GOLAB • 2026

# Why Your Go Benchmarks Are Lying

And How to Stop Them

**Kemal Akkoyun • Datadog**



**"Not all fast software is world-class,**

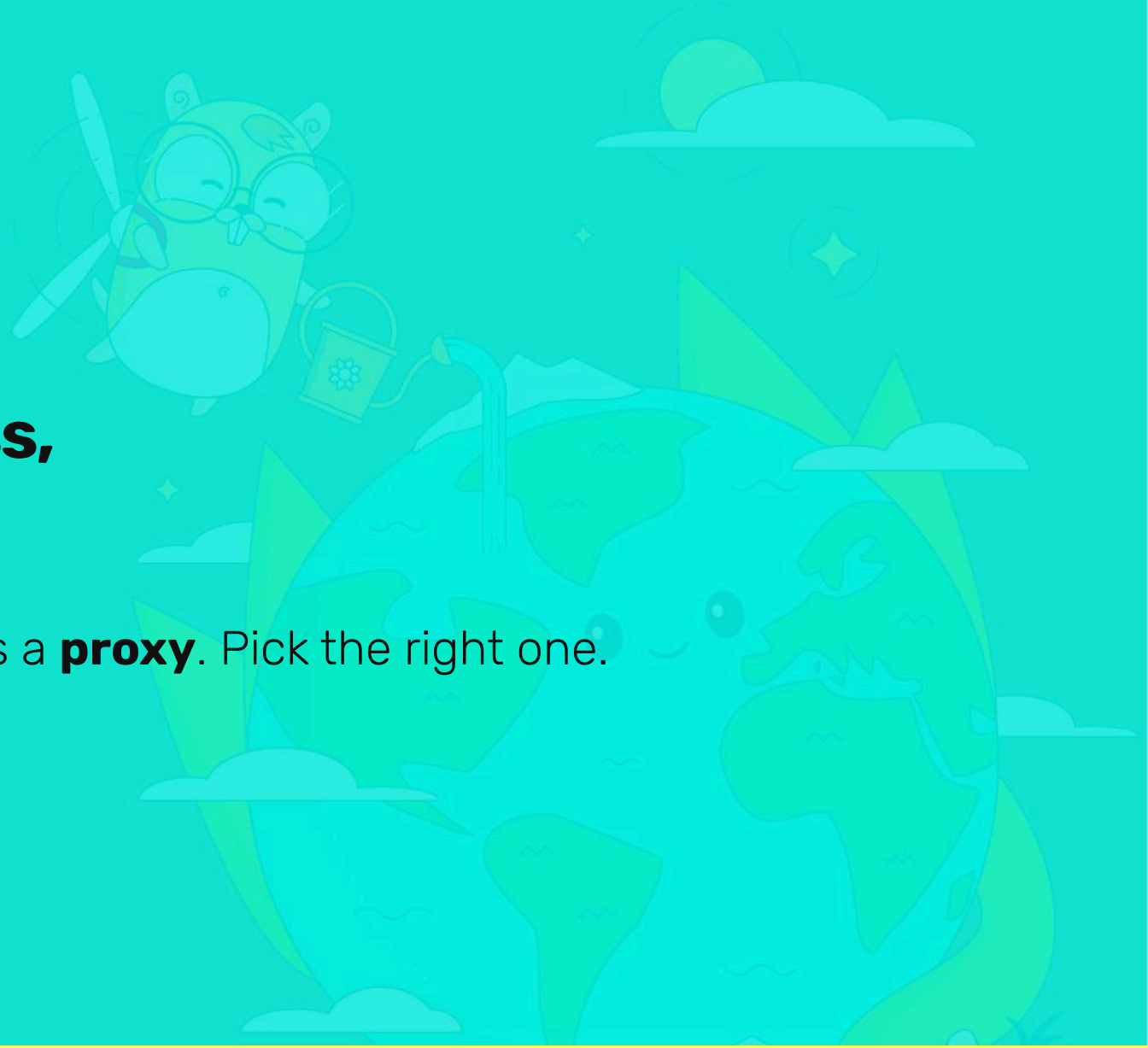
**but all world-class software is fast."**

**Performance is *the* killer feature.**

Tobi Lütke, Shopify • X, 5 May 2024

# Measure customer happiness, not CPU usage

A benchmark measures a **proxy**. Pick the right one.



# Latency and throughput

## Latency

One operation.

*Users feel this.*

## Throughput

Operations per second.

*Your ceiling.*

Improve one, damage the other.

# The cost of slowness

**0.1–0.2 s**

barely noticeable

**0.3–0.5 s**

slightly slow

**1–3 s**

you notice the wait

**5–10 s**

users leave

Half a second slower: **–20%** Google search traffic.

Nielsen, *Usability Engineering* (1993) • Google, via Coding Horror: Performance is a Feature

# Before you optimize

## Is it slow?

Profile production.

`pprof` • Continuous Profiler • OTel eBPF profiler • Parca

## Can it go faster?

Headroom is the gap to the hardware ceiling.

## Is it worth it?

SLO, error budget.

Amdahl: only the hot path pays.

Benchmark what production says is **hot**.

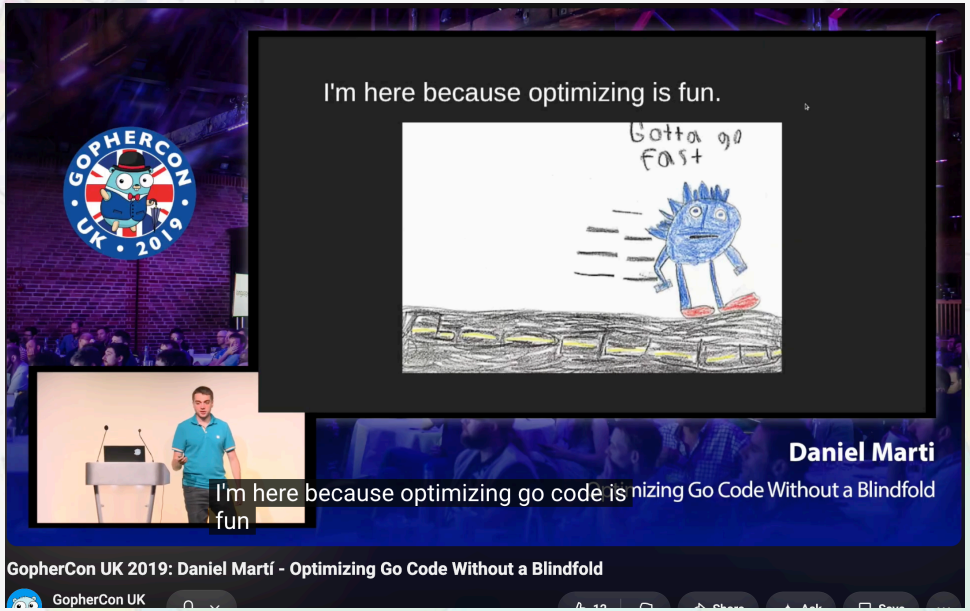
# Finding what to optimize

## Optimizing Go Code Without a Blindfold

Daniel Martí • GopherCon 2019

He covers *what* to optimize.

This talk covers *whether you can trust the number*.



# Why Datadog cares

## Our code, your process

Every ns comes out of a customer's budget.

**–3.4% CPU**

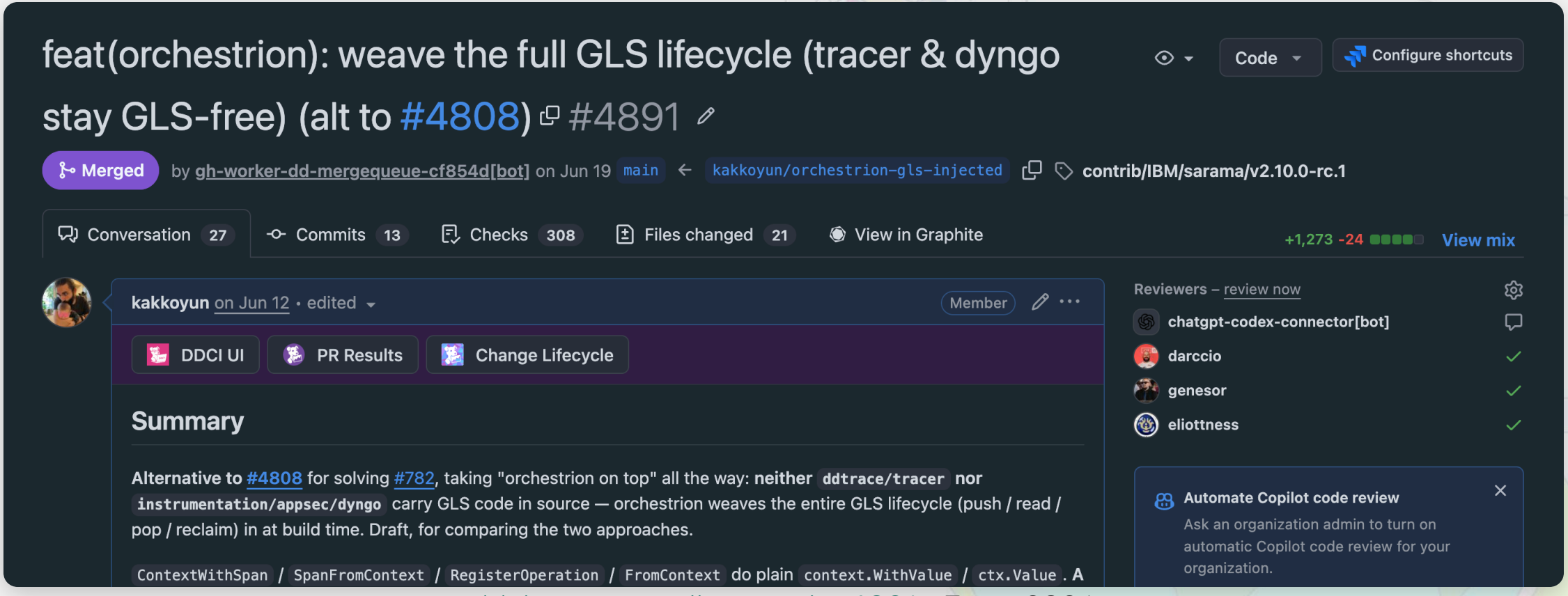
on one large service, from PGO alone.




Datadog invests in performance testing internally. This talk is the spillover.

## Overhead is product correctness

# dd-trace-go #4891




dd-trace-go pull request #4891 • June 2026

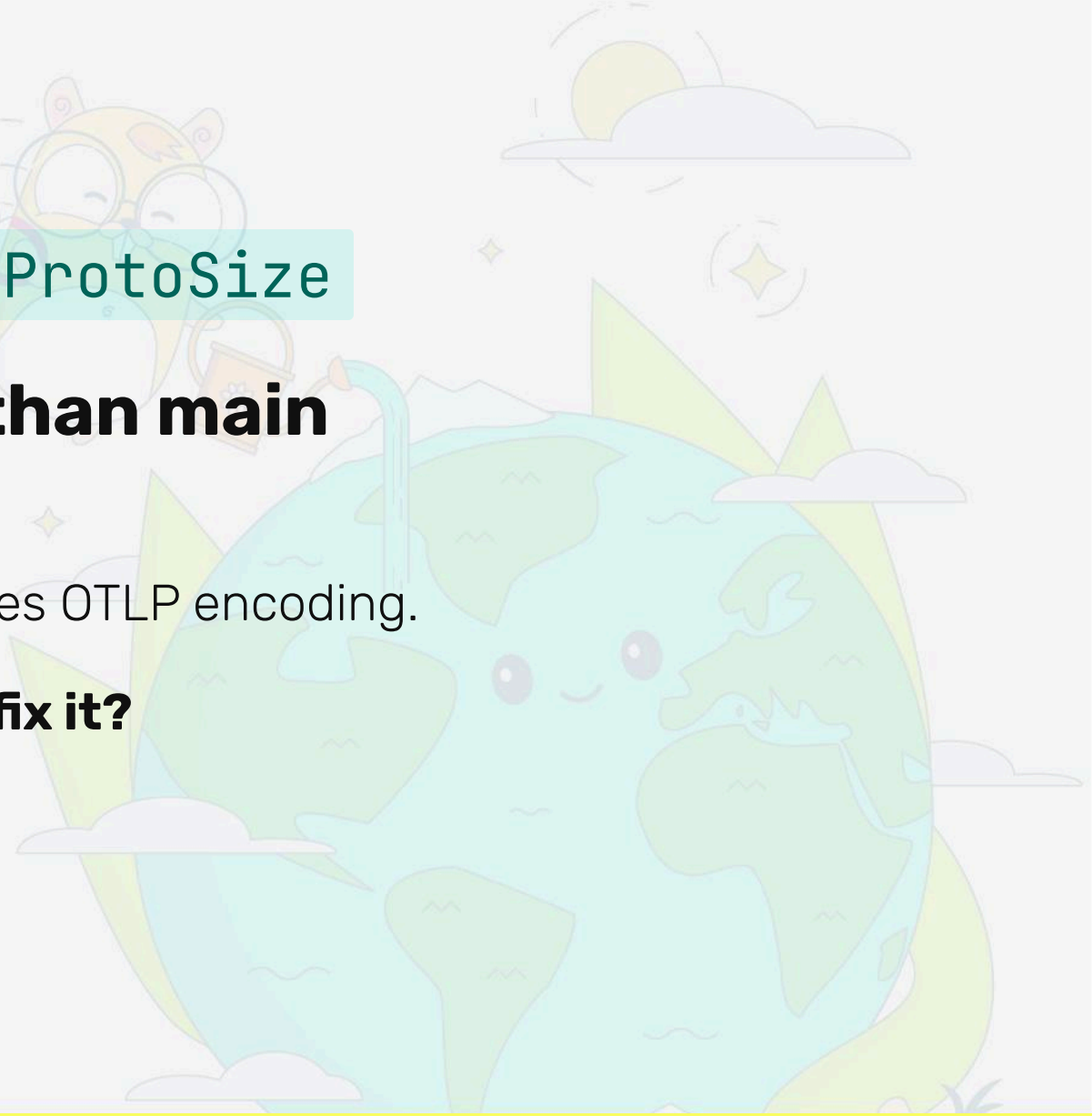
# A benchmark bot comment

`BenchmarkOTLPProtoSize`

**6-9% slower than main**

Nothing in the diff touches OTLP encoding.

**Would you fix it?**



# The last bot run

Found 4 performance improvements and 0 performance regressions! Performance is the same for 293 metrics, 2 unstable metrics, 1 flaky benchmarks without significant changes.

▸ **Explanation**

**scenario:BenchmarkOTLPProtoSize/1000spans**

- 🟩 `execution_time` [-21.477µs; -20.111µs] or [-6.341%; -5.937%]

**scenario:BenchmarkOTLPProtoSize/100spans**

- 🟩 `execution_time` [-2.205µs; -2.068µs] or [-6.502%; -6.096%]

**scenario:BenchmarkOTLPProtoSize/10spans**

- 🟩 `execution_time` [-236.612ns; -221.988ns] or [-6.830%; -6.408%]

**scenario:BenchmarkOTLPProtoSize/1span**

- 🟩 `execution_time` [-33.981ns; -32.119ns] or [-8.046%; -7.605%]

Same benchmark, final commit: **6–8% faster**.

# The benchmark that matters

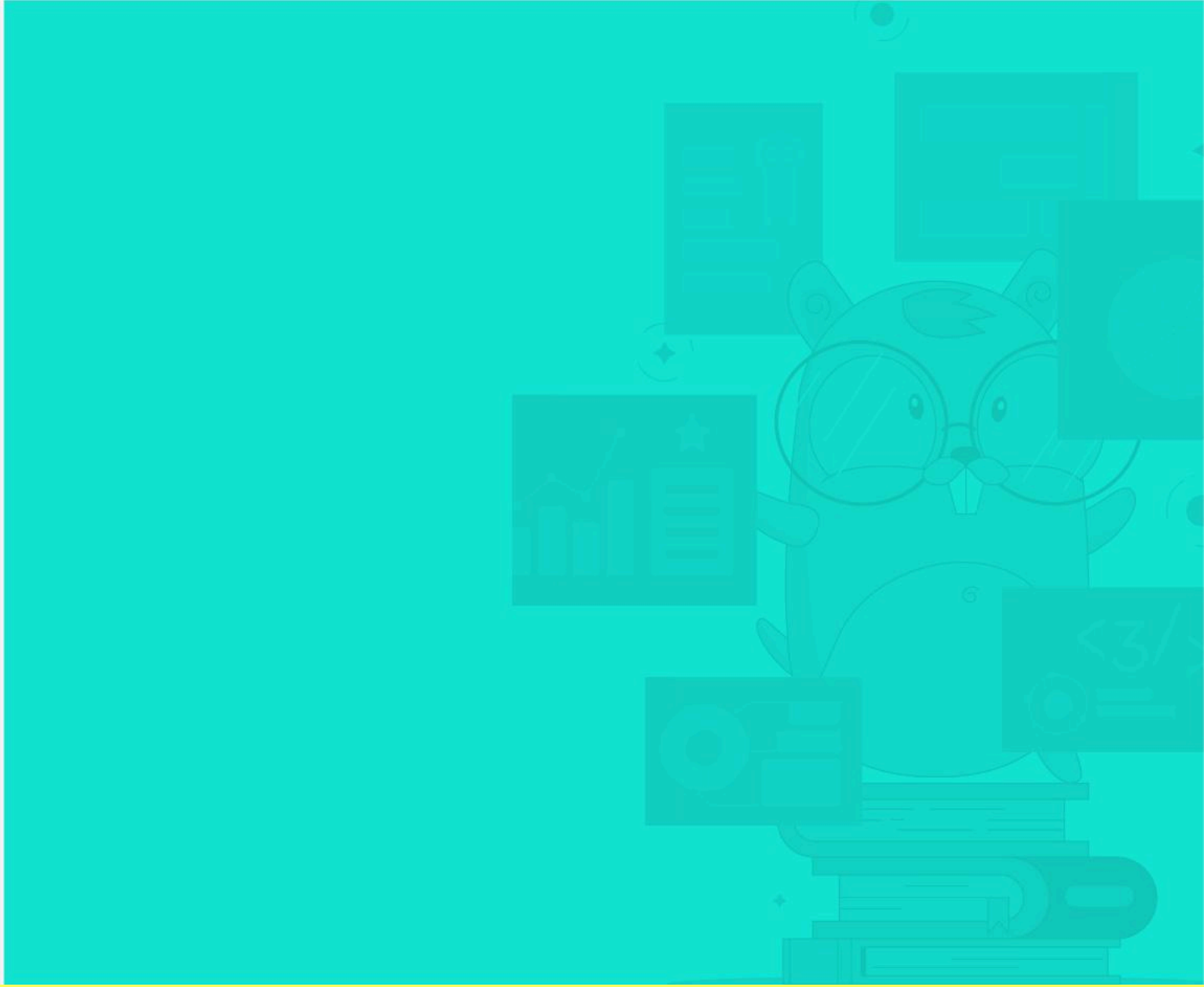
| Benchmark | main | #4891 | Δ | allocs |
|---|---|---|---|---|
| ContextStackPushPop | 236n | 252n | +6.7% | 1→1 |
| GLSPopFuncSameGoroutine | 400n | 410n | +2.4% | 4→4 |
| GLSPopFuncCrossGoroutine | 27.1µ | 27.6µ | +1.9% | 8→8 |
| ContextStackDepthScaling (Peek, depth 0-10k) | 16-65n | 16-65n | ~0% | 0→0 |

So ~16 ns added to a push/pop cycle with no new allocations, and Peek/pop unchanged. The woven `ContextWithSpan` additionally captures a goroutine-scoped popper once per span (first-wins): one closure allocation per span, which these data-structure microbenchmarks do not exercise. All deltas are single-digit-to-low-double-digit nanoseconds on operations that are a small fraction of a full StartSpan/Finish, and there is zero cost in non-orchestrion builds.

The code I changed: **~16 ns** per push and pop. No new allocations.

01

# A Loose Cable



# September 2011



nature

News | Published: 22 September 2011

## Particles break light-speed limit

Geoff Brumfiel

*Nature* (2011) | Cite this article

**2879** Accesses | **13** Citations | **854** Altmetric | Metrics

**"Trivial things we are clever enough to rule out."**

Antonio Ereditato, OPERA coordinator • *The Guardian*, 22 Sep 2011



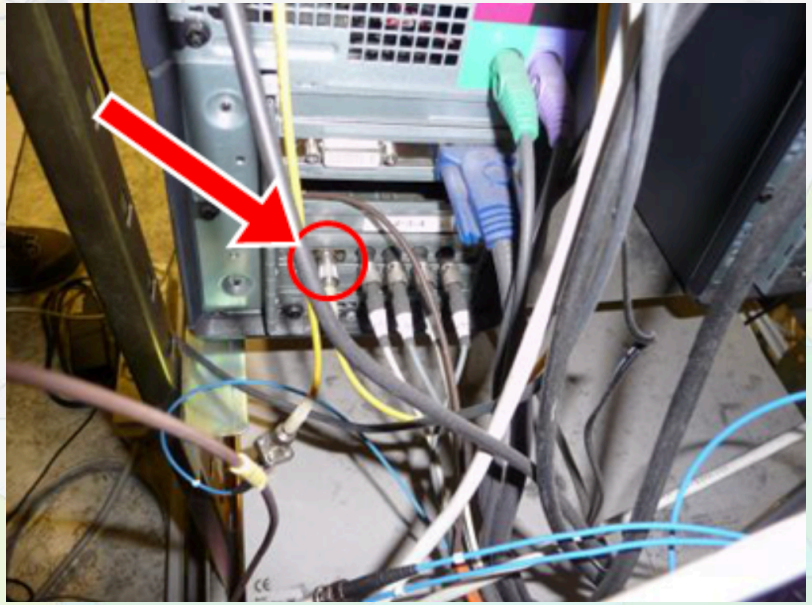
# The cause

## A loose fibre connector

**~73 ns** early bias

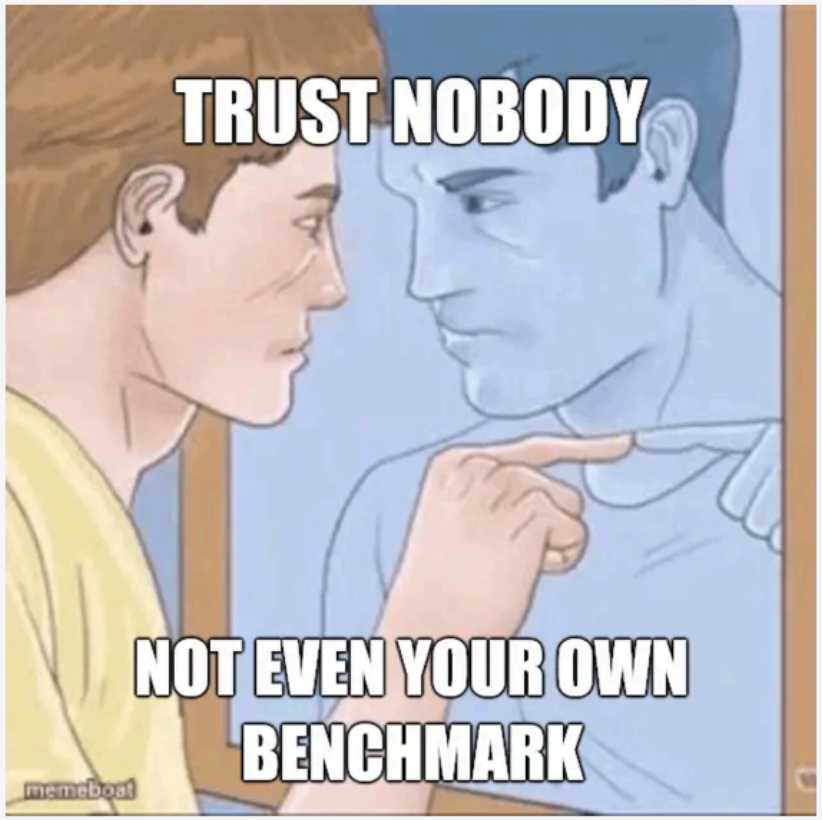
A second fault pushed back and *partly offset* it

CERN, 22 Feb 2012 • Cartlidge, *Science* 335(6072):1027



# Your cables

compiler • statistics • machine




# systematic error survives expert review

# Three questions

01 **Real work?**

02 **Stable sample?**

03 **Above the noise?**

# One command, two habits

```
$
go test -bench=. -benchmem -count=10 ./...
```

```
$ bloop_bench_test.go

BenchmarkHash_BLoop-16    6834830    356.2 ns/op    0 B/op    0 allocs/op
```

`-count=10` : one run is one draw.

`-benchmem` : always show allocations.

02

# Is It Real Work?

Question 1: real work?

# The compiler is not neutral

It sees the result is unused.

It removes the work, **correctly**.

Your loop still runs `b.N` times.

It just runs empty.

# Dead-code elimination

dce_bench_test.go

```go
func makeBuffer(n int) []byte {
    return make([]byte, n) // heap-escaping allocation
}

func BenchmarkMakeBuffer_DCE(b *testing.B) {
    for range b.N {
        makeBuffer(64) // result discarded → call removed
    }
}
```

## `make bench-dce`

```
$ dce_bench_test.go
```

```
DCE        0.2532 ns/op    0 B/op    0 allocs/op
correct   11.14   ns/op   64 B/op    1 allocs/op
```

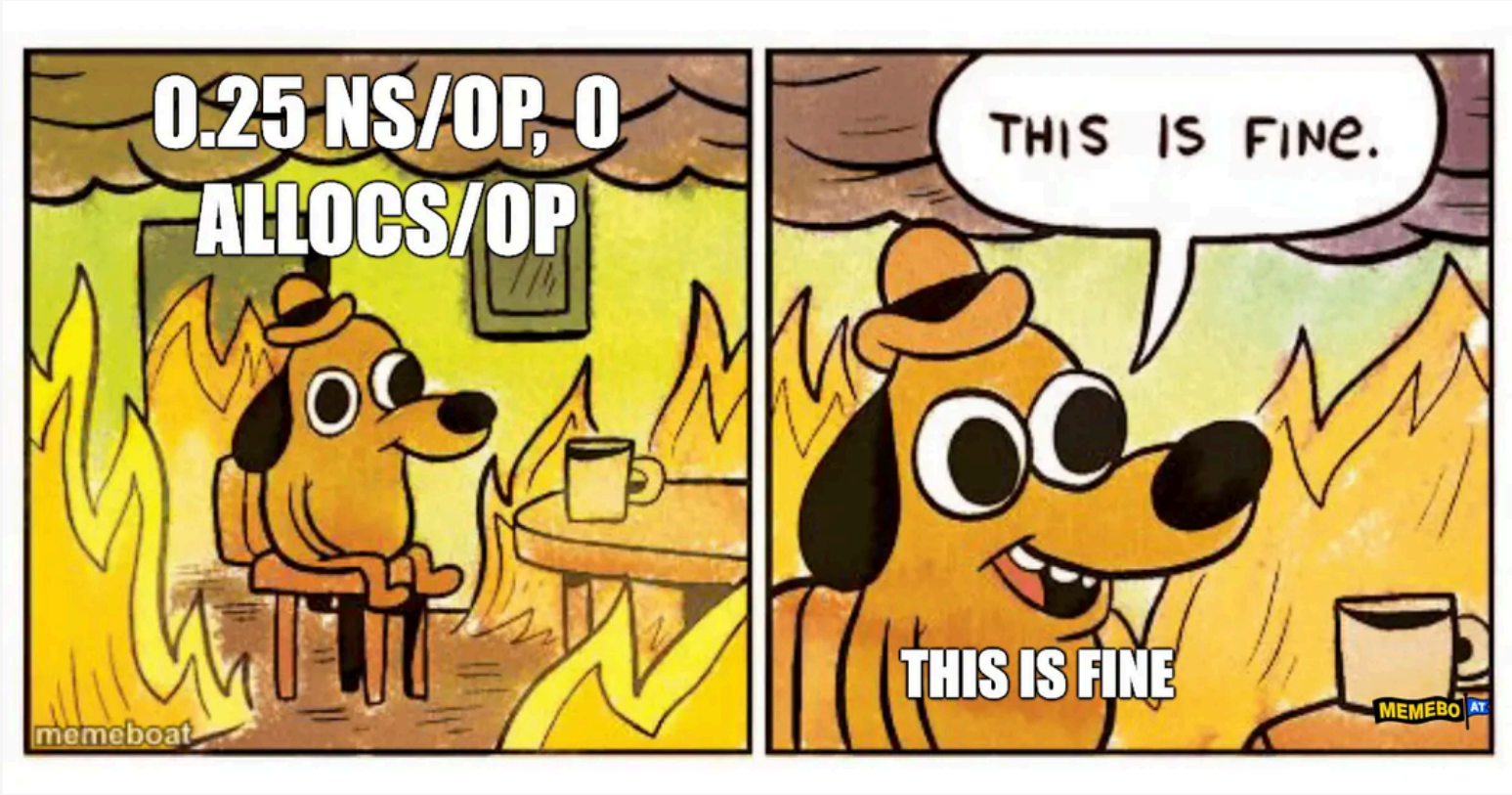
A quarter of a nanosecond. **0 allocs/op.**

The work is gone.

## `ns/op` can lie

## so check `allocs/op` too

Always `-benchmem`.




# the compiler deleted your benchmark

# `b.Loop()` • Go 1.24, fixed in 1.26

bloop_bench_test.go

```
func BenchmarkHash_BLoop(b *testing.B) {
    data := make([]byte, 1024)    // setup excluded automatically
    copy(data, payload)

    var s [32]byte
    for b.Loop() {
        s = sha256.Sum256(data)
    }
}
```

**Go 1.26** stopped blocking inlining in the loop. Upgrade first.

# What `b.Loop` removes

| | `for range b.N` | `for b.Loop()` |
|---|---|---|
| ResetTimer at start | No | **Yes** |
| StopTimer at end | No | **Yes** |
| Setup re-runs on ramp-up | Yes | **No** |
| DCE prevented | No | **Yes** |

Must be written literally as `b.Loop()` • `testing.B.Loop`

# The cache can lie too

**Same input every iteration**

The cache remembers it.

You measure the **cache**.

**Fresh input every iteration**

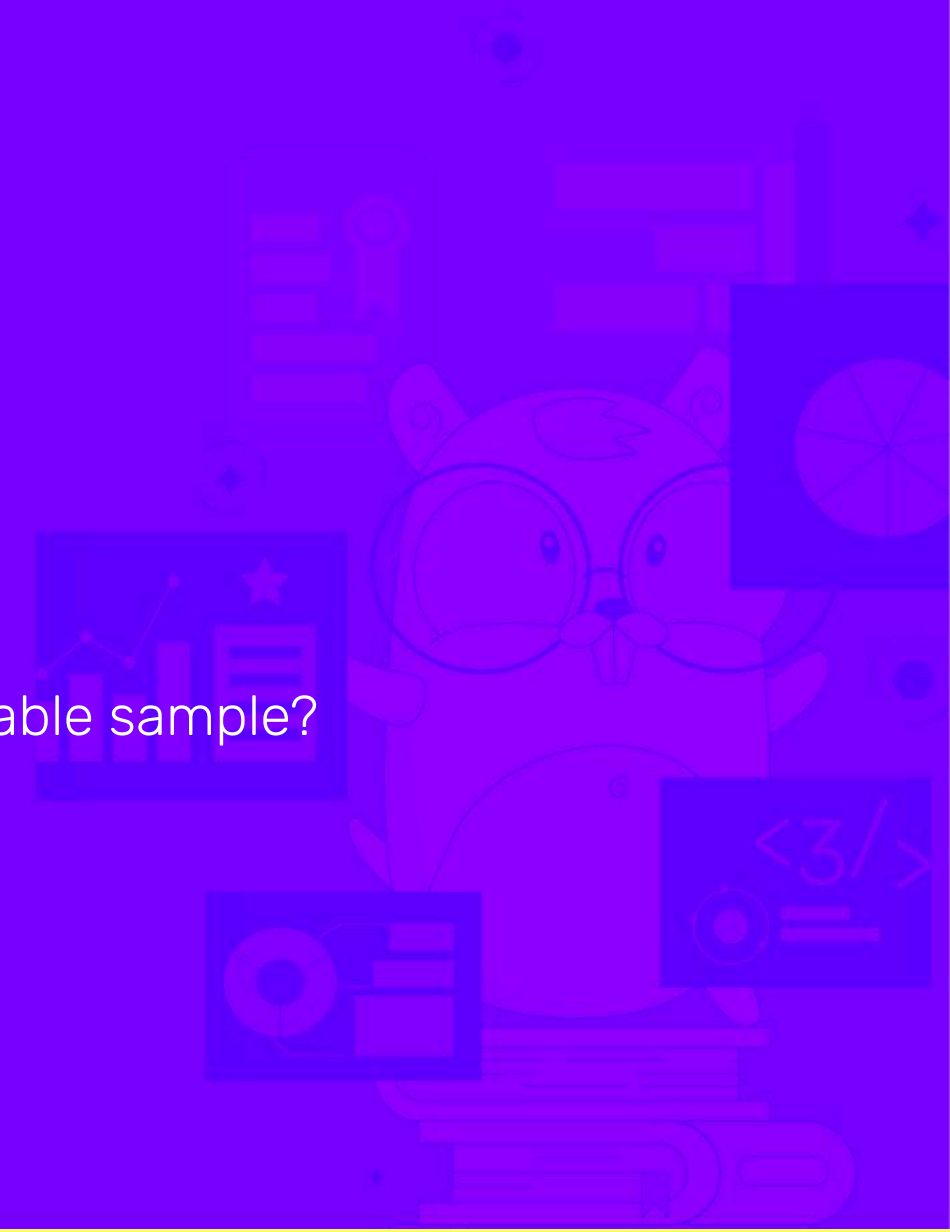
The fake 50% speedup **disappears**.

Teiva Harsanyi • How to Write Accurate Benchmarks in Go • *100 Go Mistakes* #91

03

# The Regression That Was a Speedup

Question 2: stable sample?



# Read the benchmark first

dd-trace-go

```
// The entire timed loop:
proto.Size(tracesData)
```

Struct built **before** `ResetTimer`.

Calls **none** of the code the PR touched.

Locally: **<0.1%** run-to-run variance.

# Same machine, both branches




**The PR was faster.** CI inverted the sign.

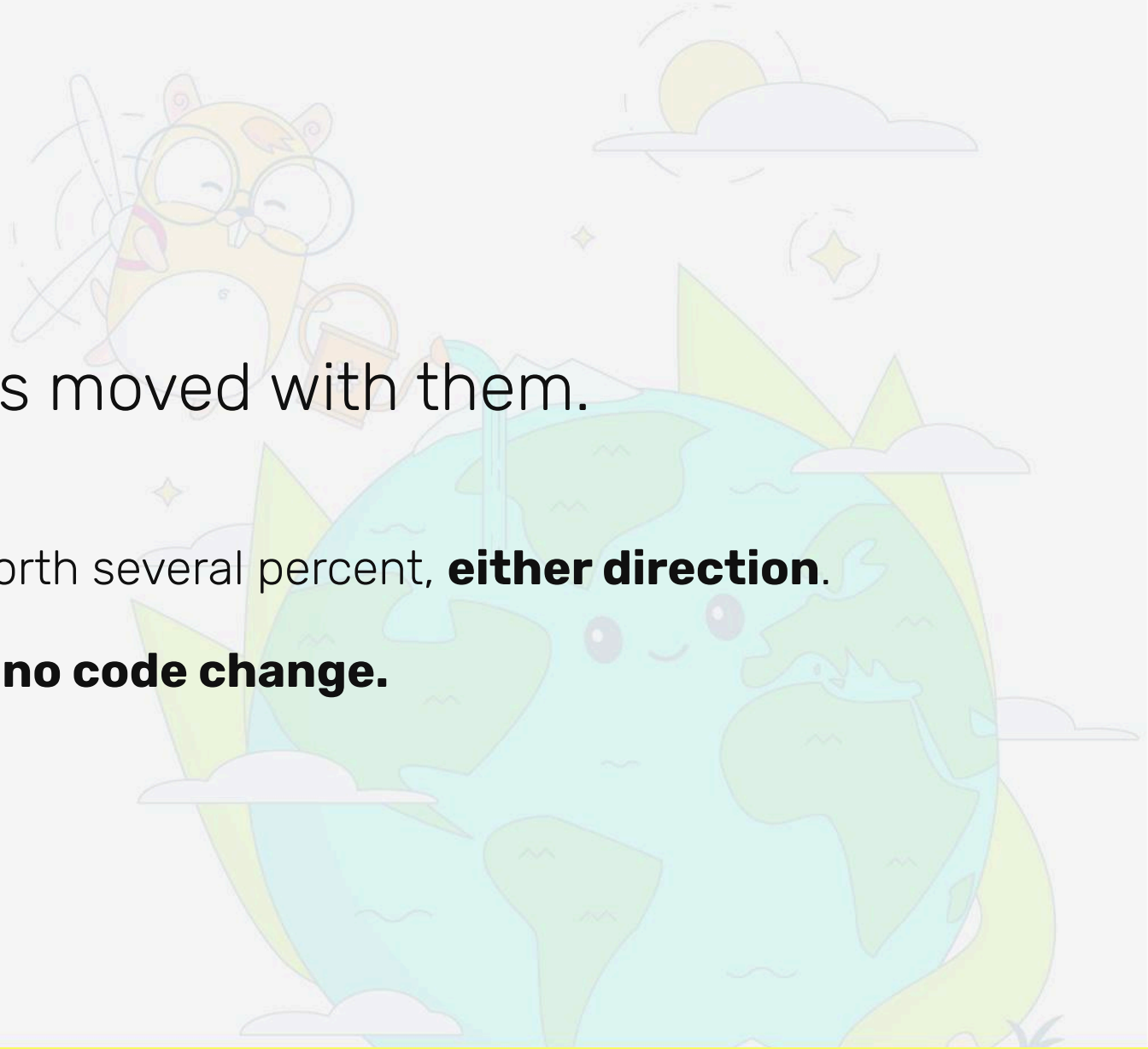
# The mechanism

Function addresses moved.

Cache lines and branch targets moved with them.

At sub-microsecond scale, alignment is worth several percent, **either direction**.

**Resolution: no code change.**



# A known phenomenon

**Performance Matters**

Emery Berger • Strange Loop 2019

Code layout alone swings performance **±10%**.



# A noisy result can be directionally wrong

It blocks good changes and waves regressions through.



# One number is a point sample

```
$ noisy.txt

BenchmarkMakeBuffer_Correct-16    41877204    39.39 ns/op
...
BenchmarkMakeBuffer_Correct-16    52521198    27.54 ns/op
```

Same binary. **43% swing.**




you, when someone says "p-value"

# benchstat

```
$
go test -bench=. -benchmem -count=10 . | tee new.txt
benchstat old.txt new.txt
```

Is B different from A, beyond the noise?

`~` means **no**. That is a result.

It cannot tell you whether the **machine** is trustworthy.

golang.org/x/perf/cmd/benchstat



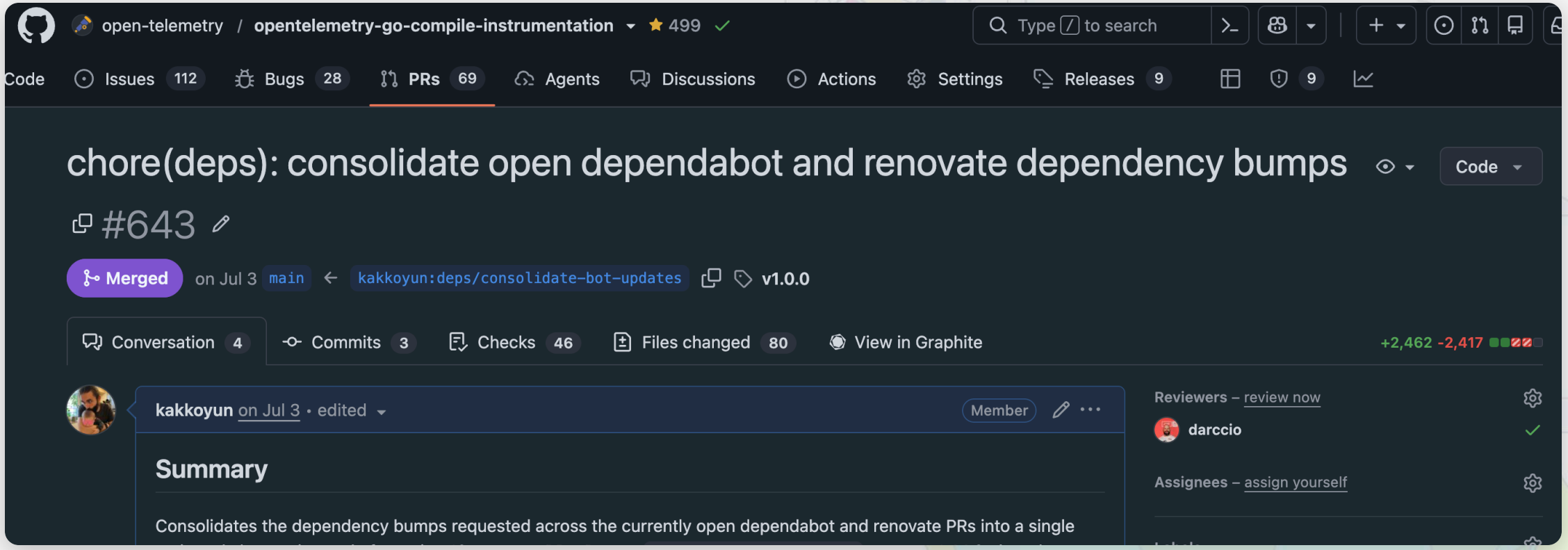

# more samples will not fix a noisy machine

04

# It Was the Machine

Question 3: above the noise?

# OpenTelemetry Go compile-time instrumentation #643




open-telemetry/opentelemetry-go-compile-instrumentation #643 • July 2026

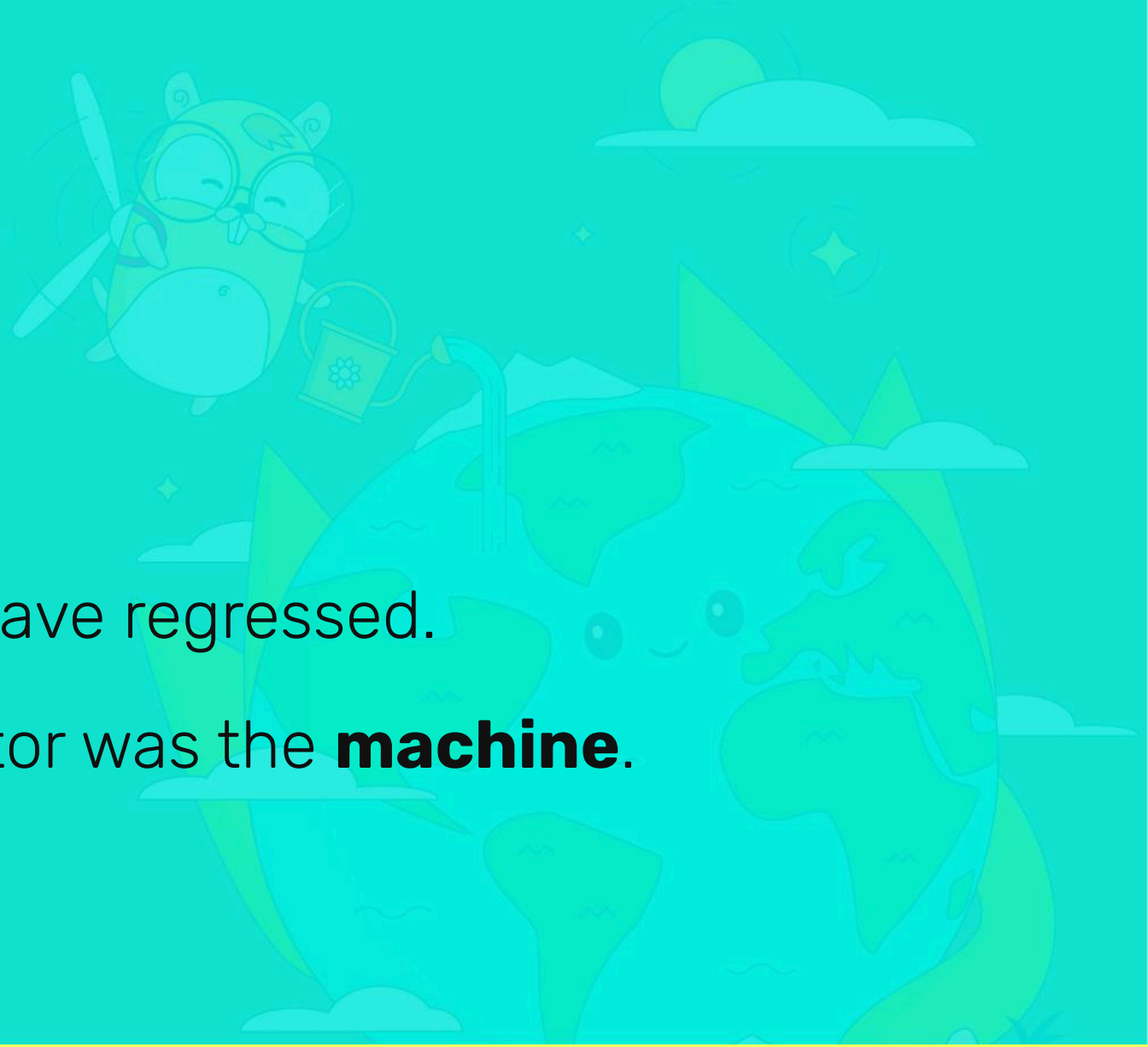
# A second red number




A local run after a dependency fix. It looks like a serious regression.

`largeidle` **shares**

**zero changed dependencies**



It cannot have regressed.

The common factor was the **machine**.

# What CI said

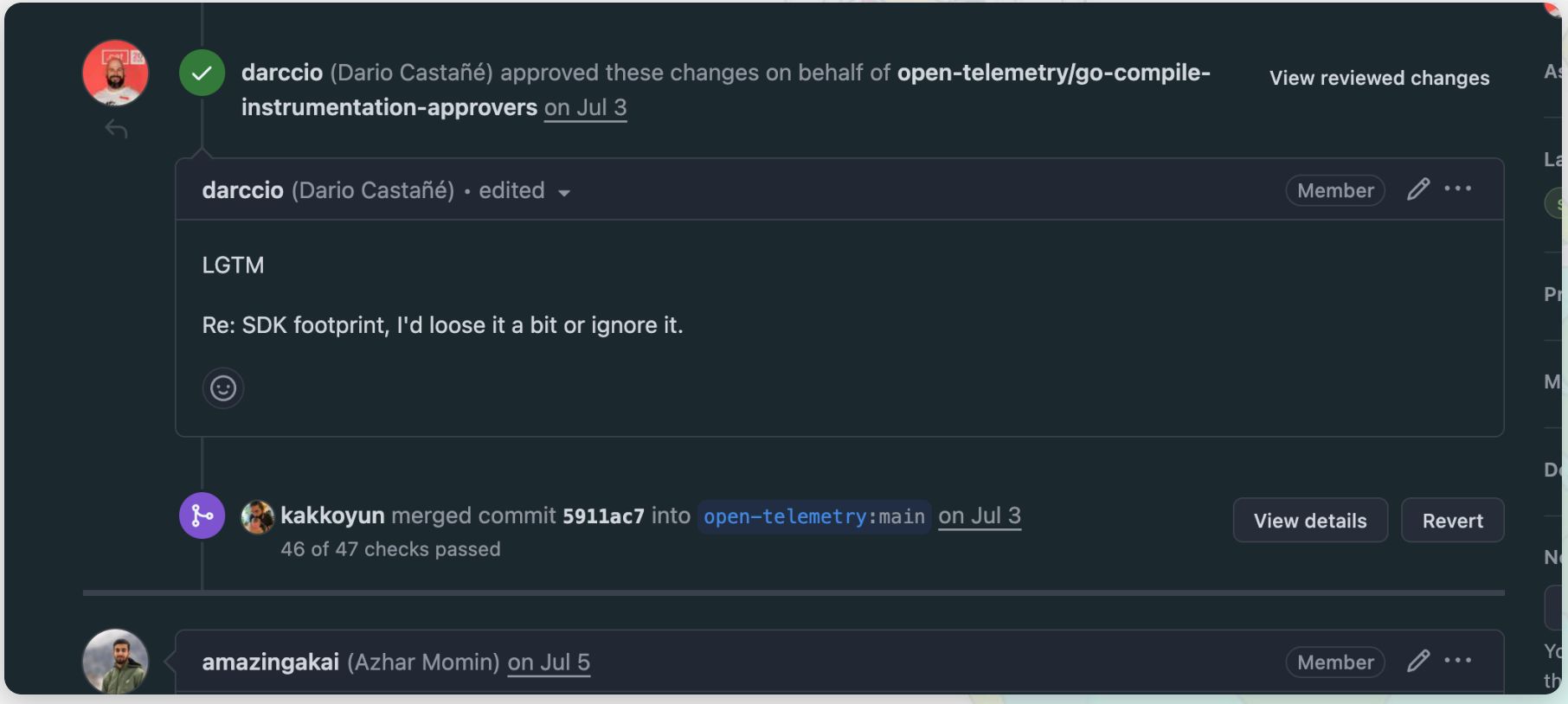
`Overhead Threshold Check` fails consistently, not flaky. Reran it once to confirm:

| Run | baseline plain | baseline otelc | overhead | ceiling |
|---|---|---|---|---|
| 1st | 2.146s | 13.197s | +514.9% | 500% |
| rerun | 3.432s | 21.344s | +521.9% | 500% |

`largeidle` (+105%) and `multi` (+81%) stay well under their 150% ceiling both times. Only `baseline`, which the test comment says measures pure OTel SDK overhead with no instrumentation rules matched, crosses its ceiling.

`largeidle` **+105%**, `multi` **+81%**: under the ceiling. Only `baseline` failed, **twice**.

# The ceiling moved, on purpose




A real signal gets a **decision**, not a fix: #650 raised it to 550%.

# The mirror image

#4891

CI said slower.

The laptop was right.

Cause: **code layout**.

#643

The laptop said slower.

CI was right.

Cause: **machine load**.

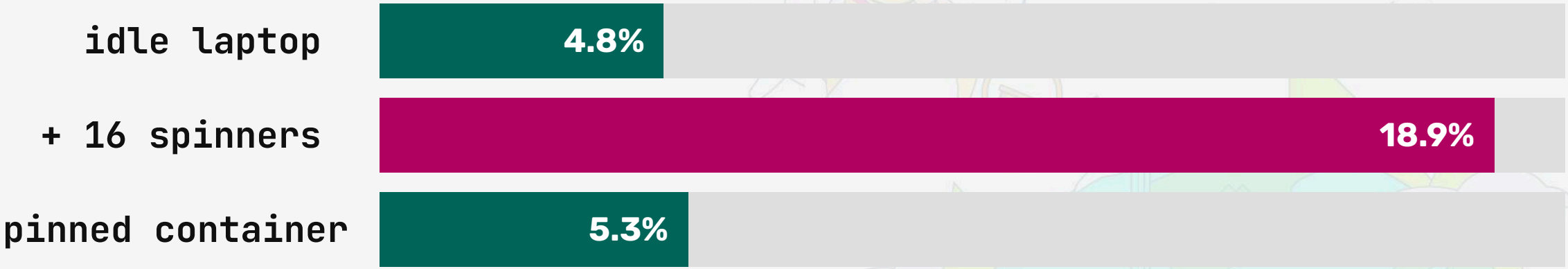
Neither environment is authoritative by default.

Same tell: **a benchmark moved that could not have moved.**




# PR #4891 and PR #643

# What does isolation actually buy?




Run-to-run **spread**. Shorter is better.

One local run, August 2026 • Apple M4 Max • container under Apple Virtualization.framework • `demo/results/cv-summary.txt`

# 5.25% is not a triumph

# It is a ceiling

For this benchmark, under this load, a change smaller than the spread is invisible.

05

# How We Run It at Datadog

Question 3, at hardware scale.

# What we do at Datadog

**Dedicated hardware**

not shared runners

**An overhead budget**

per SDK

**Several workloads**

per SDK

**Gate the release**

not only the PR

# Why not just add more runners?

Shared runners hide real regressions and invent phantom ones.

> There is often only one physical machine for some configurations, and often backlogged with work […]

The Go project, about its **scarce** builders.

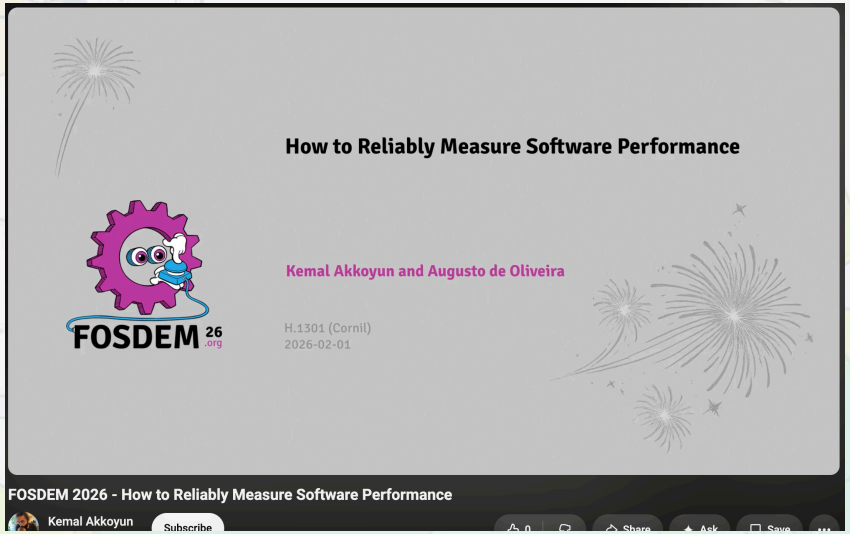
go.dev/wiki/SlowBots

# FOSDEM 2026, Software Performance devroom

## How to Reliably Measure Software Performance

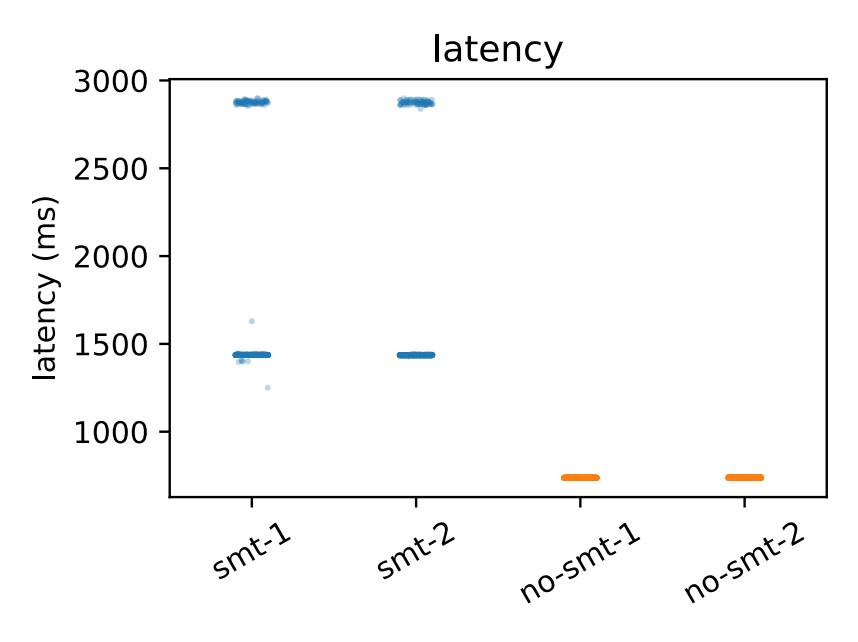
Kemal Akkoyun & Augusto de Oliveira

The SMT and DFS experiments on the next slides come from this talk.



# What's the impact of disabling SMT?

**SMT**: simultaneous multithreading, two hardware threads per core



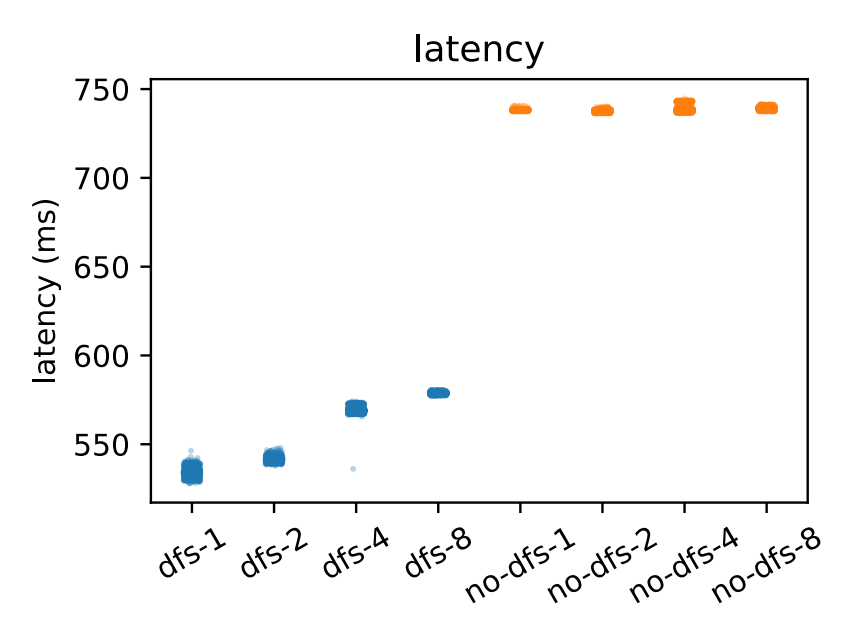

Spread: **~24%** → **under 0.25%**

**~100× less variance**, and twice as fast.

AWS m5.metal, FOSDEM 2026 experiment

# What's the impact of disabling DFS?

**DFS**: dynamic frequency scaling, the clock follows load and heat




Spread: **0.38%** → **0.04%**

**~10× less variance**, and the mean got *slower*.

AWS m5.metal, FOSDEM 2026 experiment

# Three sysfs writes

```
echo off         | sudo tee /sys/devices/system/cpu/smt/control
echo performance | sudo tee /sys/.../scaling_governor
echo 0           | sudo tee /sys/devices/system/cpu/cpufreq/boost
# Intel with intel_pstate: no cpufreq/boost, use this instead
echo 1           | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo
```

## Bare metal

Spread ~24% → **under 0.25%**

## In a VM

The write may **succeed and be ignored**.



# you, after three sysfs writes

# No bare metal? Rent the scarce machine

**1**

`/prombench` comment on a PR

**2**

Dedicated nodes come up

**3**

PR vs baseline, same workload

**4**

Results, then the nodes go away

Prometheus • prometheus/test-infra

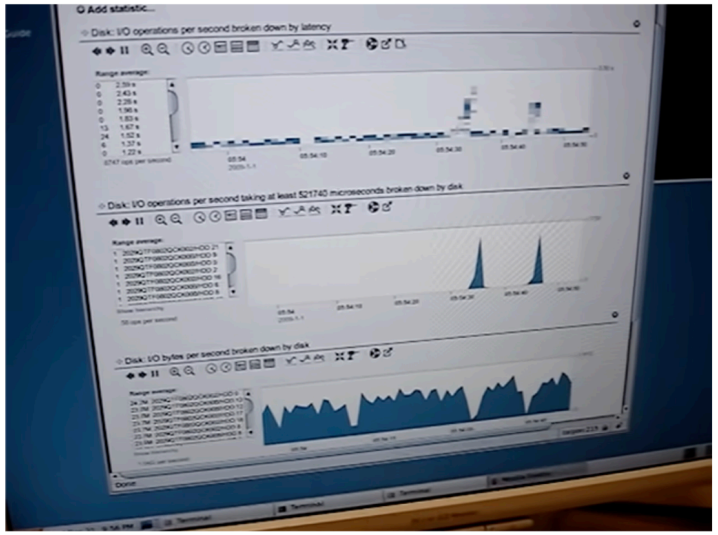
# Noise goes all the way down



**Shouting in the Datacenter**

Bryan Cantrill
8.08K subscribers
Subscribe

**4.8M views 17 years ago**
Brendan Gregg from Sun's Fishworks team makes an interesting discovery about inducing disk latency. For a ca. 2020 retrospective on this 2008 video: • Bryan Cantrill talks Sun Microsystems, DTr ...more




**Don't shout in the datacenter.** • Brendan Gregg

# Workload archetypes

| ARCHETYPE | PATTERN | SHAPE |
|---|---|---|
| **Idle** | background workers | `low RPS, low CPU` |
| **Latency** | microservices, APIs | `high RPS, low CPU/req` |
| **Throughput** | queues, batch | `high CPU, many clients` |
| **Enterprise** | business apps + DB | `mixed CPU / I/O` |

`largeidle` is Idle. `multi` is Enterprise. Both moved:

**shared environment, not shared cause.**

# The load generator can lie too

| | | | |
|---|---|---|---|
| Server stalls | Generator waits | Stall never recorded | p99 looks better |

Less measured, better numbers.

Gil Tene • How NOT to Measure Latency • coordinated omission

# A macro gate is a budget

```
scenario: multi
baseline: uninstrumented process
ceiling:  150% overhead
result:   230%   → FAIL
```

Micro gates compare commits. Macro gates compare against a **budget**.

06

# Detecting Change Over Time

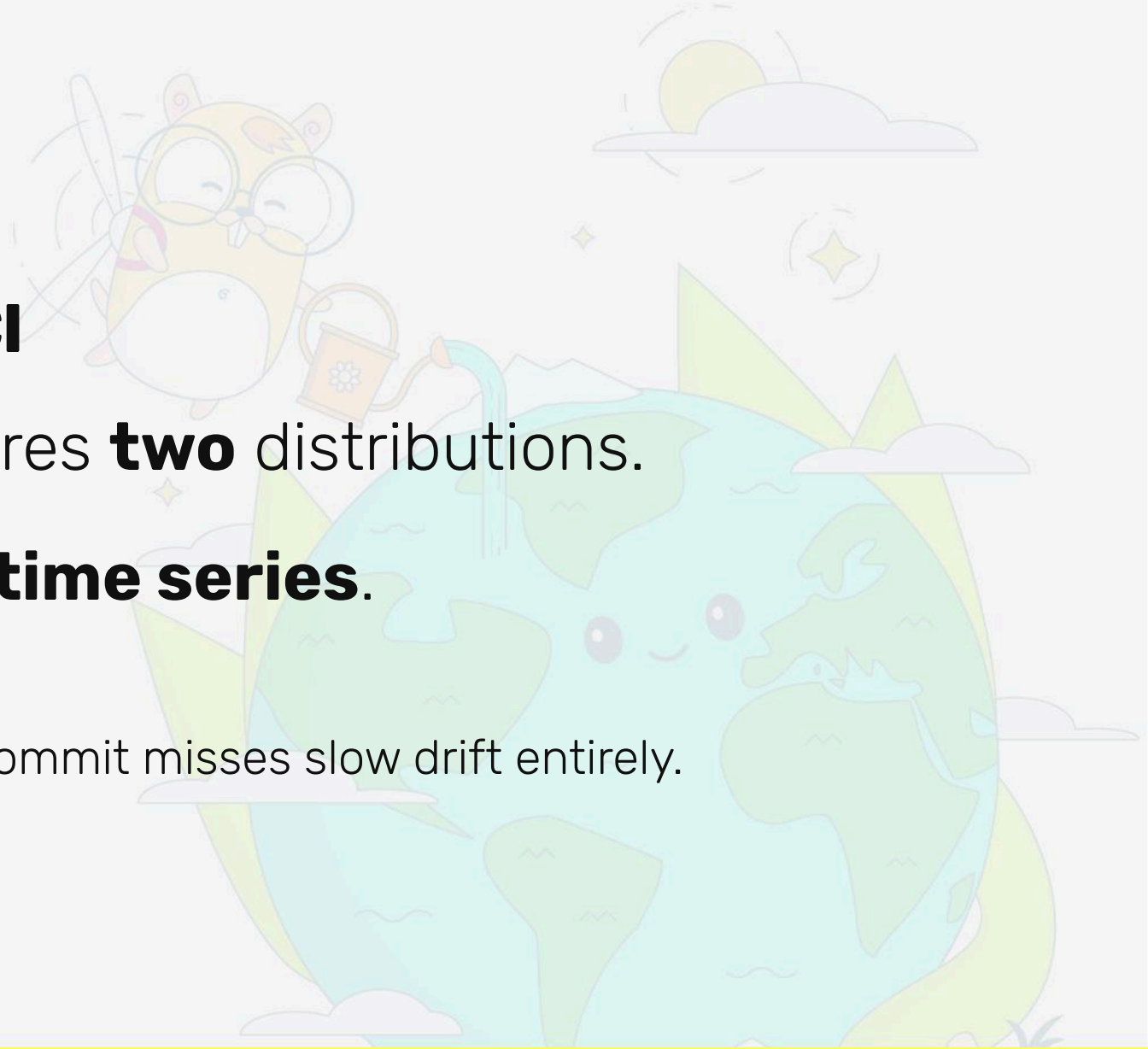
Question 2, across days.

# A/B is the wrong model for CI

Benchstat compares **two** distributions.

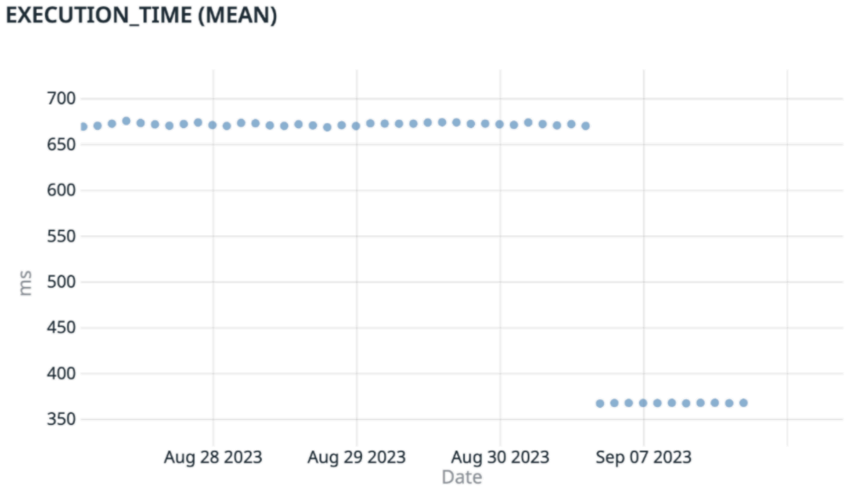
CI has a **time series**.

Comparing to the parent commit misses slow drift entirely.

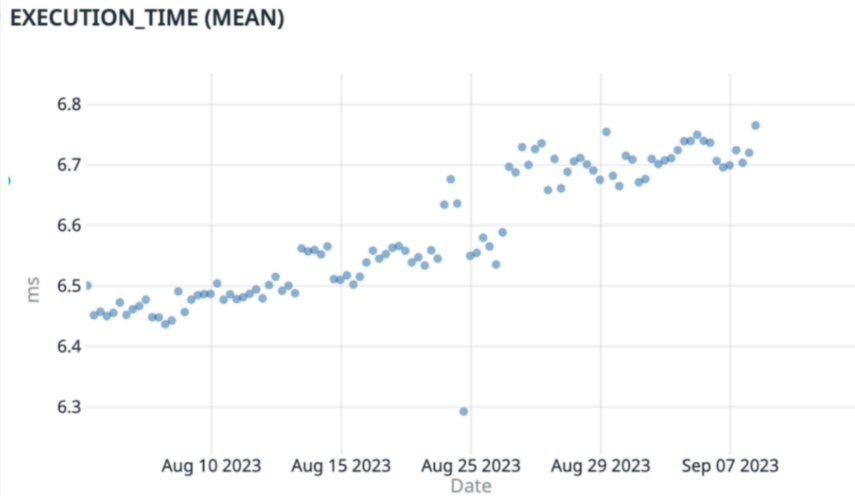


# What regressions look like

## In a slide deck




## In real data



# Change-point detection

Not *"slower than its parent?"*

But *"where did it shift and stay shifted?"*

**Why**

Finds lasting shifts.

Ignores one-off spikes.

**Implementations**

**ED-PELT** • Akinshin

**e-divisive** • Apache Otava, Nyrkiö

Netflix, at scale



you, nodding along to change-point detection

# Change points in the wild

## Netflix TV UI

Static thresholds out. Anomaly and change-point detection in CI.

Fixing Performance Regressions Before They Happen

## Henrik Ingo • FOSDEM 2026

Same devroom. Continuous benchmarking with change points and fewer false positives.

Continuous Performance Engineering HowTo

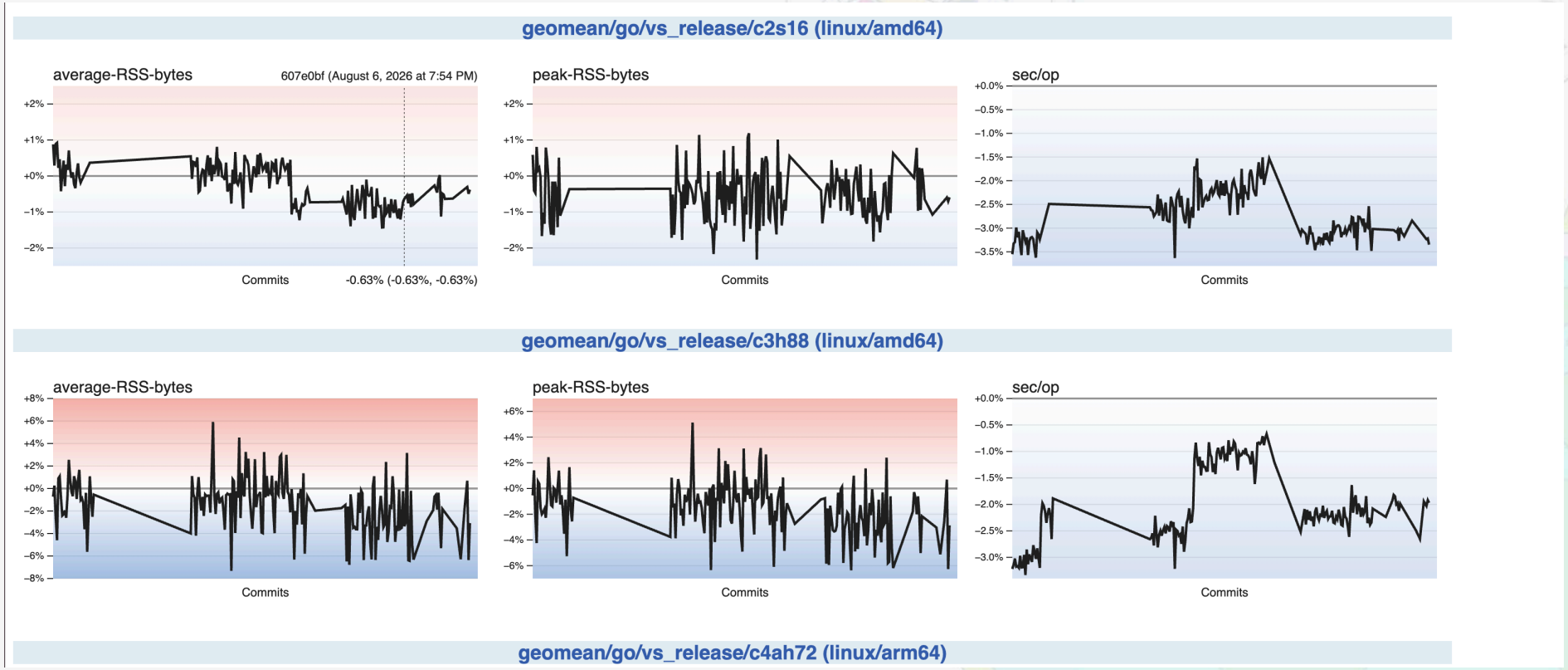
# Why a baseline, not a pair

> We never report performance numbers in isolation, and only relative to some baseline. […] The state of a machine or VM on one day is likely to be very different than the state of a machine or VM on the next day.

That is the Go team, about **their own hardware**.

go.dev/wiki/PerformanceMonitoring

# The Go team does this




perf.golang.org/dashboard • every commit • against the last release

# Before merge, on request

`perf_vs_parent`

the change, in isolation

`perf_vs_tip`

the change, against today's tip

Opt-in per change, on the scarce builders. The trusted signal is still the time series.

go.dev/wiki/PerformanceMonitoring

# Two patterns

## PR gate

Fast. Strict. Every PR.

Benchmarks that regressed before.

`-count=10`, same-machine A/B

## Nightly

Complete. Pinned runner.

All controls applied.

`-count=20`, change-point detection

CI **detects**. A controlled machine **explains**.




# BenchmarkOTLPProtoSize, eleven days later

# Keep a ledger of false positives

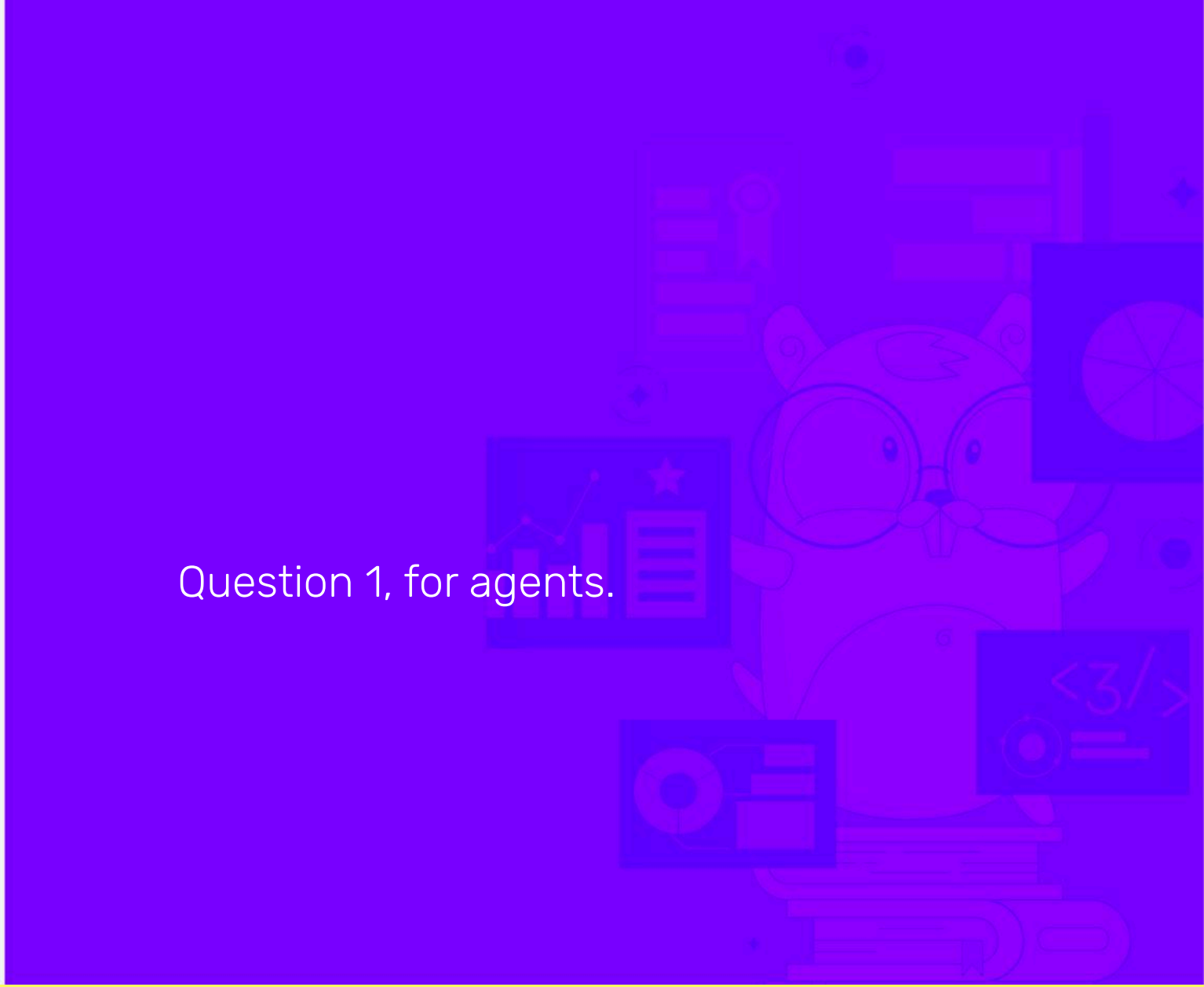
> Known `BenchmarkOTLPProtoSize` false positive, documented for #4891 as a same-package code-layout artifact. Touches no code I changed; local A/B was ~+0.3%. No action.

Documented noise is **benchmark hygiene**.

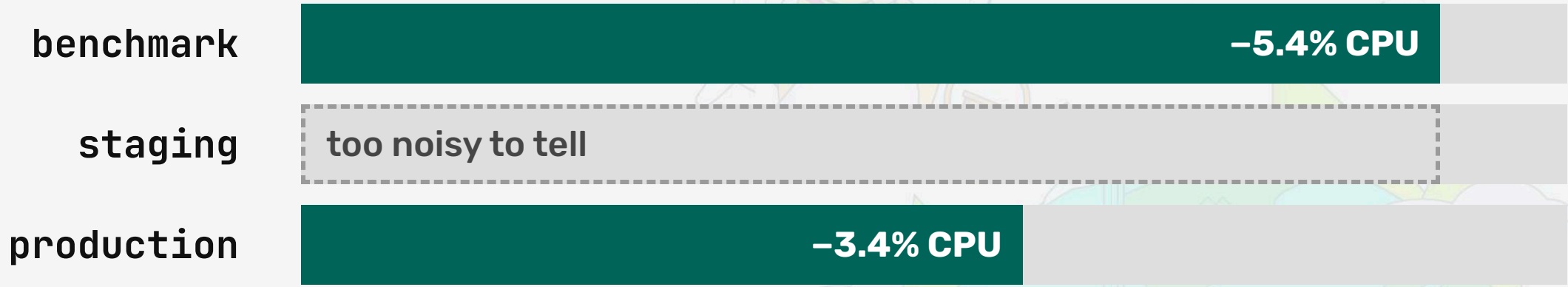
07

# Agents Need Real Benchmarks

Question 1, for agents.



# A benchmark is a prediction




PGO on one large Datadog service • Datadog blog, May 2024

"Get the benchmark wrong, and the agent finds real speedups on the wrong hill."

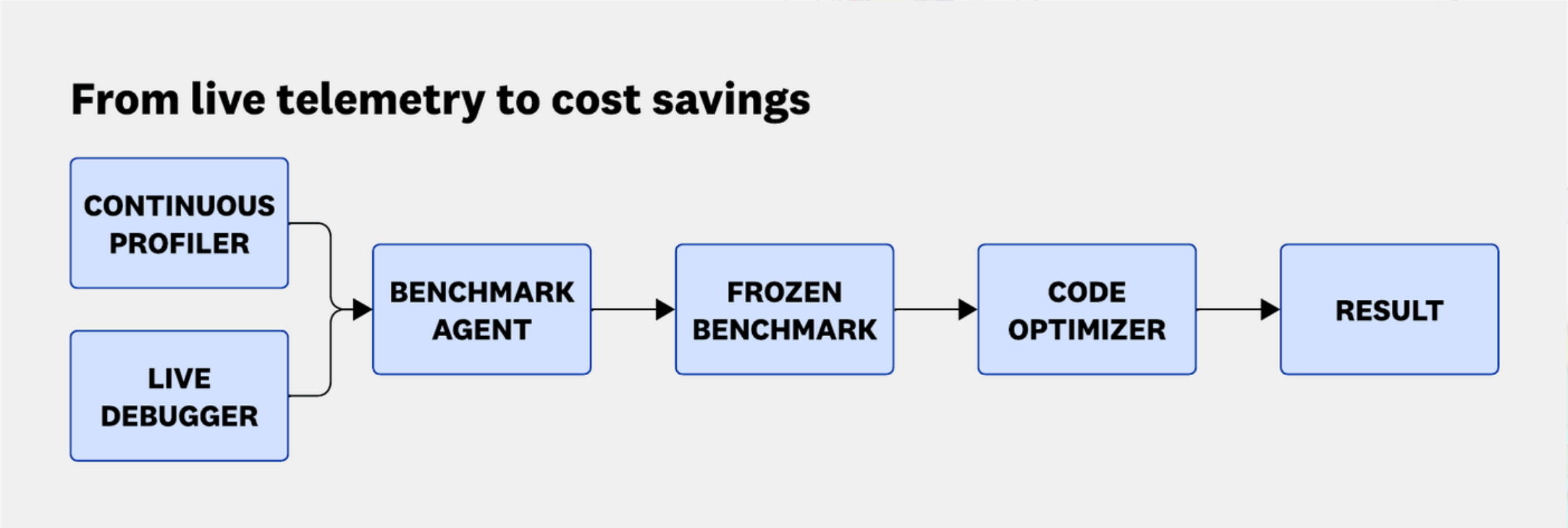
Datadog engineering blog, June 2026






the agent, after it made the benchmark faster

# Ground the benchmark in production




**DODO:** Datadog Observability-Driven Optimizer

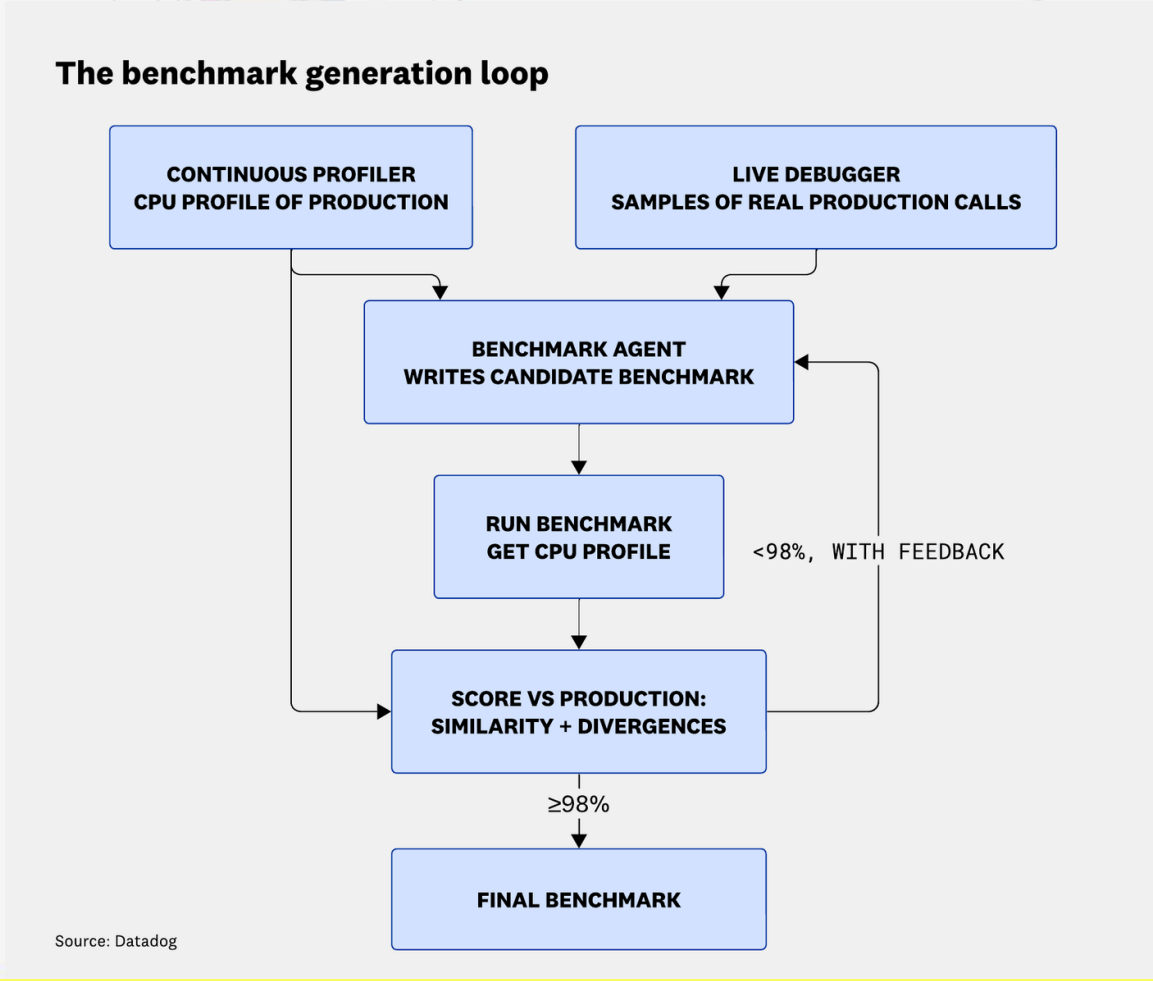
# Fit first, then freeze

Real calls.

Real state.

Same CPU.

Then **freeze** it. The optimizer cannot touch the benchmark.



# What it found

**>8%** of a critical service's CPU

**O(10k)** cores, around the clock

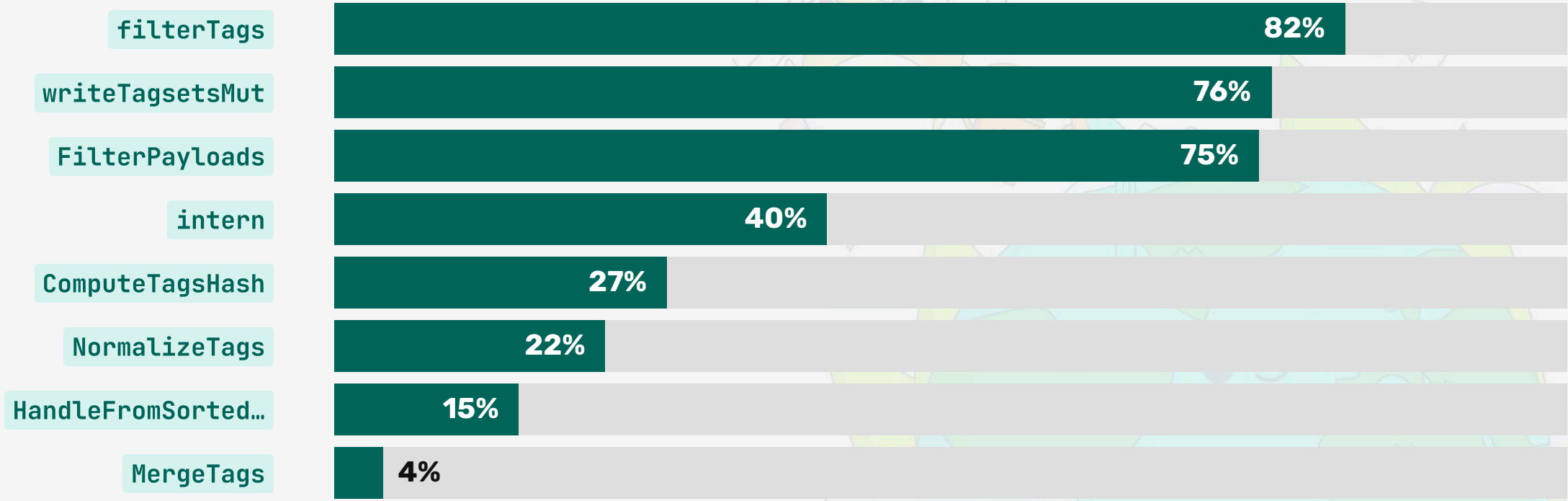
`NormalizeTags`

About 25% of real tags have uppercase letters.

The agent found a fast path for that mix. A synthetic benchmark hides it.

Datadog blog, June 2026: production-grounded code optimization

# Even optimized code had room




CPU saved in each function, one mature Datadog service • Datadog blog, June 2026

# `dodo-cli` • internal beta

`explore`

Ranks the hottest production functions by how much they could gain.

`optimize go <func>`

Grounded benchmark first, then an agent loop.

## Languages

Go • Java • Python (experimental)

## Agents

Claude Code • Codex

# Benchmarking is not easy

**Investment**

dedicated hardware

**Maths**

statistics you trust

**Tooling**

benchstat • change points • agents

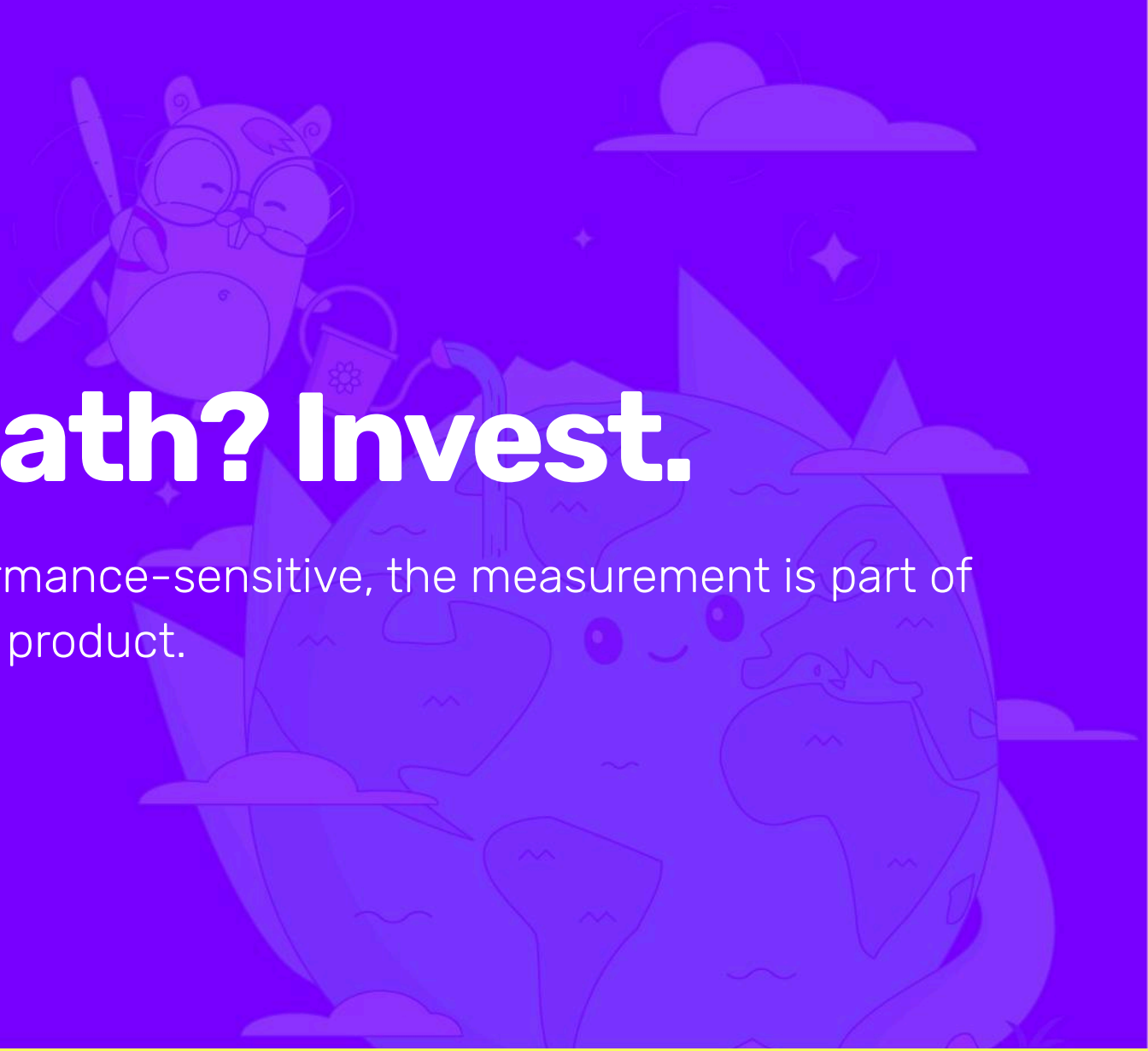
**A keen eye**

what could not have moved

Datadog does it. The Go team does it.

# On a hot path? Invest.

A library, a tool, or a service: if it is performance-sensitive, the measurement is part of the product.



# Now you know what to do

| | LAPTOP + MICRO | CI + MACRO |
|---|---|---|
| **Real work?** | `b.Loop` • `-benchmem` | archetypes · budgets |
| **Stable sample?** | `-count=10` • benchstat | time series · change points |
| **Above noise?** | control benchmark • know the ceiling | bare metal · SMT + DFS off |

A sub-10% micro delta can be layout noise.
A shared runner cannot be fixed with more samples.

# Take it with you

## Check my numbers

Talk repo: decks, script, captured results

Earlier FOSDEM version



**Scan for the talk repo**

# Questions?

kakkoyun.me • github.com/kakkoyun

linkedin.com/in/kakkoyun • bsky.app/profile/kakkoyun.me • x.com/kakkoyun_me

Talk repo github.com/kakkoyun/golab-26

