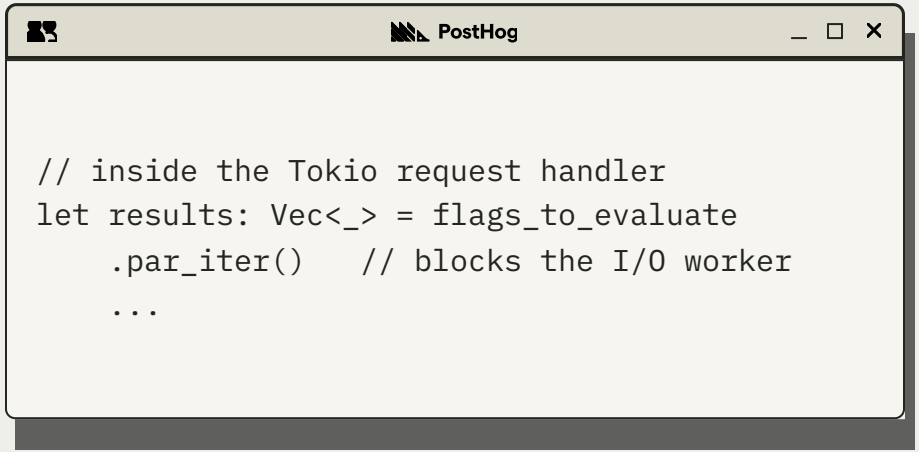
Rayon

does many things at once

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



The culprit



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

PostHog

```
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
```

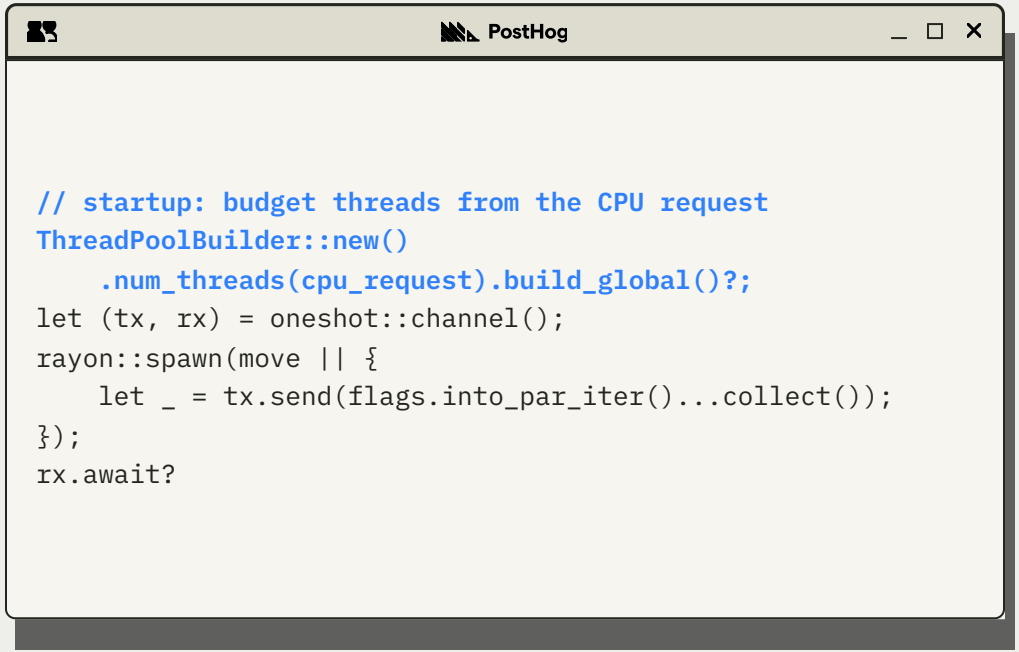
rayon::spawn

moves the work off the I/O worker and onto the Rayon pool

rx.await

yields, so the worker keeps polling other futures

Budget the threads



Budget the threads

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

Skip Rayon for small requests

PostHog

```
// startup: budget threads from the CPU request
ThreadPoolBuilder::new()
    .num_threads(cpu_request).build_global()?;
if flags.len() < PARALLEL_THRESHOLD { // ~200
    return eval_sequential(flags);
}
let (tx, rx) = oneshot::channel();
rayon::spawn(move || {
    let _ = tx.send(flags.into_par_iter()...collect());
});
rx.await?
```

Budget the threads

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

Do not parallelize small requests

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

Add backpressure

PostHog

```
// startup: budget threads from the CPU request
ThreadPoolBuilder::new()
    .num_threads(cpu_request).build_global()?;
if flags.len() < PARALLEL_THRESHOLD { // ~200
    return eval_sequential(flags);
}
let _permit = rayon_permits.acquire().await?;
let (tx, rx) = oneshot::channel();
rayon::spawn(move || {
    let _ = tx.send(flags.into_par_iter()...collect());
});
rx.await?
```

Budget the threads

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

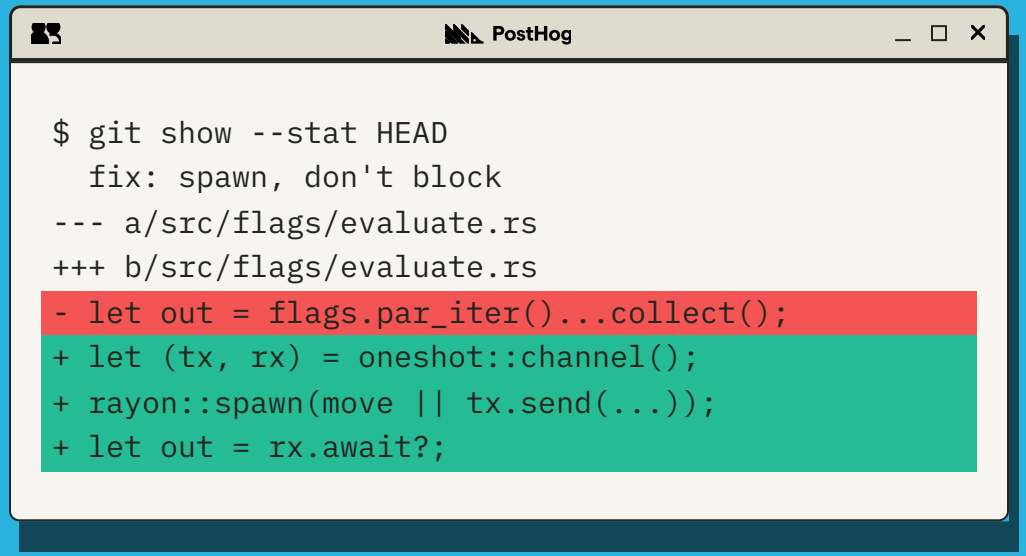
Do not parallelize small requests

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

Add backpressure

a semaphore limits Rayon to N batches at one time

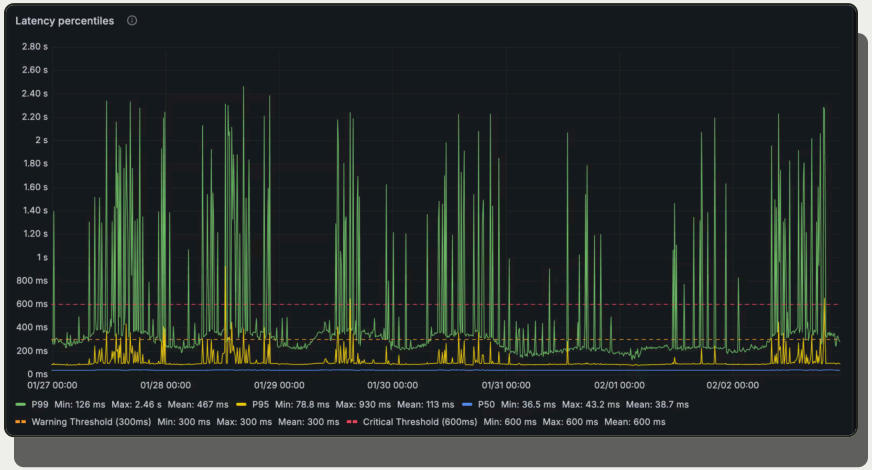
So, we shipped it.



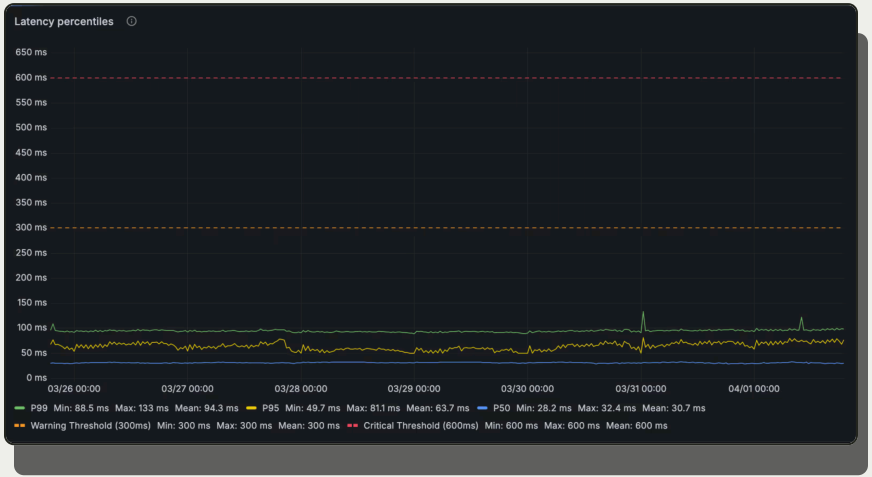
And it worked.

p99, one week apart, from production dashboards

Before: spikes to 2.5s



After: flat at ~94ms



What we learned

Do not trust the symptoms

the metrics pointed to the database, but the cause was a blocked I/O runtime



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 trust library defaults

each thread pool claimed all cores in a limited container. Set the thread counts yourself



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 trust library defaults
each thread pool claimed all cores in a limited container. Set the thread counts yourself

Do not parallelize small work
most requests got no benefit from parallel dispatch. A threshold fixed this



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 trust library defaults
each thread pool claimed all cores in a limited container. Set the thread counts yourself

Do not parallelize small work
most requests got no benefit from parallel dispatch. A threshold fixed this

The best optimization is doing less
fewer queries, fewer dependencies, fewer failure modes



Could agents have done this?



What we could hand off

Specifiable

Port evaluation into Rust
move flag logic out of SQL and into memory

Write the dispatcher
rayon::spawn with a oneshot channel and a semaphore

Add the guards
thread budget, size threshold, and semaphore

Judgment

Choose the architecture
Rust over Node, and delete PgBouncer

Distrust the obvious suspect
every metric said database, and it was wrong

Read the real system
connect CFS throttling to a blocked runtime



Agents write the code. People close the loop.

(for now. check back after the next model release)



Thank you

Connect with me

dylan@posthog.com
dylanamartin.com
linkedin.com/in/dmarticus
@dmarticus on all socials



Further reading

posthog.com/blog/even-faster-more-reliable-flags
posthog.com/blog/untangling-rayon-and-tokio
ryhl.io/blog/async-what-is-blocking