

ObservabilityCON

Always-on profiling across 6,000+ services — and the GenAI that reads it

Chris Zhang

Senior Software Engineer · Uber

Ask me about **Food**



What you'll leave with

1 A sampling bug you probably have

Random 1% sampling misses most services. More sampling won't fix it.

2 How to profile 6,000 services nobody opted into

The hard part was never the profiler.

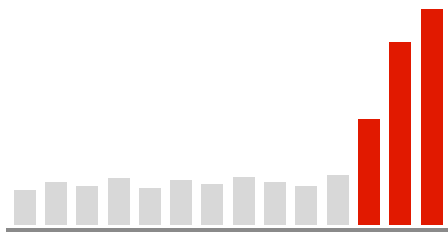
3 Why an AI can only fix what it can see

Agents need complete data to act on profiles.

Plus the two times we got it wrong, which are the more useful half.

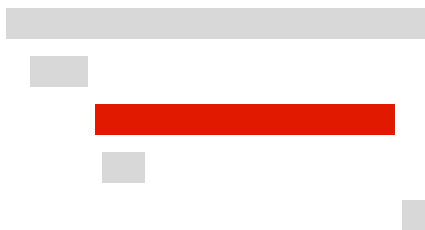
Your service is slow. Now what?

METRICS



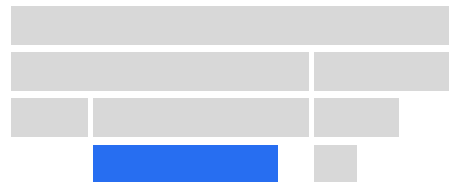
Something is slow.

TRACES



It's in this service.

PROFILES

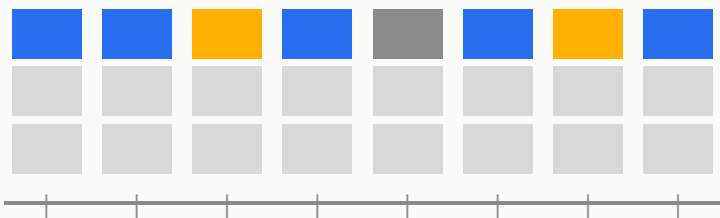


`encodeResponse()`
`handler.go:142`

It's this function. This line.

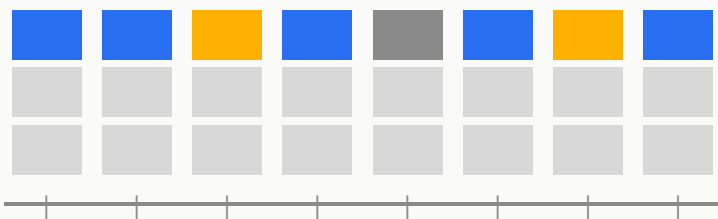
Metrics say something is wrong. Traces say where. A profile says why.

A profile is a survey of your own program



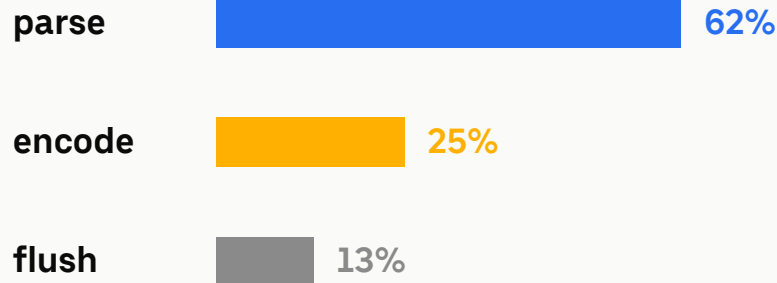
Many times a second, ask every thread:
what function are you in, and who called you?

A profile is a survey of your own program



Many times a second, ask every thread:
what function are you in, and who called you?

ADD UP THE SAMPLES



That ranking is a profile.
Same idea for memory, locks, blocked time.

“Profile it when it looks slow”

- ✗ By the time someone looks, the moment is gone.
- ✗ Ask 6,000 teams to set it up, and most never will.

So it has to be always-on, and it has to be infrastructure.

Credit where it's due

Storage, query, symbols, cost at scale.

Solved by Grafana Pyroscope — not by us.

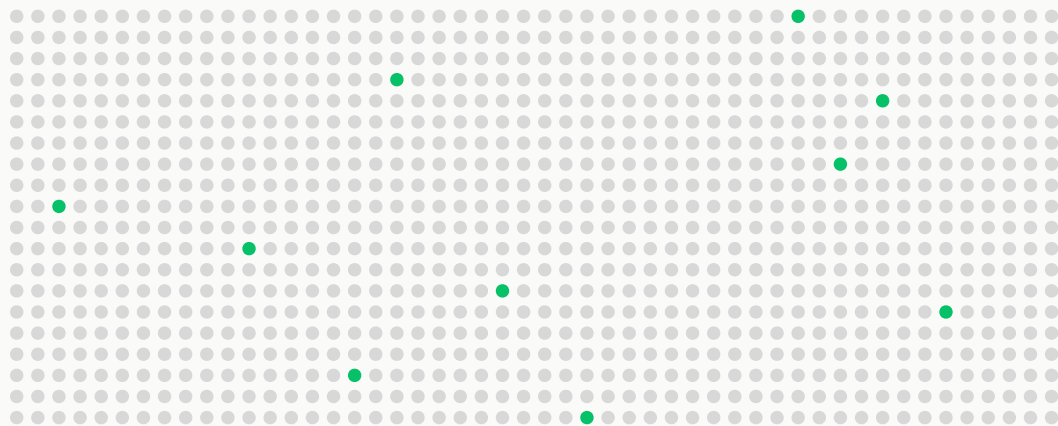
But a backend can't tell you who to profile.

It has no idea which of our 6,000 services exist right now.

Act 1

Who gets profiled?

The obvious answer: sample 1%



Too expensive to profile all of them. So sample 1%.

1,000 instances: hash the ID, keep 1 in 100. About ten land.

That works — because a thousand is a big number.

When people hear “Uber scale”

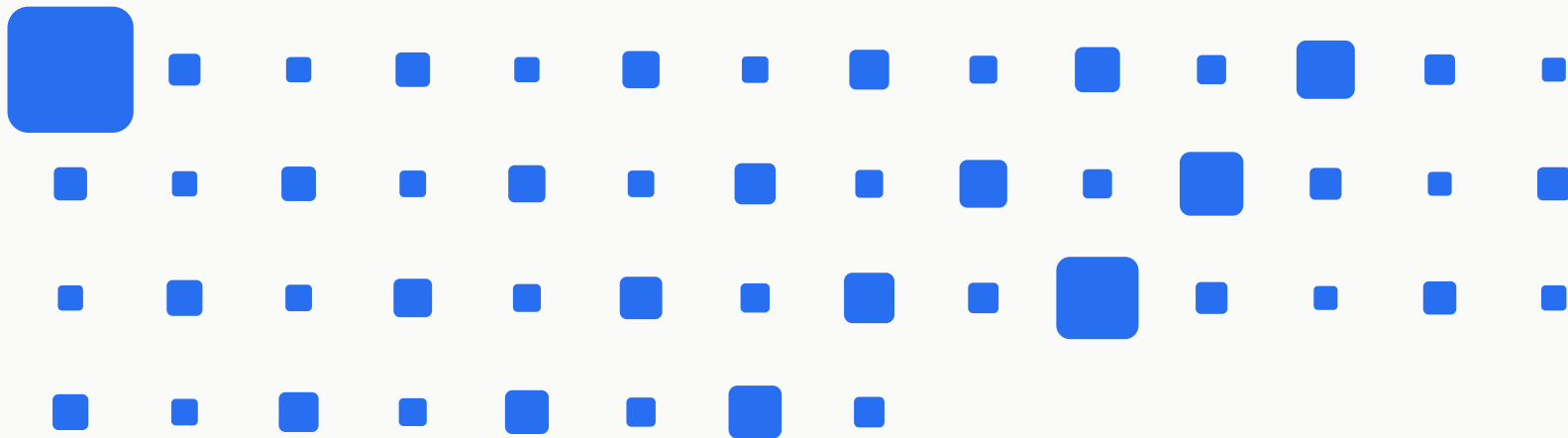
one giant system

One tap. A lot of services.



If any one of them is slow, the person holding the phone is the one who waits.

It's 6,000+ small ones



Go · Java · Python · TypeScript

stateless services, stateful stores, stream processors

It actually looks like this



17

instances in the median service

76% have fewer than 50 · 100-service snapshot, 11,710 instances, 13 zones

Flip a 1% coin 17 times

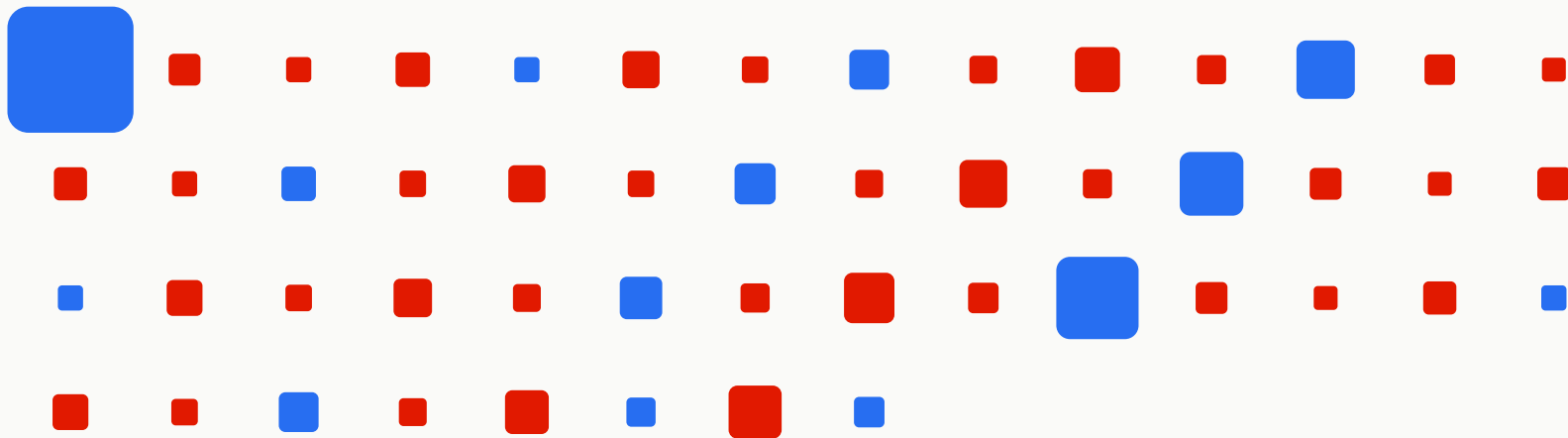


Nobody.

About 84% of the time, that service gets no profile at all.

$$0.99^{17} = 0.84$$

We ran it across 100 real services



71

of 100 services produced zero CPU samples

82 were more likely to be missed than caught

“So just profile the big services.”

✗ **Small services sit on critical paths.**

A 17-instance service in front of checkout takes everything with it.

✗ **Instance count is not importance.**

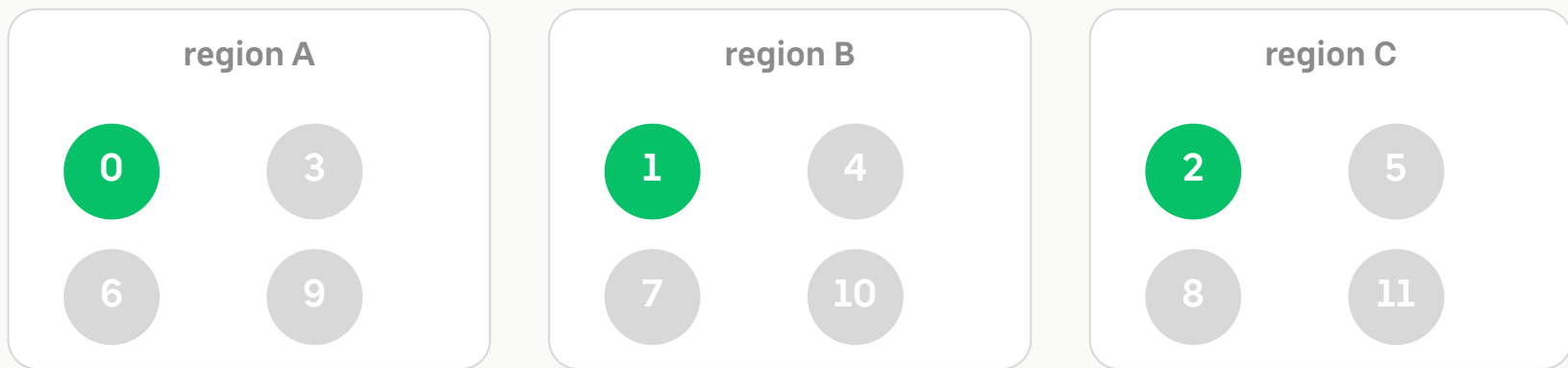
Cheap to run, expensive to lose.

✗ **It is a decision to be blind.**

76% of our services have fewer than 50 instances.

The tail is not the unimportant part. It is most of the company.

Stop flipping coins. Assign positions.



Number every instance across regions. Turn on the first few.

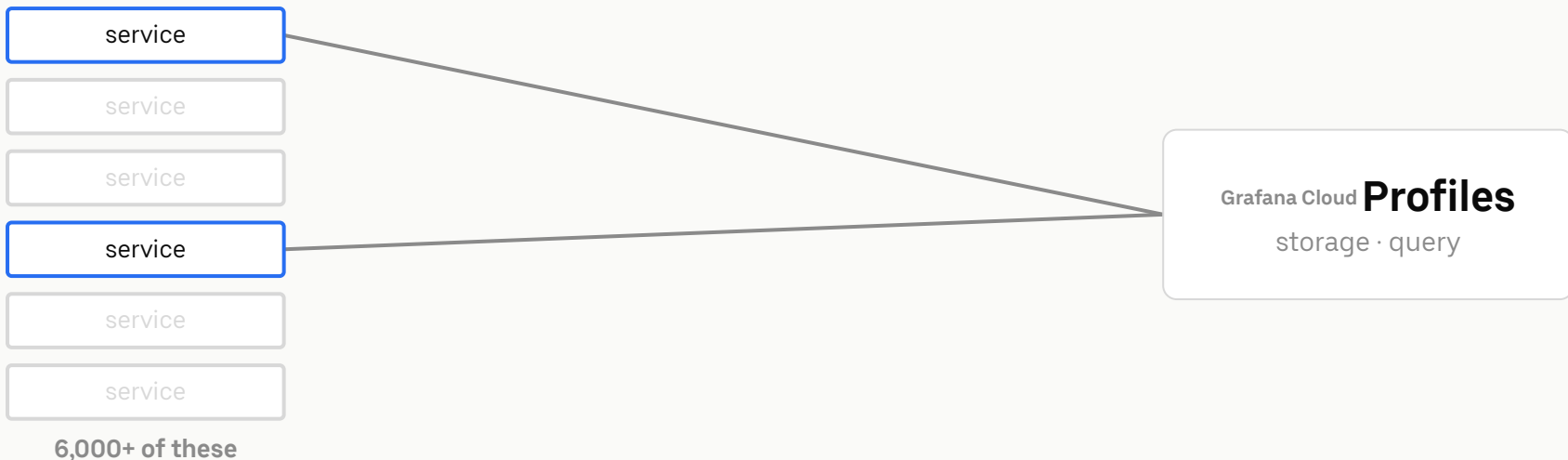
Exact budget. Reproducible. It can never skip a service.

illustrative fraction shown; the production default is 1%

Same 100 services, same 1% budget

	Coin flip	Placement
Services with zero CPU samples	~71	0
More likely missed than caught	82	0
Mean non-empty-zone coverage	7%	24%

Before: every service on its own



✗ **Each flips its own coin**

71 of 100 got nothing

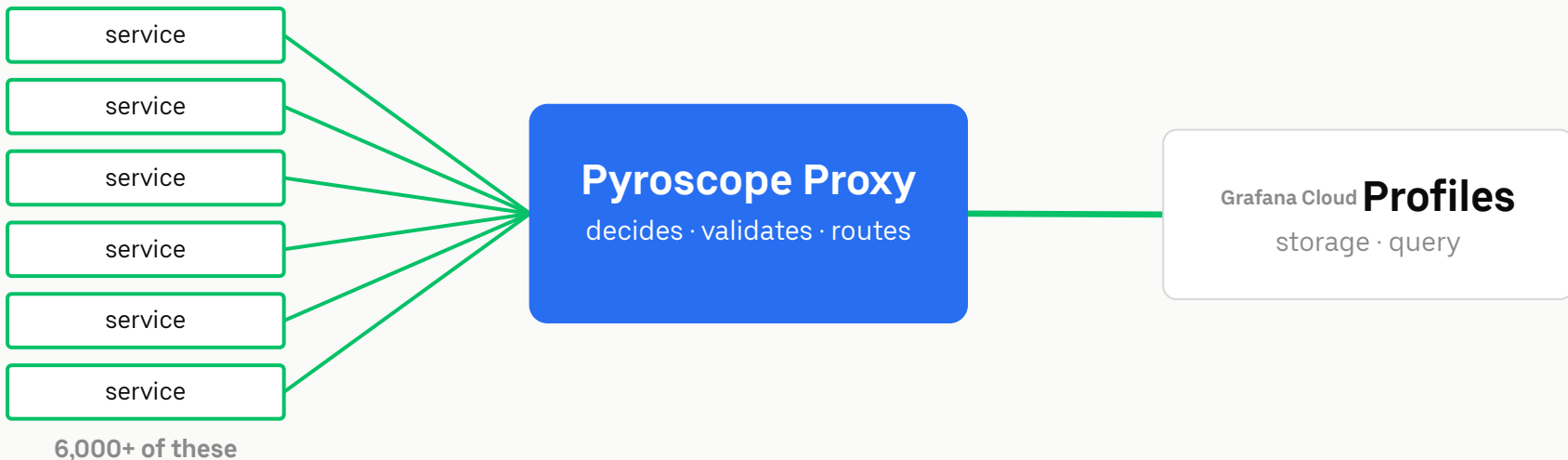
✗ **Each holds credentials**

keys copied into every app

✗ **No central switch**

a bad type means 6,000 repos

After: one place decides



✓ **One place decides**
by position — nobody missed

✓ **One place holds the keys**
uploads validated and routed

✓ **One switch per profile type**
off everywhere, in minutes

Profiling must never become an availability risk.

Can't reach the proxy? Stop collecting.

Can't upload in time? Drop the profile.

Never block the application. Non-negotiable, on both sides.

We turned on mutex profiling for Cadence. **Every instance crashed.**

The cause: an outdated dependency's hand-written assembly, deep in the tree.

**One switch: off everywhere, in minutes —
then back on once the dependency was upgraded.**

Always-on only works if you can turn it off.

In production today

630M

enablement checks
per day

55M

upload requests
per day

99.997%

of checks resolve
in under 10 ms

<1%

CPU overhead on a
profiled instance

Default-on across all 6,000+ services, all four languages, every workload type.

Act 2

What always-on actually buys you

latency · cost · incidents · OOMs · regressions · other pipelines

The smallest fix we ever shipped

FlowEval · one of our largest stateless CPU consumers

```
for _, step := range evaluationSteps {  
-   id := obj.UUID.String()  
+   id := obj.cachedID  
    ...  
}
```

That one line

1.6%

of the service's total CPU

148× slower than reading the field

- ✗ No alert fired — below any threshold
- ✗ Invisible to request tracing
- ✗ No incident. Nobody was looking.

~530 cores. From one line of code.

“Too small to alert on” and “too small to matter” are not the same thing.

Nobody went looking for this.
The profile was already running.

That is the entire argument for always-on.

A bigger one, ranked by mutex profiling

55.6%

of all blocking samples

across the stateless service population

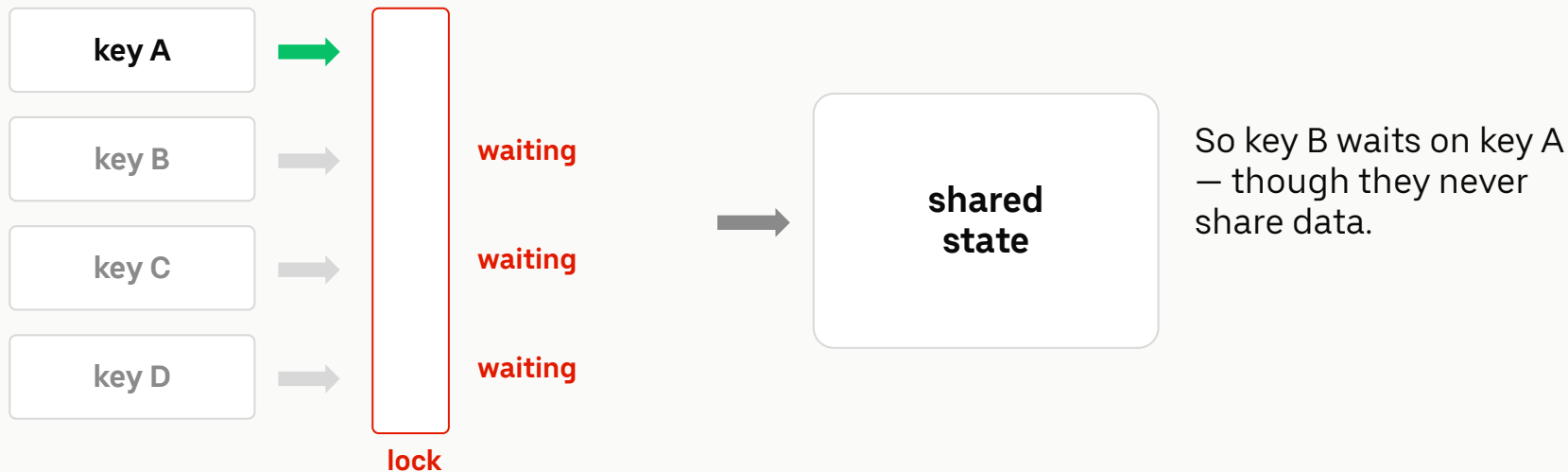
One service. One code path.

FlowEval again — a different profile type.

You only get a ranking like this if every service is in it.

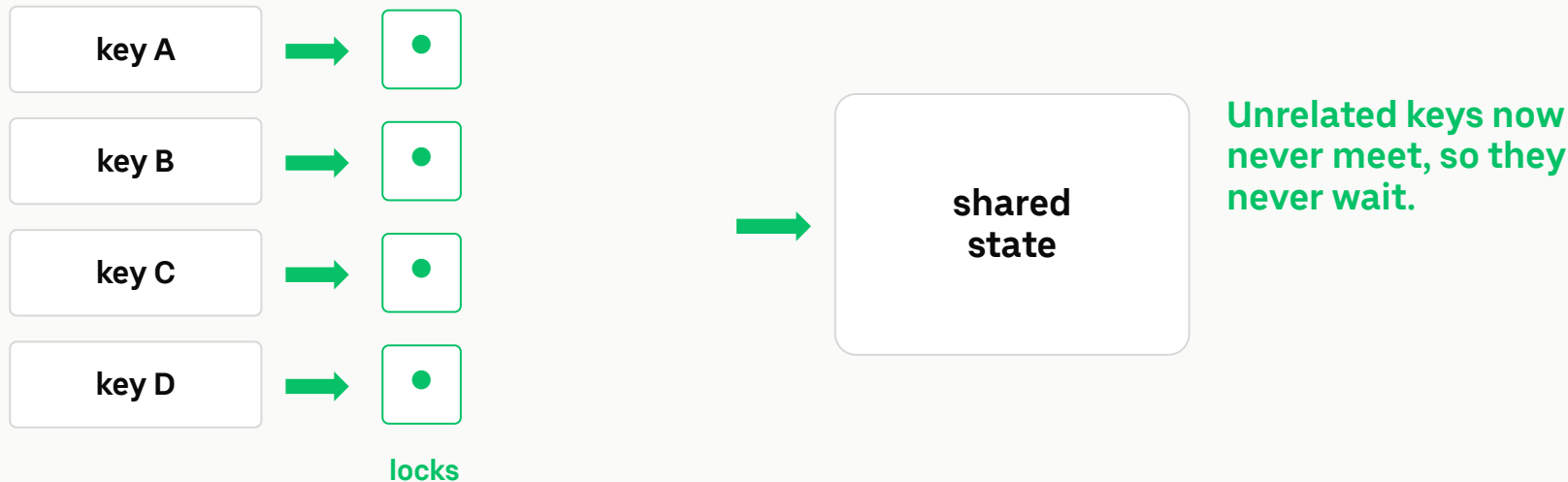
The cause: one lock for everything

A lock lets one request at a time touch shared state.



The fix: one lock per key

Give each key its own gate, created on first use.



- one lock, shared → + a lock per key

Result

85–97%

latency reduction
at 500 concurrent goroutines

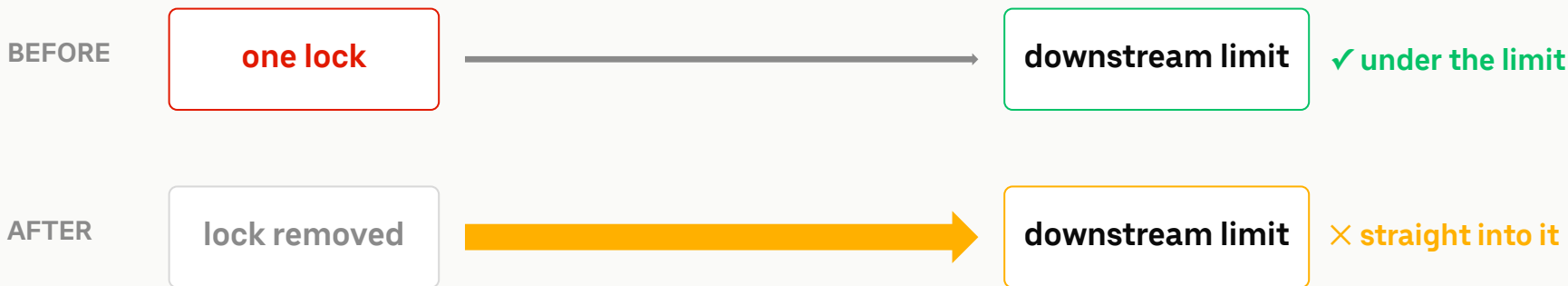
~1,200

cores recovered
(estimated)

gone

the contention class left
the ranking entirely

The fix worked. That was the problem.



Nothing inside the diff could have told you.

20 documented wins — a few of them

~70K

cores reclaimed by two services

~25K

cores from a single service

~10×

throughput
cluster scheduler

~61%

faster on the hot path
delivery pricing gateway

21%

smaller footprint, 868 → 682 cores
real-time catalog refresh

87%

of a +109% deploy regression recovered
feed ranking

~15%

of service CPU reclaimed
tax computation

Different services, different signals. The profile was already running.

Act 3

**When nobody has to read the
profile**

Three layers on the same data

ASK

Grafana Assistant

a person asks, in plain language

ANALYZE

Internal MCP integration

the same profiles, read by software

ACT

An internal agentic pipeline

an agent writes the fix and checks it

Each one only works because coverage is complete.

A Java OOM landed on my desk

🔖 Debug OOM in bullseye-backend-worker 🔗 📄

Shared by zzzhang@uber.com on 3/25/2026, 3:20:03 PM • Viewed 17 times



grafanacloud-ubercloud-profiles

we encounter some OOM for this bullseye-backend-worker service at "Mar 23, 13:09:54PDT
java.lang.OutOfMemoryError: Java heap space" can you use the memory profile to help me find out why is that, what's the root cause?

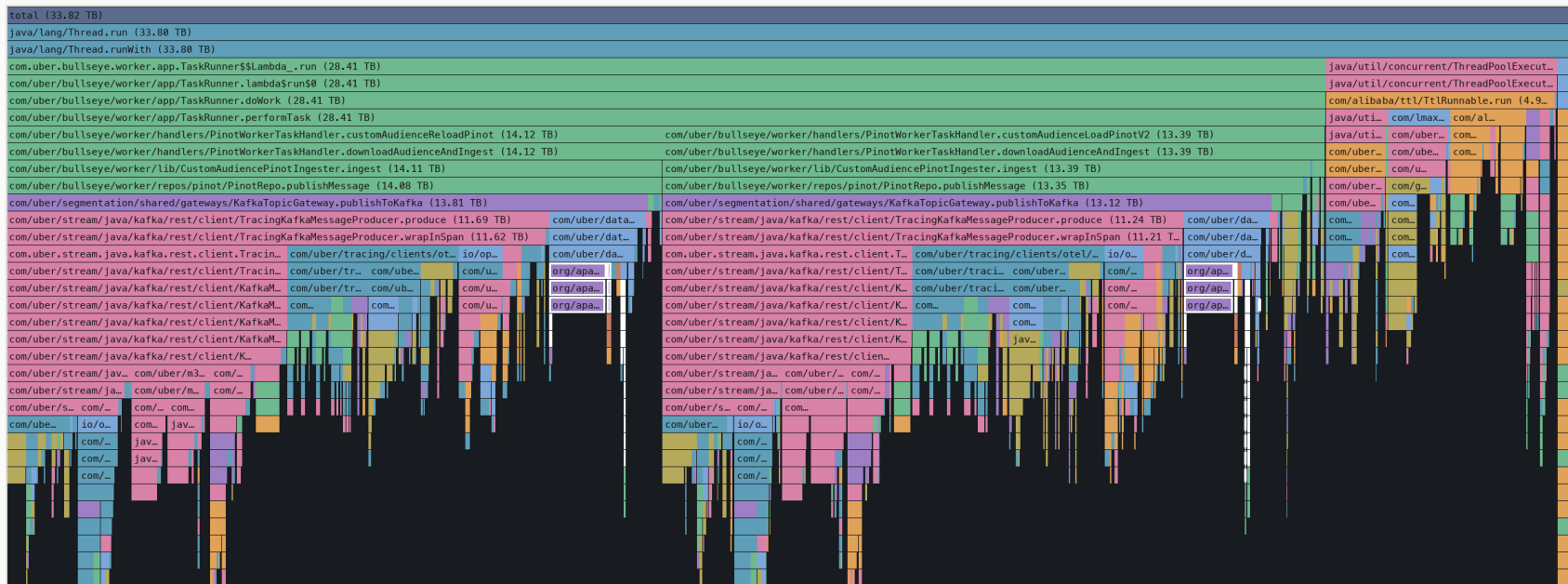
Mar 25, 2026 at 3:18 PM

**“I am not a
Java developer.”**

— me

So I typed the question the way I would say it out loud.

What it was reading



bullseye-backend-worker · recent allocation profile · outlined: the Avro path it flagged

It answered in the language of the fix

68x

allocation spike

Exact functions driving TLAB pressure — Avro, Jaeger

No Java expertise required

Immediate fixes to target:

- Replace `EncoderFactory.binaryEncoder(null, null)` with a `ThreadLocal` reusing the buffer via `binaryEncoder(stream, existingEncoder)`
- Use batch-level spans instead of per-message spans in `TracingKafkaMessageProducer`
- Replace `AbstractMessageLite.toByteArray` with a streaming serializer writing into a pooled `ByteBuffer` in `MessageBuffer.getProduceTask`
- Pre-size `HashMap` and `ArrayList` with expected capacity to eliminate resize allocations

Who reads profiles through our API?

1,435

agent reads via MCP, in one week

the biggest caller of our profile API

agents, through the MCP server

1,435

twelve engineers, through the CLI

641

an internal monitoring UI

633

CI

71

an AI context-refresh pipeline

46

an alerting pipeline

15

We built the API for engineers. Agents are now its biggest caller.

one week of Pyroscope Proxy API logs, Oct 2026 · ≈2,800 reads · Grafana's own UI is not counted here

Why you can't just point an LLM at a profile

```
yourpkg.evaluateKey
```

```
sync.(*Mutex).Unlock
```

```
runtime.unlock
```

what you actually fix

your code · zero flat time

what a model sees first

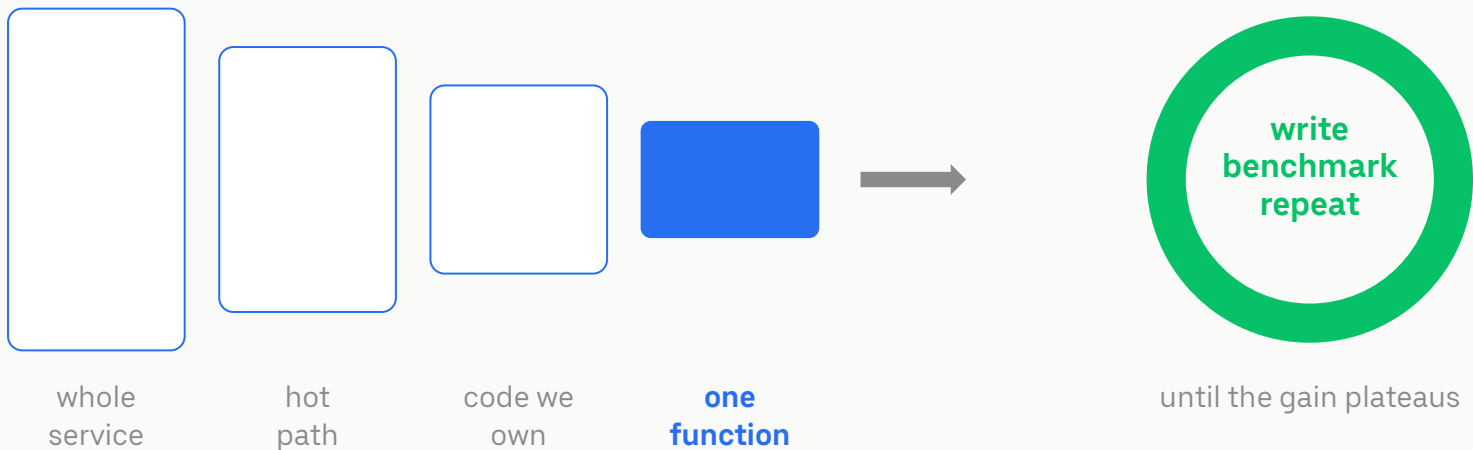
the hottest frames · standard library

Rank by hotness, and the model optimizes the Go standard library.

Pair the model with program analysis

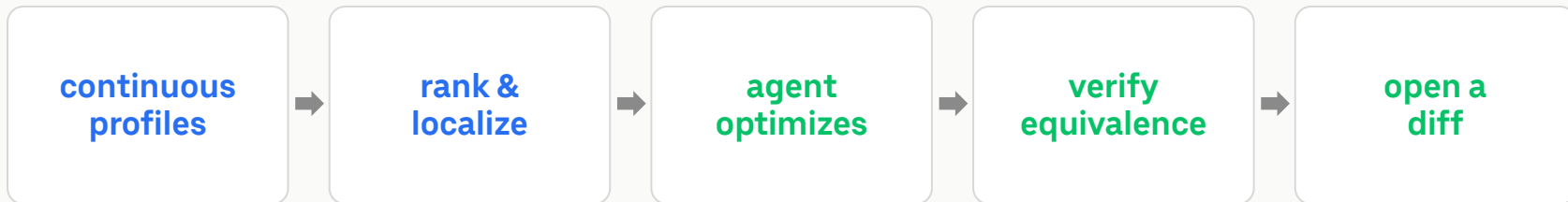
SYMBOLIC · narrows the problem

NEURAL · writes the code



Program analysis decides what to look at. The model decides what to write.

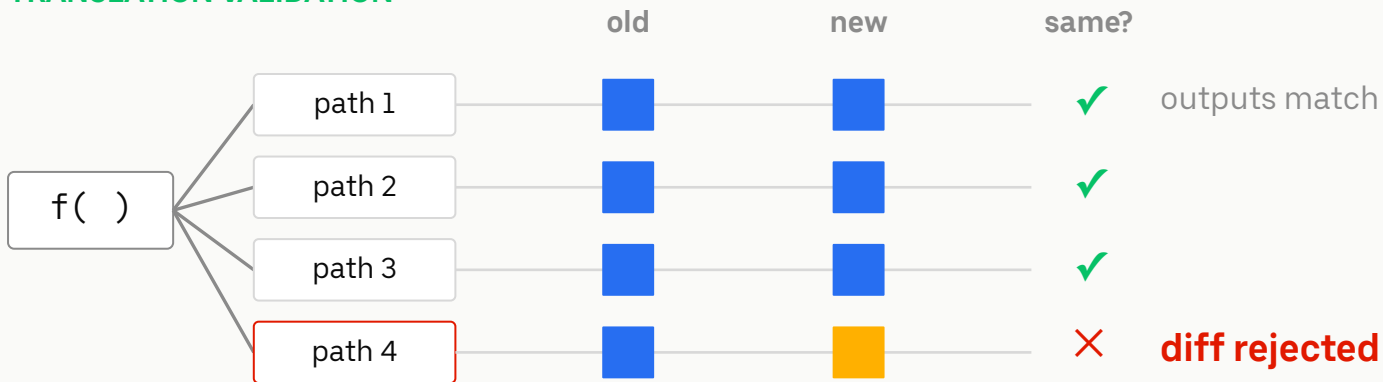
The loop, with no human trigger



Nobody triggered it. The profile did.

A faster function that returns different answers is not an optimization

TRANSLATION VALIDATION



One path disagrees, and the diff never reaches a human.

None of this works on partial data.

**An AI can only fix
what it can see.**

Two things to take home

1 An AI can only fix what it can see.

Coverage is the precondition — for people and for agents.

2 A fix can work and still break things.

Measure the fix — then keep measuring everything else.

WITH THANKS

Two teams. Two years.
About twenty-five people.

My thanks to every colleague at Uber and at Grafana who built this with me.

To most engineers at Uber, it is just a flamegraph that is already there.



Thank you

zzzhang@uber.com

CONTRIBUTIONS

Positions, not coin flips

0 of 100 services missed, same 1% budget

Profiles that agents can act on

MCP access · an agentic fix loop

A control plane in front of Pyroscope

default-on, 6,000+ services, fail-open

Open source — in progress

we are working on releasing Pyroscope Proxy