Zero-Touch

Go Instrumentation

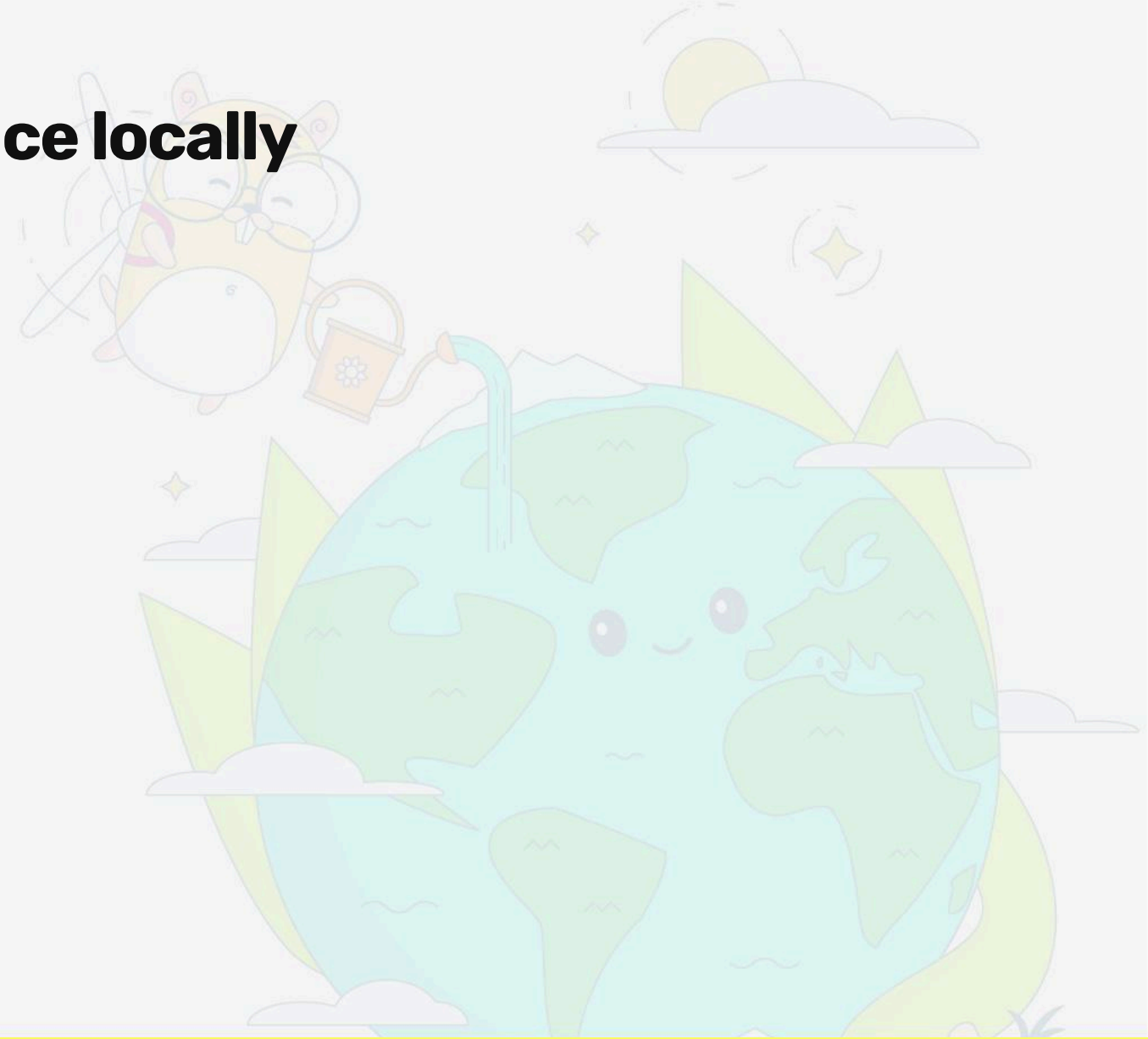
How to Instrument Go Without Changing a Single Line of Code

Kemal Akkoyun • Datadog

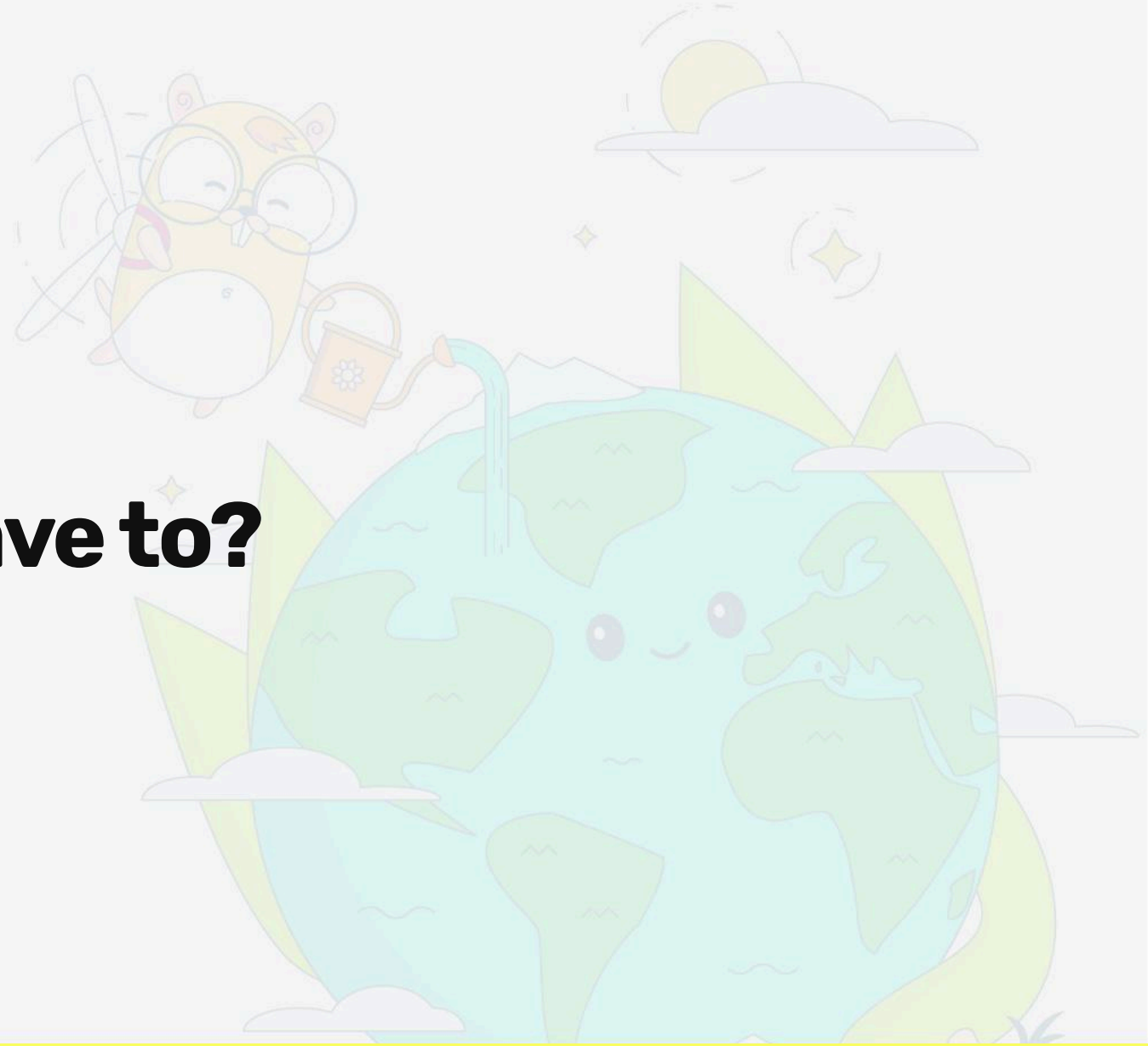


The bug you cannot reproduce locally

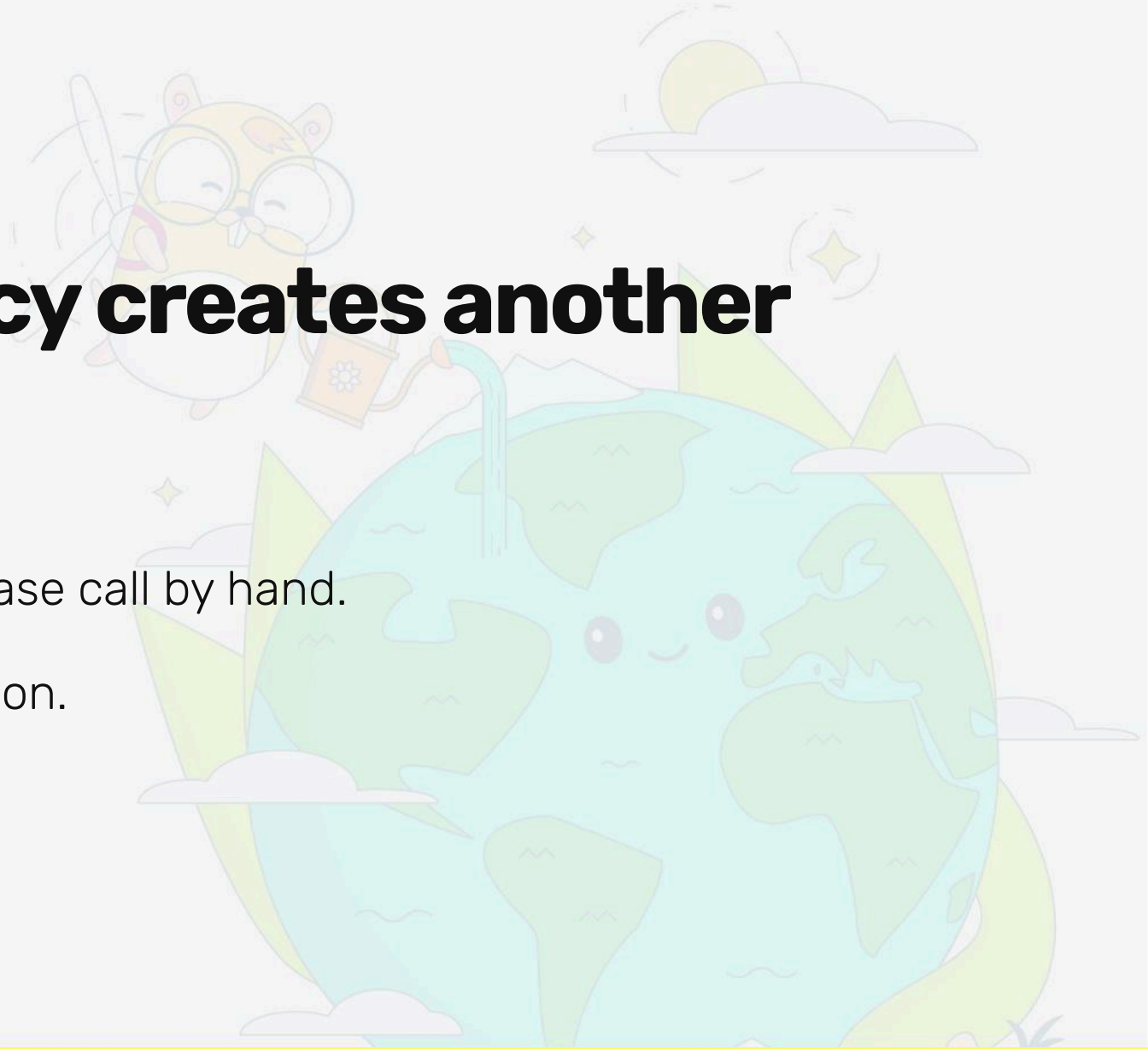
- Added a log line. Redeployed.
- Wrong place. Did it again.



What if you did not have to?



Every new dependency creates another place to forget



You wrapped every HTTP client and database call by hand.

The first service gets careful instrumentation.

The fiftieth inherits gaps.

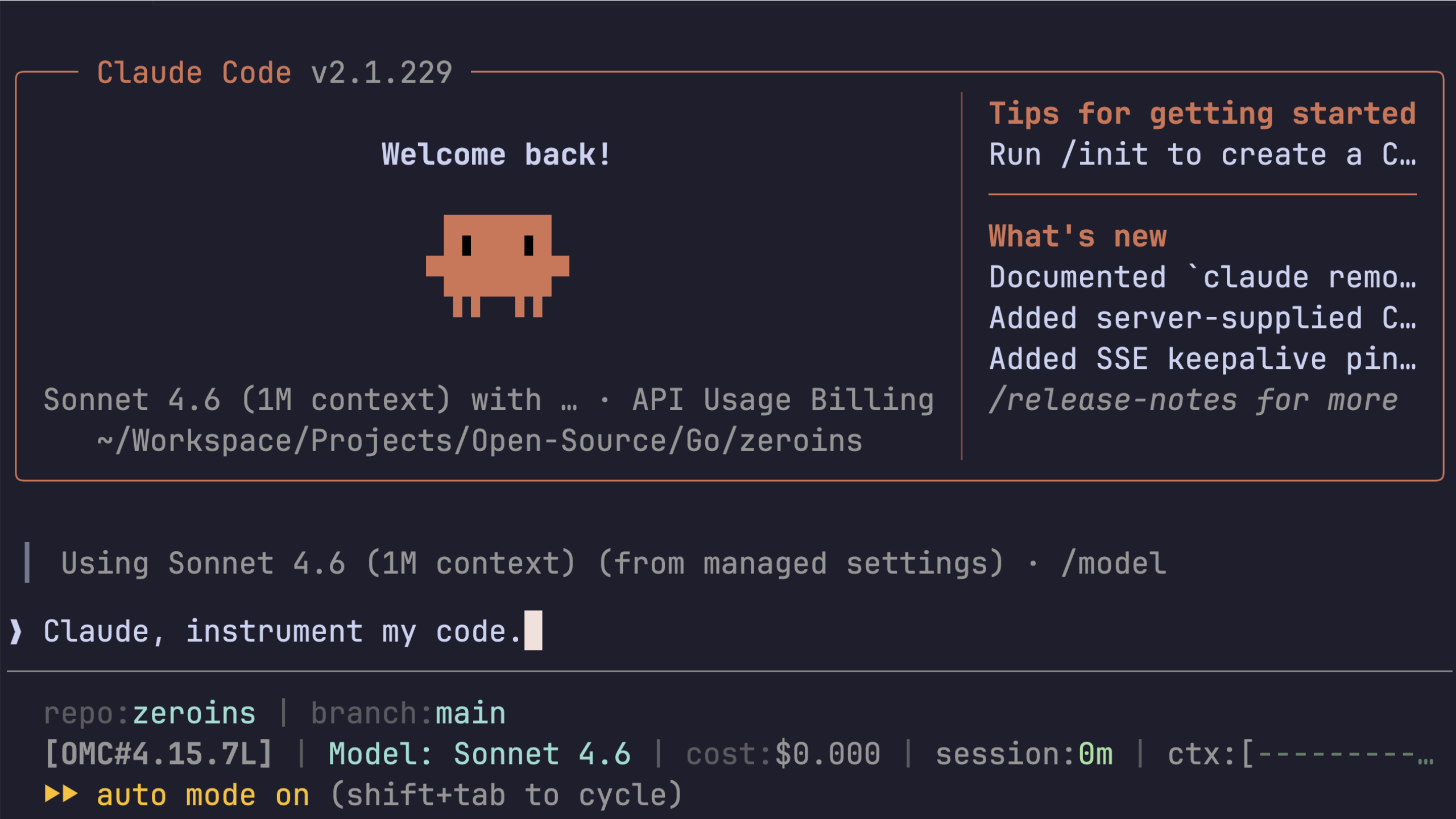
Go compiled away the attachment point

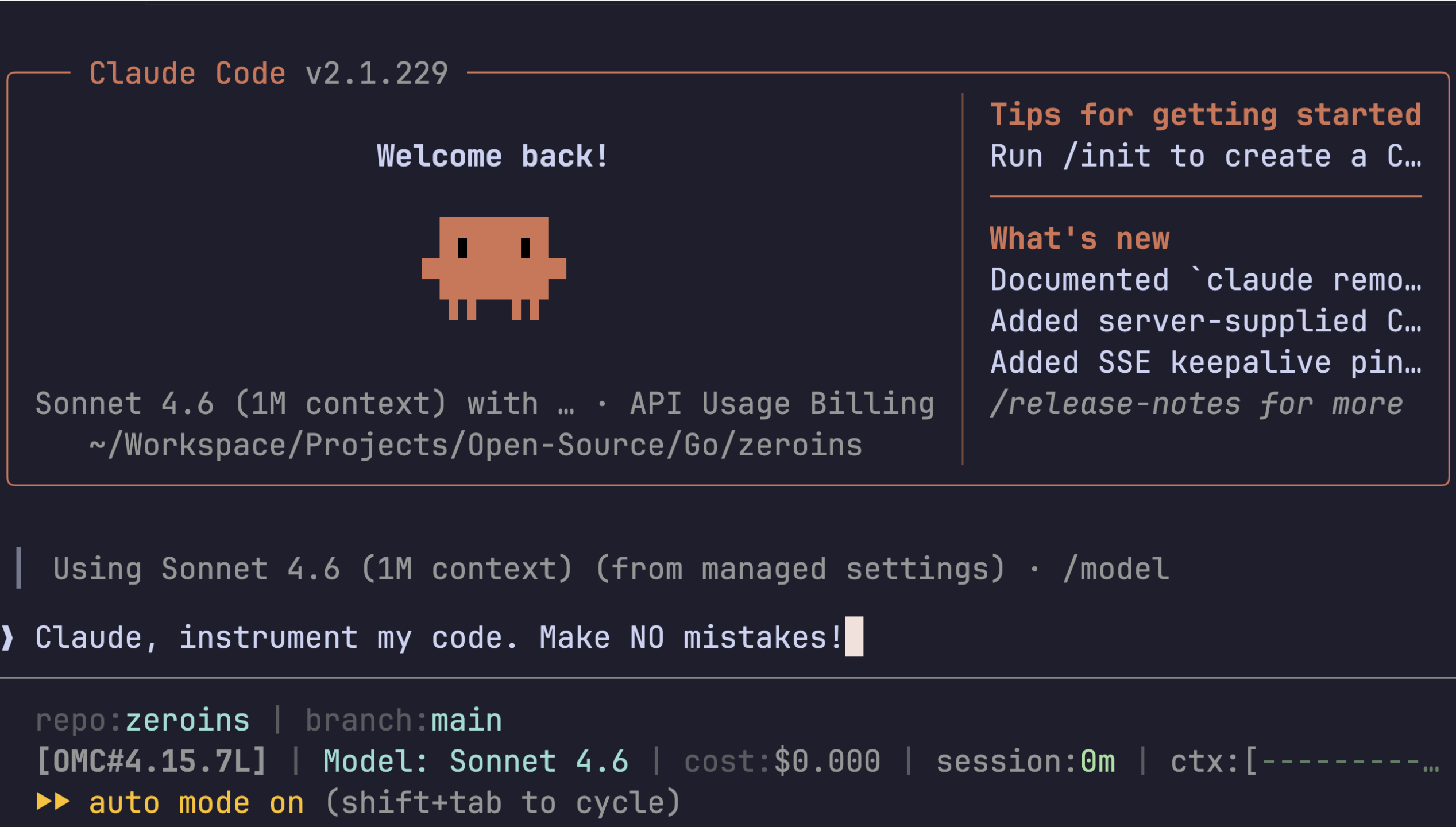
RUNTIME	ZERO-CODE MECHANISM
JVM	`-javaagent`, JVMTI, bytecode rewriting
Python	Import hooks, `sys.settrace`
.NET	CLR profiling APIs and startup hooks
Go	Native binary. No classloader. No bytecode.

Go made deployment simple. It made late instrumentation hard

Hand-written instrumentation decays

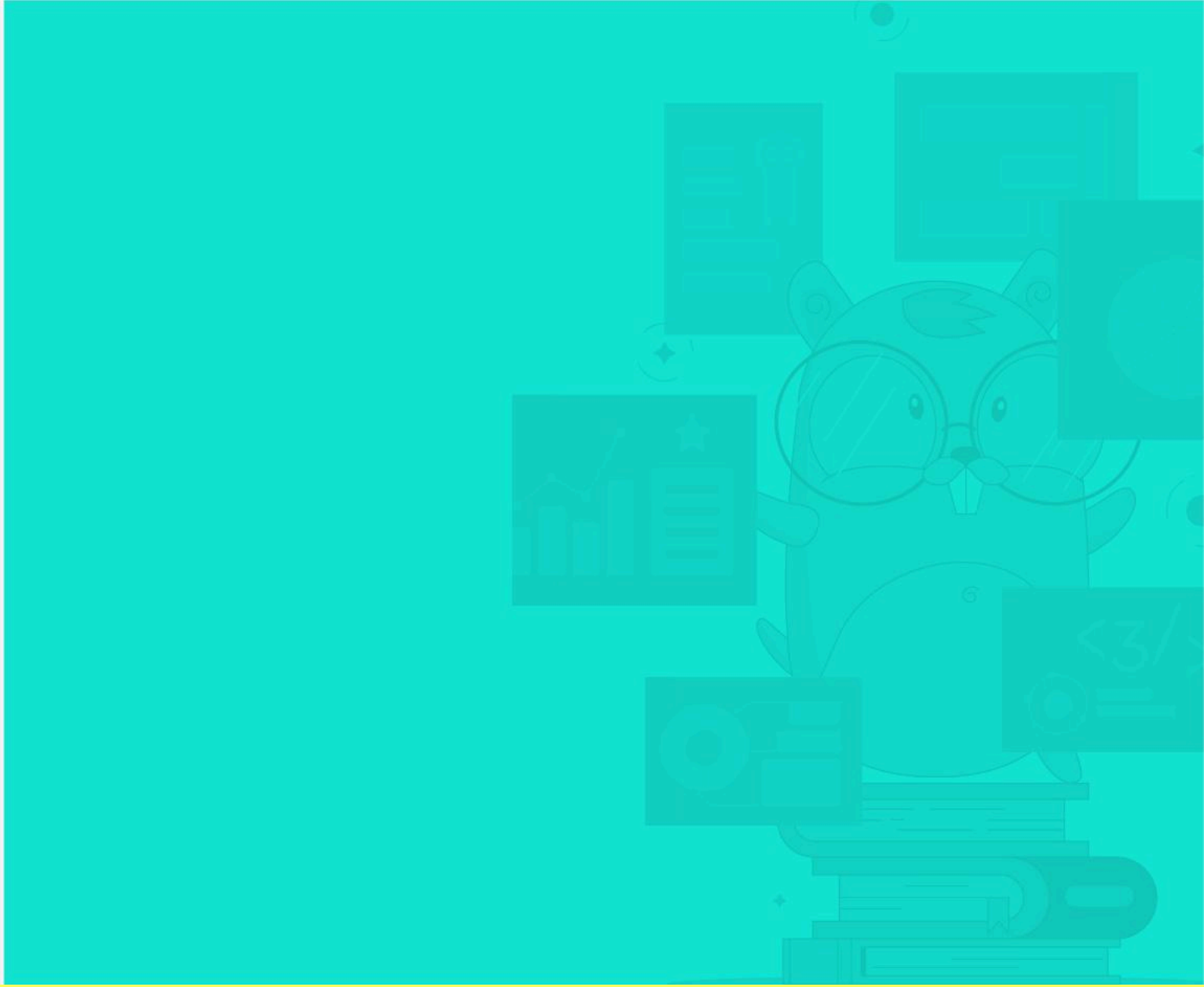



Claude Code v2.1.229
Welcome back!
Sonnet 4.6 (1M context) with … · API Usage Billing
~/Workspace/Projects/Open-Source/Go/zeroins
Tips for getting started
Run /init to create a C…
What's new
Documented `claude remo…
Added server-supplied C…
Added SSE keepalive pin…
/release-notes for more
Using Sonnet 4.6 (1M context) (from managed settings) · /model
❯ Claude, instrument my code.
repo:zeroins | branch:main
[OMC#4.15.7L] | Model: Sonnet 4.6 | cost:$0.000 | session:0m | ctx:[---------…
▶▶ auto mode on (shift+tab to cycle)


Claude Code v2.1.229
Welcome back!
Sonnet 4.6 (1M context) with … · API Usage Billing
~/Workspace/Projects/Open-Source/Go/zeroins
Tips for getting started
Run /init to create a C…
What's new
Documented `claude remo…
Added server-supplied C…
Added SSE keepalive pin…
/release-notes for more
Using Sonnet 4.6 (1M context) (from managed settings) · /model
❯ Claude, instrument my code. Make NO mistakes!
repo:zeroins | branch:main
[OMC#4.15.7L] | Model: Sonnet 4.6 | cost:$0.000 | session:0m | ctx:[---------…
▶▶ auto mode on (shift+tab to cycle)

01

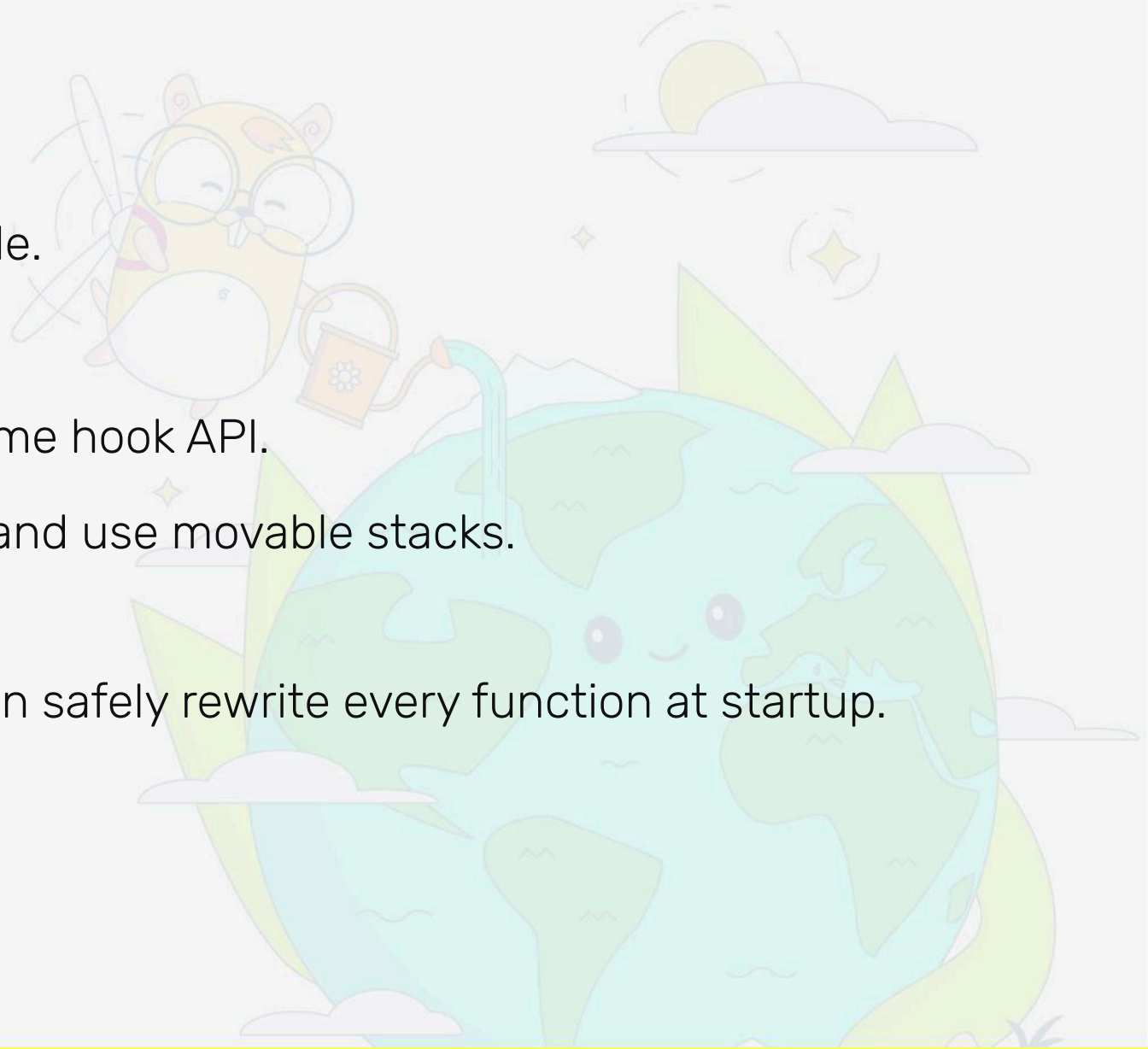
Why Go resists instrumentation



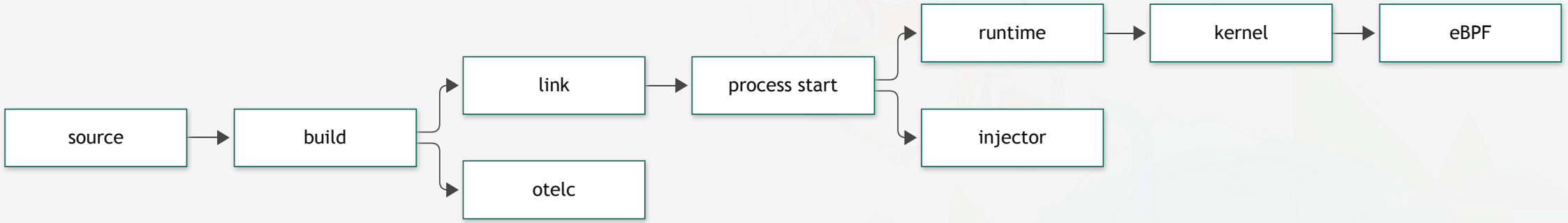
What makes Go different

- The compiler emits native machine code.
- Pure-Go builds are static by default.
- There is no classloader or general runtime hook API.
- Goroutines move between OS threads and use movable stacks.

There is no single point where an agent can safely rewrite every function at startup.



Let the constraint pick the layer



LAYER	TRADE-OFF
Build	Rebuild required; rich Go semantics; portable binary
Process start	No source change; binary/linking restrictions
Kernel	No rebuild; Linux, privileges, and kernel contracts

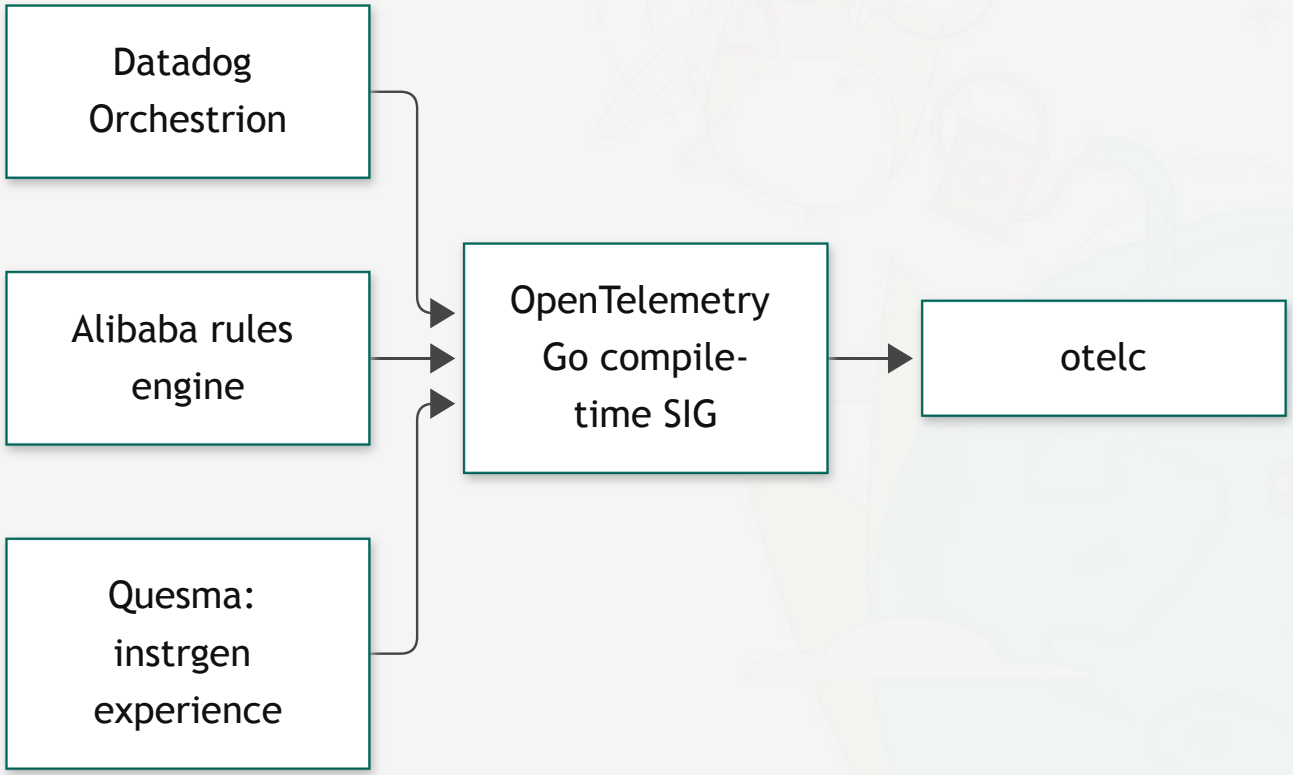

Three intervention points.
Four signals...
Traces • Metrics • Logs • Profiles

otelc: instrument during the build

Traces SDK / integrations	Metrics SDK / integrations
Logs SDK / integrations	Profiles SDK / integrations

The build path can reach all four. Coverage depends on the bundle.

Three organizations converged upstream



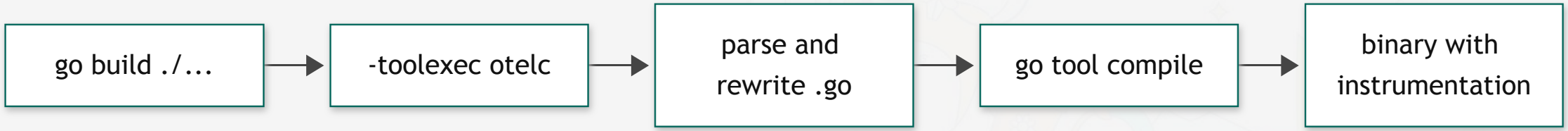
`otelc` is a vendor-neutral tool built by the SIG.

A short convergence timeline

DATE	MILESTONE
Jun 2024	Elastic's profiler joins OpenTelemetry
Jan 2025	Alibaba, Datadog, and Quesma form the Go compile-time SIG
2025	Beyla becomes OpenTelemetry eBPF Instrumentation
Jul 2026	`otelc` v1 becomes stable
Oct 2026	OBI v1.0.0 ships a stable core

The projects did not collapse into one agent. They learned to compose.

`-toolexec` puts otelc before the compiler



about 20%

Growth in customer adoption of Orchestrion-based Go auto-instrumentation

`otelc` v1 is stable upstream. One build-command change can standardize instrumentation across a fleet.

Build-time instrumentation is portable

	OTELC
Runtime privilege	None
Linux kernel dependency	None
Build pipeline change	`otelc go build`
Tested build targets	Linux, macOS, Windows
Go baseline	Go 1.26+
Runtime cost	Injected SDK and instrumentation code

The trade-off is build ownership, not operating-system access.

Signal depth depends on the integration bundle

Upstream otelc today

- Traces
- HTTP and gRPC metrics
- Go runtime metrics
- Trace IDs in `log`, `slog`, Logrus, and zap output

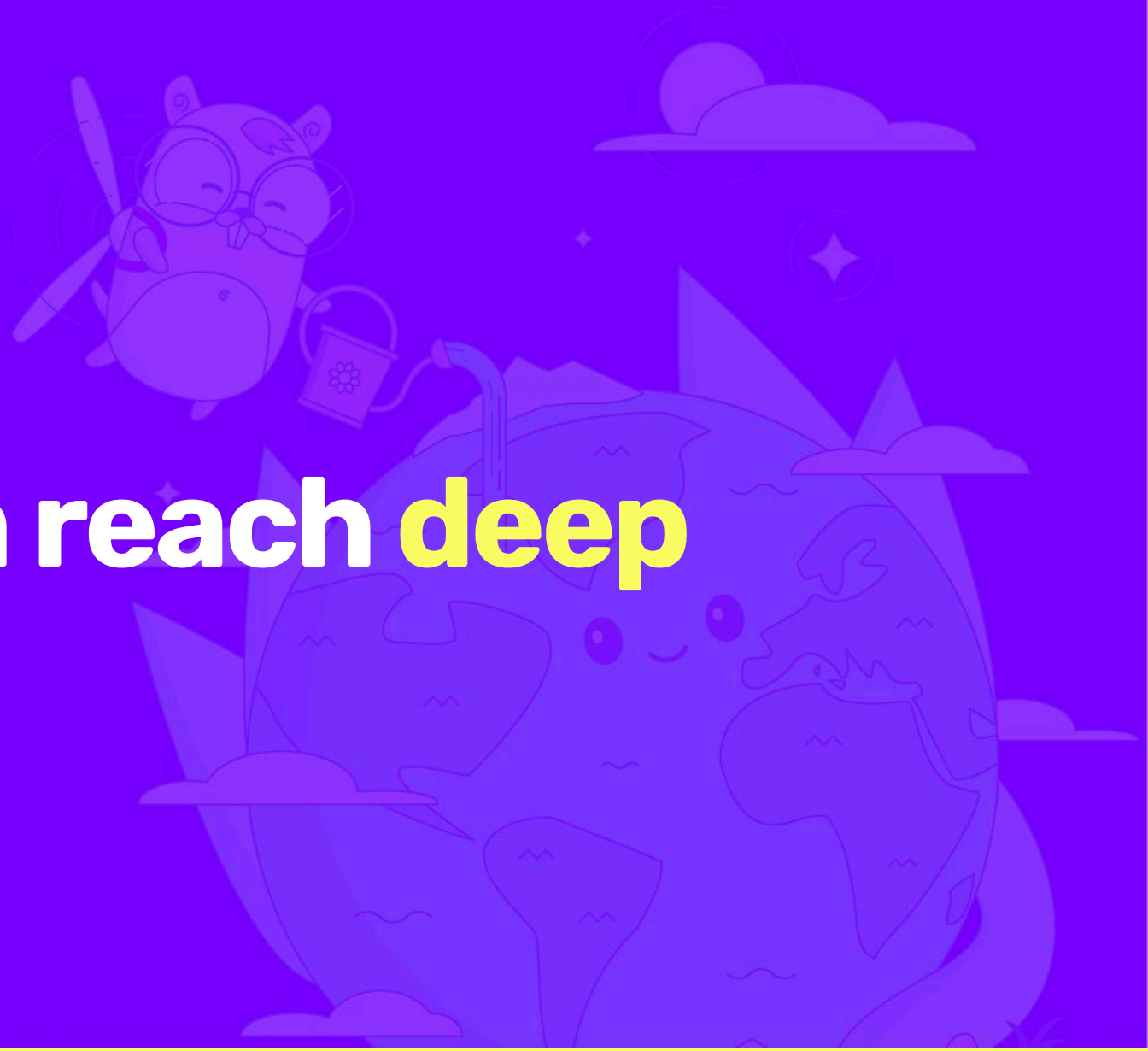
Orchestrion + dd-trace-go

- Traces
- Runtime metrics
- Correlated logs
- Continuous profiles

The build path can reach all four signals. The SDK and integrations determine coverage.

Not every Datadog integration has moved to otelc yet.

And it can reach deep



Runtime internals are reachable. They are not an API.

runtime/proc.go

```
// gopark should be an internal detail,
// but widely used packages access it using linkname.
// Notable members of the hall of shame include:
// ...
```

The Go team calls this access pattern a "hall of shame," not an instrumentation API.

Compile time can reach Go internals! Dark Magic!

gls.orchestrion.yml

```yaml
join-point:
  struct-definition: runtime.g
advice:
  - add-struct-field:
      name: __dd_gls_v2
      type: any
```



The aspect adds goroutine-local storage, typed accessors, and exit cleanup.

`dd-trace-go` owns this aspect. Orchestrion applies it. otelc ships its own `runtime.g` rule.

`go:linkname` variables are fragile

Unlike function linknames, variables have no definition/reference separation.

When both sides were uninitialized BSS symbols, the linker chose by symbol loading order.

> "the choice is arbitrary. In the implementation it depends on the symbol loading order"
>
> Go team, golang/go#72032

This broke between Go 1.22 and 1.23.

Go 1.25 now keeps the larger symbol. Go 1.23.9 and 1.24.3 got the fix too.



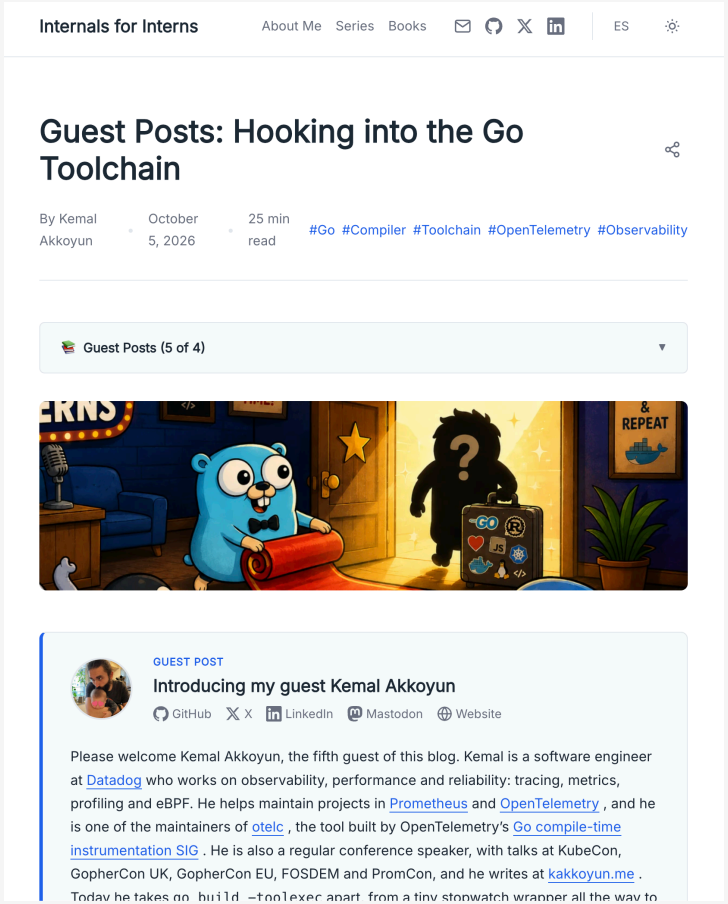
the linker picked by symbol loading order

The build path has boundaries too

- Requires control of the Go build pipeline.
- Requires Go 1.26 or newer for current otelc.
- Coverage stops at the supported integration set.
- Toolchain internals such as `-toolexec` and `go:linkname` can change.

No root access and no kernel version gate, but the binary must be rebuilt.

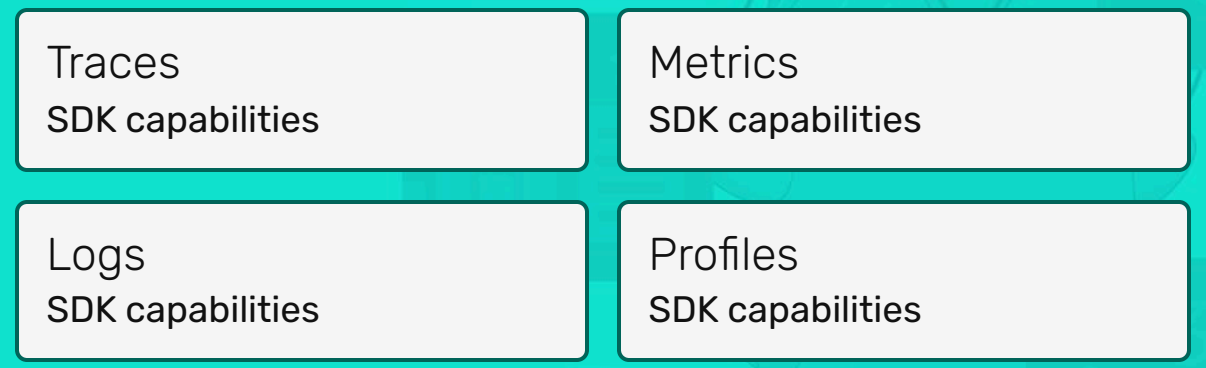
The long version: Hooking into the Go Toolchain



This section is the short version. The guest post starts from a stopwatch wrapper and builds up to `otelc`.

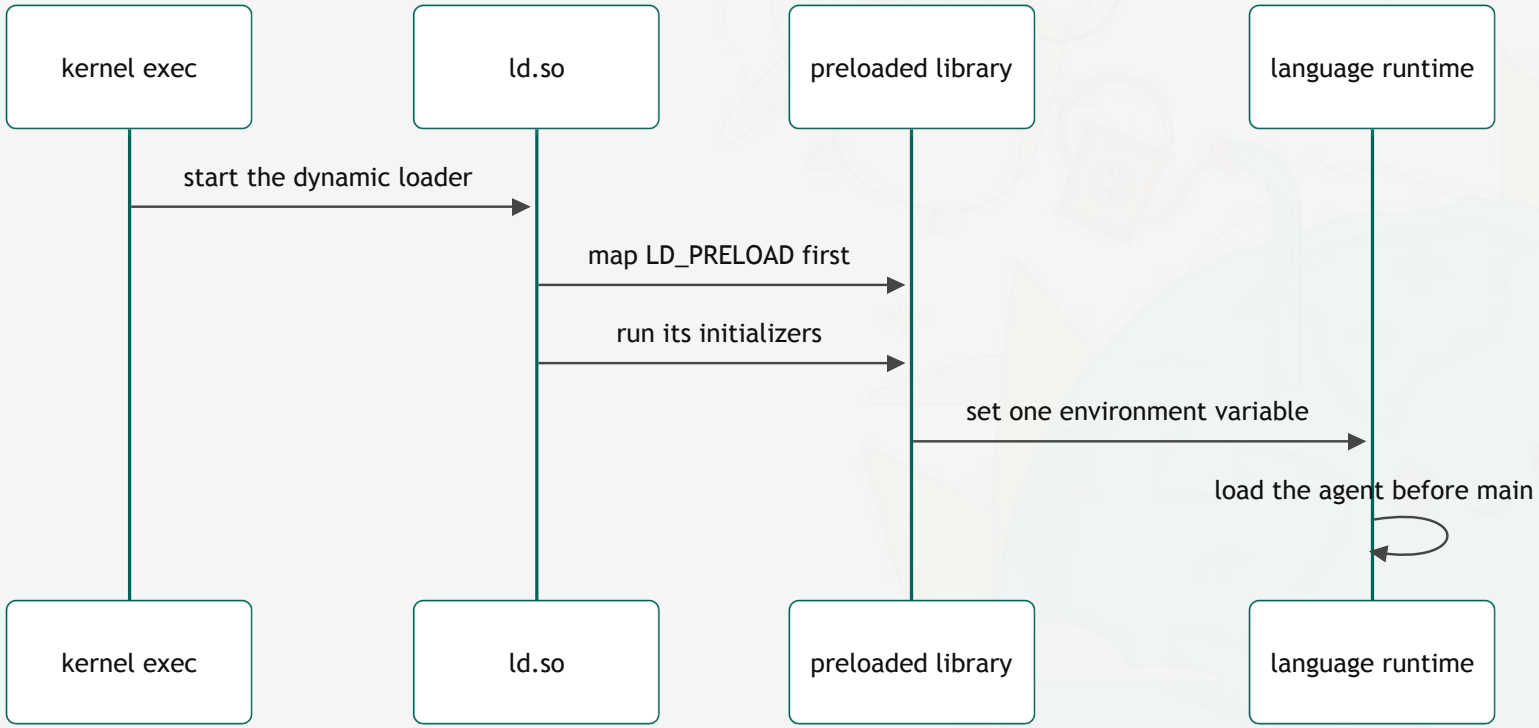
Hooking into the Go Toolchain • Internals for Interns

Process start: the injector



The SDK supplies the signals. The Go hook shown here is in development.

How process-start injection works



ld.so(8): preloaded objects load "before all others" • System V ABI: shared-object initializers run "before the executable file gains control"

Runtime hooks that injectors use

RUNTIME	THE HOOK THE INJECTOR SETS
Java	`JAVA_TOOL_OPTIONS=-javaagent:…`
Node.js	`NODE_OPTIONS=--require …`
.NET	`CLR profiler and startup-hook variables`
Python	`sitecustomize` on `PYTHONPATH`
Ruby	`RUBYOPT=-r …`
Go	**none**

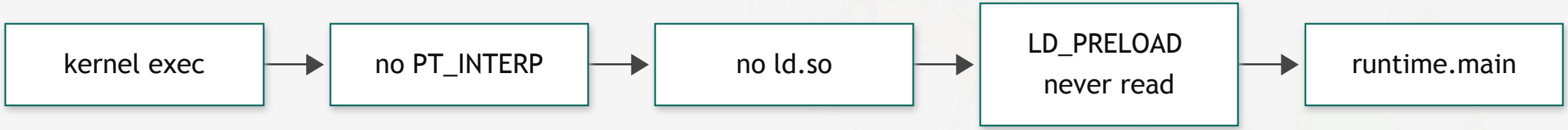
OpenTelemetry injector: Java, Node.js, .NET, Ruby, Python • Datadog SSI: Java, Python, Ruby, Node.js, .NET, PHP

Runtime injection is a narrower route

- Dynatrace OneAgent injects into eligible Go binaries, including static pure-Go ones started by a dynamic parent.
- The OpenTelemetry host injector ships Java, Node.js, .NET, Ruby, and Python agents. It does not inject Go.
- Datadog is working on a Go path for Single-Step Instrumentation.

Runtime injection works, but static Go binaries force vendor-specific workarounds.

A static Go binary has no loader



`CGO_ENABLED=0` gives no dynamic loader and no libc. Nothing reads the variable.

With musl, internal linking skips preloaded constructors (golang/go#28909).

`LD_PRELOAD` reaches only some Go binaries

```
$
LD_PRELOAD=./tracer.so ./python-app
LD_PRELOAD=./tracer.so ./go-app

go build -ldflags="-linkmode=external"
```

BUILD	RESULT
Dynamic, including a default cgo build	Injection can work
`CGO_ENABLED=0` or static cgo	No dynamic-loader hook
Pure-Go PIE	Loader runs, no libc

We made it work

```
LD_PRELOAD=./libdd_go_hook.so \
DD_SERVICE=my-service \
DD_TRACE_DEBUG=1 \
./my-go-app
```

`DD_TRACE_DEBUG=1` shows span creation and trace-id extraction on stderr.

`IN DEVELOPMENT`

No rebuild.

No source change.

No kernel version gate.

One line...



OBI: observe from the kernel

Traces **emitted**	Metrics **emitted**
Logs **correlation only**	Profiles **not emitted**

No rebuild. Linux and privileges.

OBI v1.0.0 shipped

30 October. Three days ago



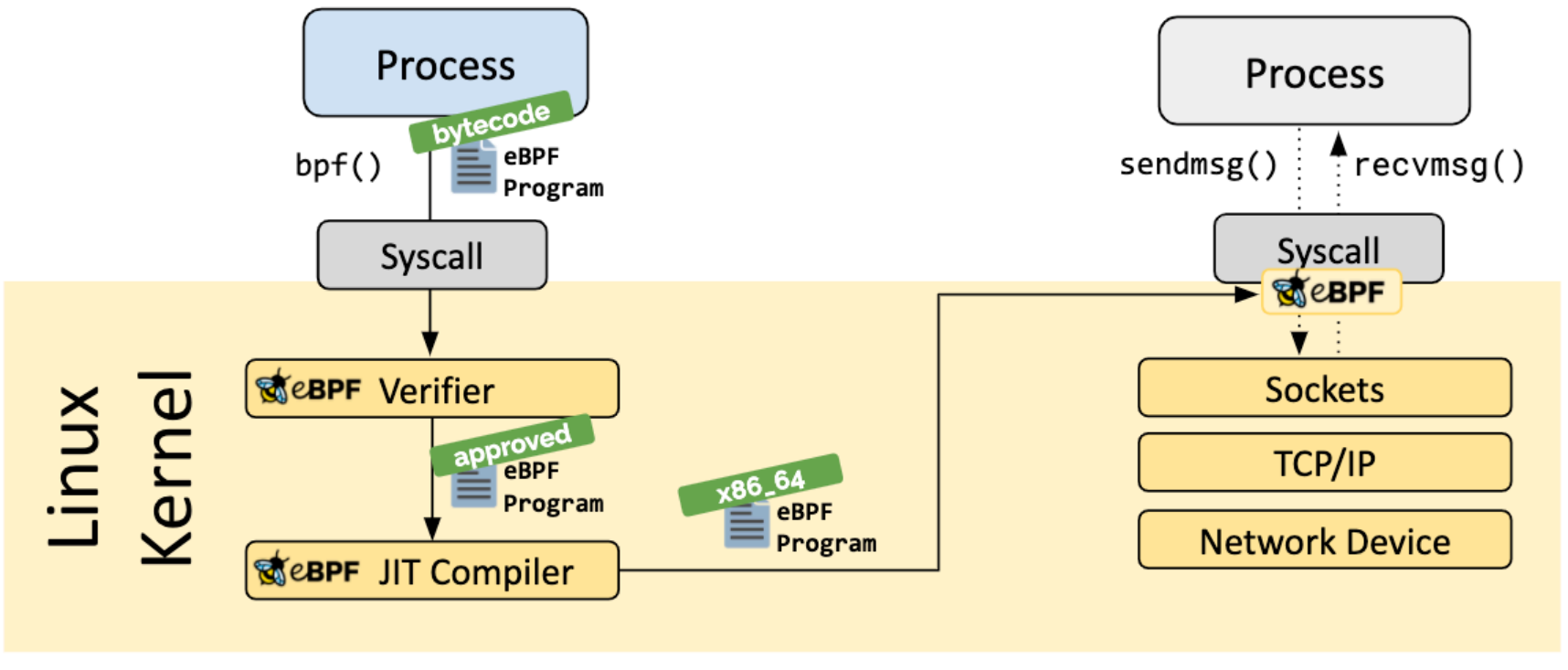
One DaemonSet, a whole node, no rebuild

OBI is the upstream successor to Grafana Beyla, donated to OpenTelemetry in 2025.

- No application rebuild.
- One DaemonSet can observe a mixed-language node.
- Network protocols provide broad coverage.
- Go-specific uprobes add library-level detail.

`V1.0.0, 30 OCT 2026: STABLE CORE` `GO MODULE API AND GRPC TELEMETRY NOT YET STABLE`

A checked program runs at an event



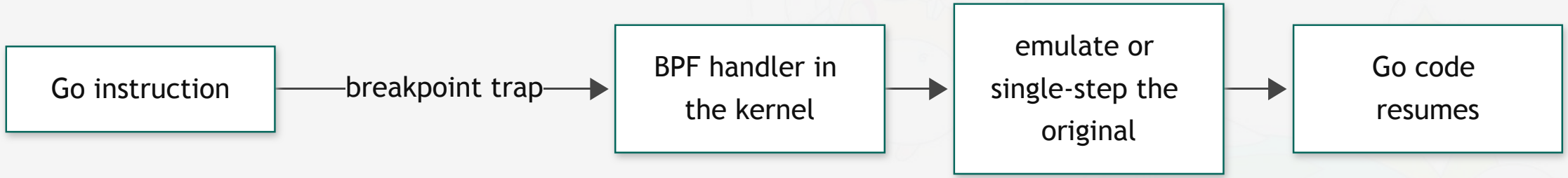
Load once. The verifier checks it. It runs when the event occurs.

Diagram: eBPF.io authors, "Loader", ebpf.io/what-is-ebpf, CC BY 4.0

For Go code, the hook is a uprobe

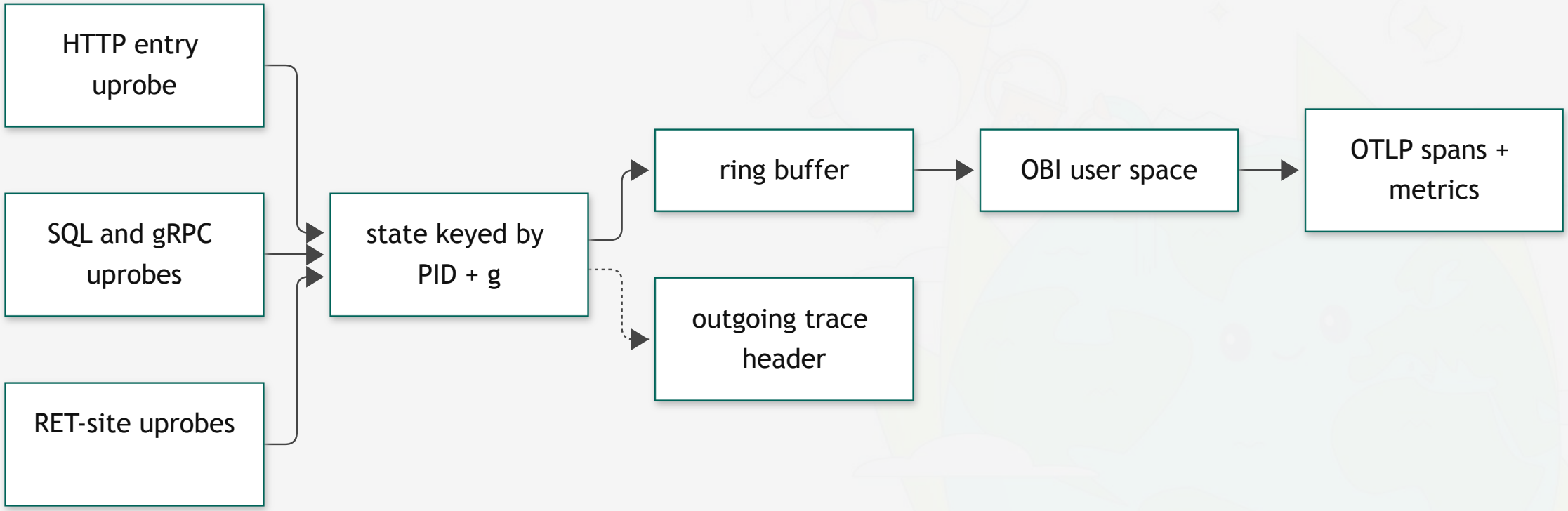
HOOK	OBSERVES
tracepoint	a named kernel event
kprobe	a kernel function or instruction
fentry / fexit	a kernel or BPF function, through a BPF trampoline
uprobe	**an instruction in a user-space binary**
uretprobe	a function return, through a return trampoline

A uprobe hit enters the kernel



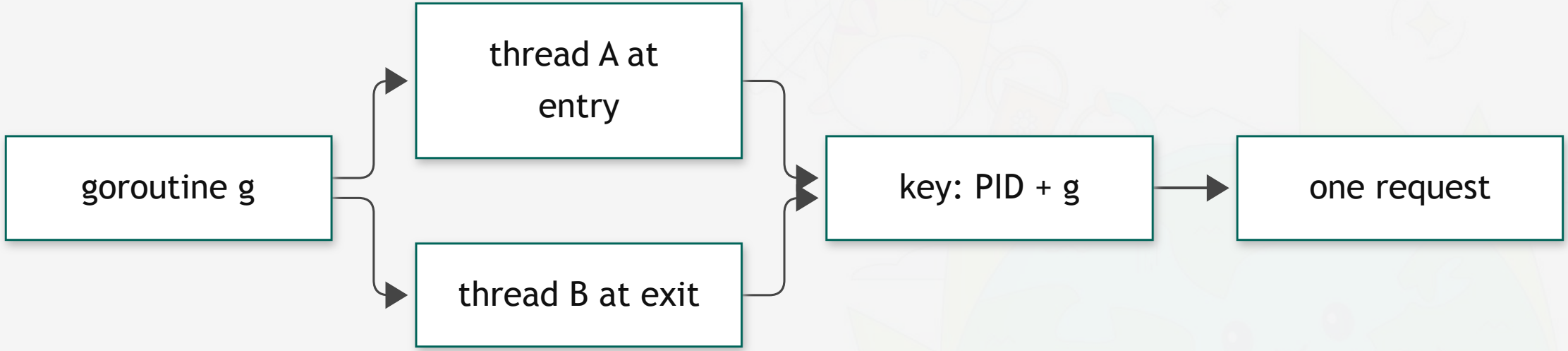
Conventional x86 path. The kernel patches the mapped code, not the file on disk. Every active hit has a cost.

OBI turns observations into spans



Probe sites give events. Correlation turns them into spans.

Read Go values, not C arguments



A goroutine is the identity. A thread is only the carrier.

amd64 ABIInternal (Go 1.17+): the first integer arguments go in `AX`, `BX`, `CX`, not in the C order.

Probe the RET, not the return address

Conventional uretprobe

Replaces the saved return address with a trampoline.

Go copies the stack when it grows. The runtime can reject the foreign frame.

`CRASH RISK IN GO`

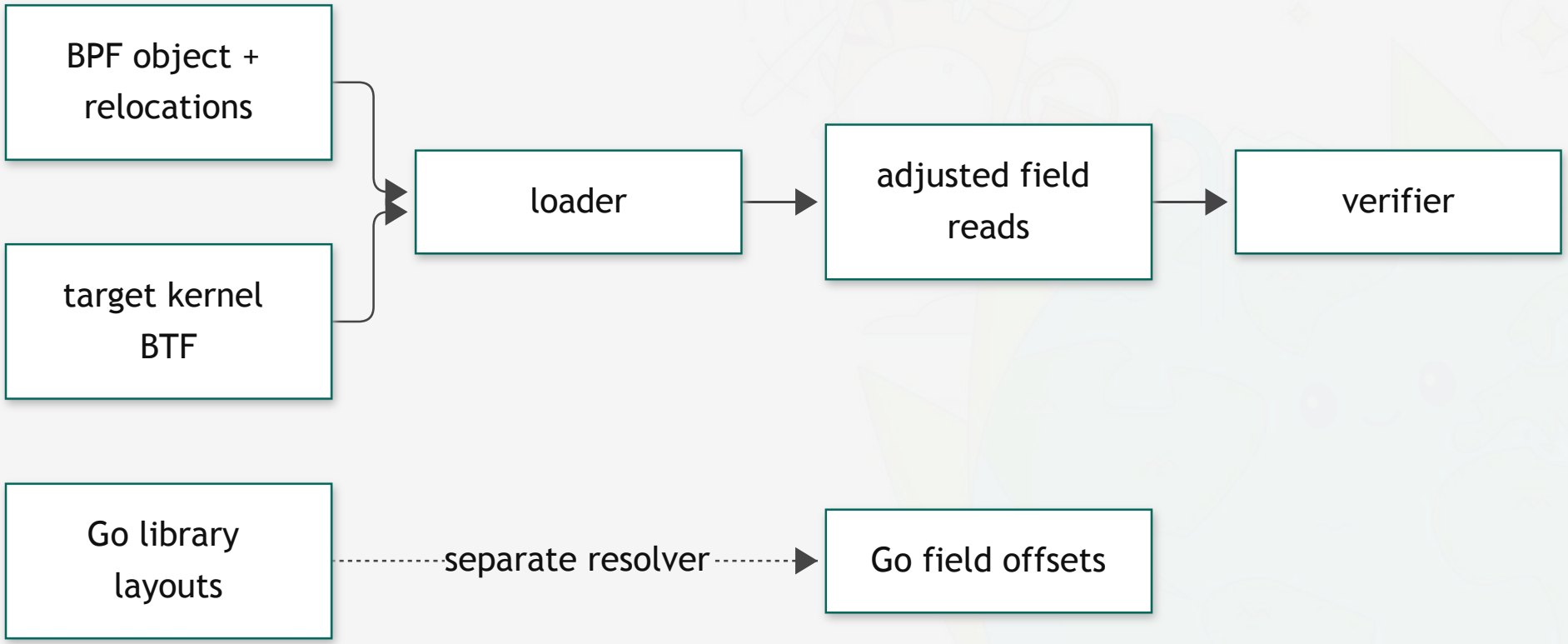
What OBI does

Finds every `RET` instruction in the function.

Attaches an ordinary uprobe at each one.

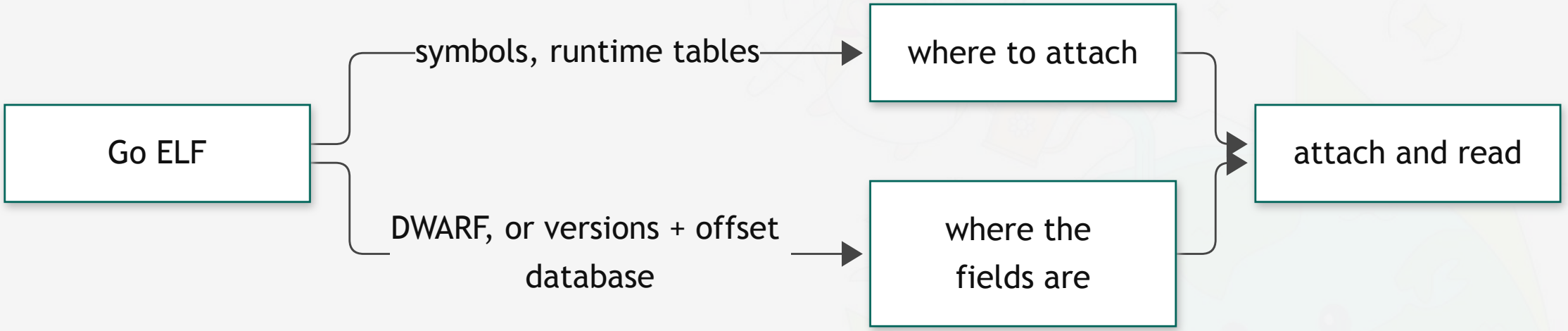
The caller address stays untouched.

CO-RE adapts kernel layouts, not Go layouts



BTF describes the kernel's types. CO-RE fixes field reads at load time.

Find the instruction and the field



A function address is not a type layout.

No source change. No rebuild. No rollout...



Go library coverage: 13 documented baselines

- `net/http`
- `golang.org/x/net/http2`
- `gorilla/mux`
- `gin-gonic/gin`
- `google.golang.org/grpc`
- `net/rpc/jsonrpc`
- `database/sql`
- `go-sql-driver/mysql`
- `lib/pq`
- `redis/go-redis/v9`
- `segmentio/kafka-go`
- `IBM/sarama`
- `mongo-driver` v1 and v2

`gin ⩾ v1.6.0, ≠ v1.7.5` • `redis/go-redis/v9 ⩾ v9.0.0` • OBI v1.0.0 support matrix

OBI emits traces and metrics. Logs get correlation.

Native signals

Traces

HTTP, gRPC, database, messaging

Metrics

RED and runtime metrics

Log correlation

Adds `trace_id` and `span_id` to selected **JSON and plain-text logs** while a span is active.

Not a log exporter. Linux 6.0+, no kernel lockdown.

OpenTelemetry: OBI trace-log correlation • OBI v1.0.0 telemetry contract

OBI needs Linux, BTF, and kernel 5.8 or a backport

REQUIREMENT	OBI
Operating system	Linux only
Architecture	`amd64`, `arm64`
Kernel	5.8+, or documented RHEL 4.18+ backports
Kernel metadata	BTF
Deployment	Host process, sidecar, or DaemonSet
macOS / Windows	Linux VM or remote cluster

Privileges depend on the mode

MODE	ADDITIONAL ACCESS
Network flow capture	`CAP_BPF`, `CAP_NET_RAW`
Application observability	`CAP_PERFMON`, `CAP_NET_RAW`, process and ELF access
Context propagation	`CAP_NET_ADMIN`
Go library propagation	`CAP_SYS_ADMIN` on kernel 5.10+

`kernel.perf_event_paranoid`, Secure Boot, and kernel lockdown can change what works.

What actually works without source changes

OBI handles

- Service boundaries
- HTTP and gRPC spans
- RED and runtime metrics
- Supported library operations
- Log correlation

Application code still owns

- Business events
- Domain-specific attributes
- Arbitrary internal functions
- Unsupported libraries
- Custom sampling decisions

Go spans you write with `otel.Tracer` and no SDK: OBI can export them through the Auto SDK. It needs `CAP_SYS_ADMIN`.

But it still has a cost...





zero-touch means zero cost... right?

Uprobes have real costs

Infrastructure cost

An attached uprobe crosses into the kernel. Heavy contention once hurt scalability across many CPUs.

Kernel 6.12 includes major RCU-protected hot-path improvements.

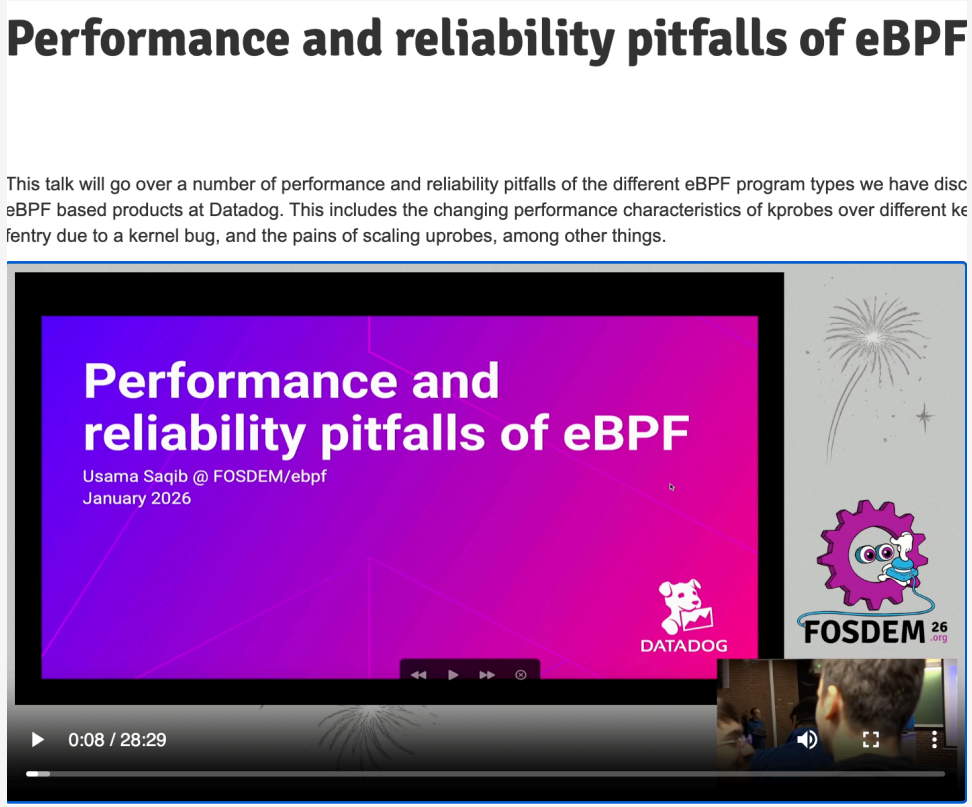
Usama Saqib, FOSDEM 2026 • Linux uprobe hot-path series • OBI PR #1851

Program and compatibility cost

Shared BPF state can serialize a concurrent application path.

Go runtime and library layout changes can invalidate offsets. Go 1.26 required an OBI symbol-resolution fix.

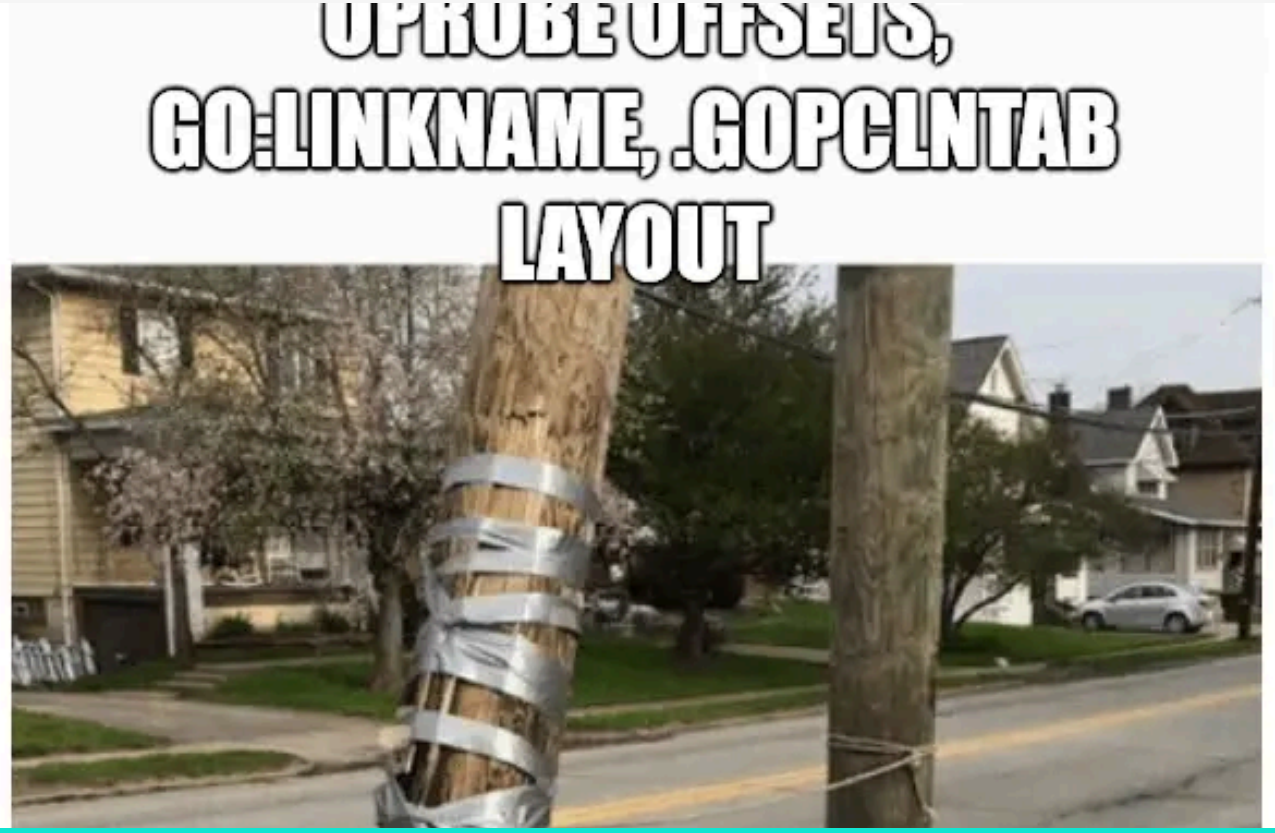
Usama Saqib: pitfalls of production eBPF



He covers the pitfalls found building production eBPF at Datadog: kprobe performance across kernel versions, a fentry kernel bug, and the pain of scaling uprobes.

Twenty-eight minutes, public, and it goes deeper than this section can.

Performance and reliability pitfalls of eBPF, FOSDEM 2026



somehow, it works

Profiles: observe CPU from the kernel

Traces
not emitted

Metrics
not the application signal

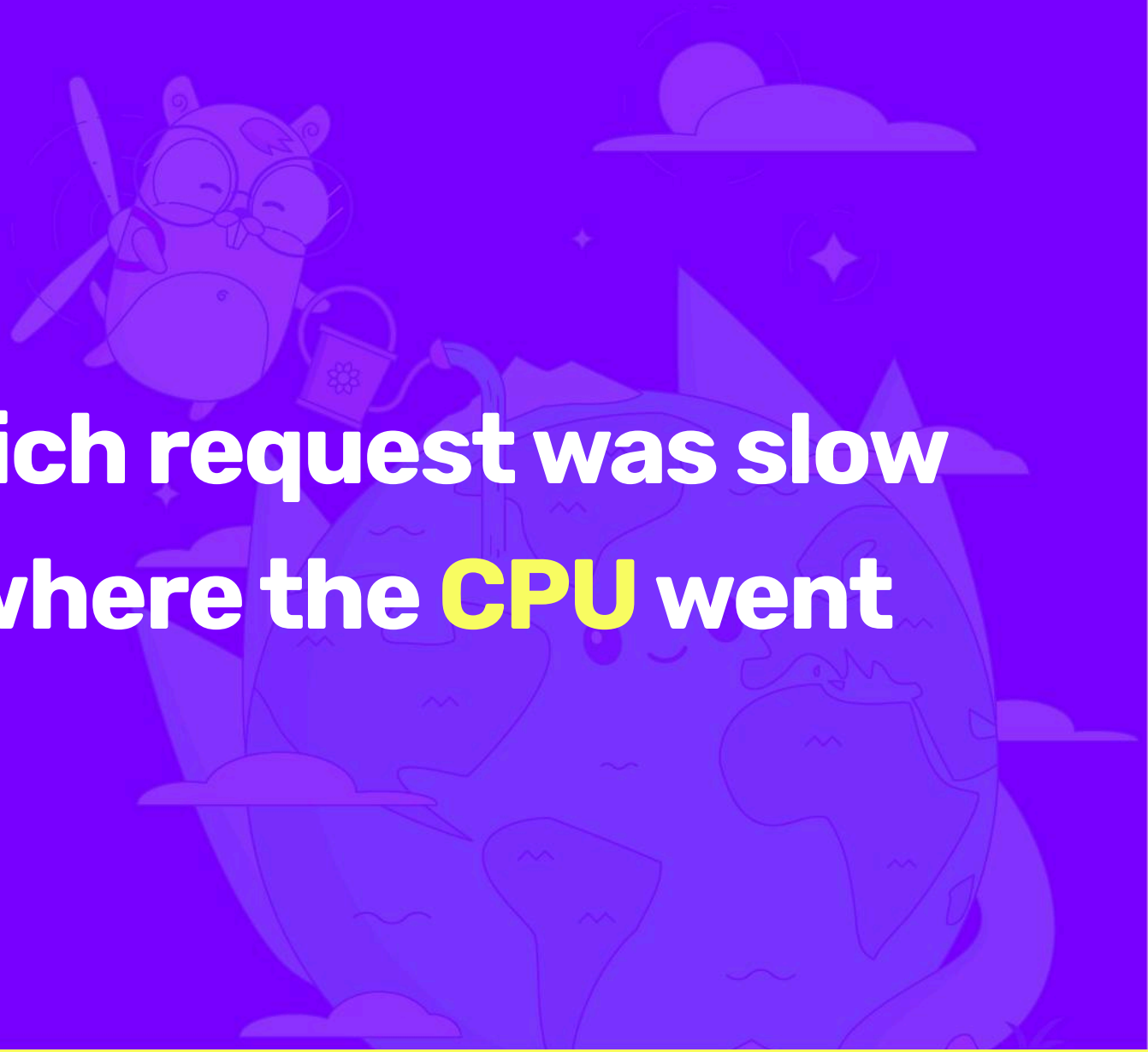
Logs
not the application signal

Profiles
CPU samples

A rebuild-free route to profiling. Linux node access required.

Traces tell you which request was slow

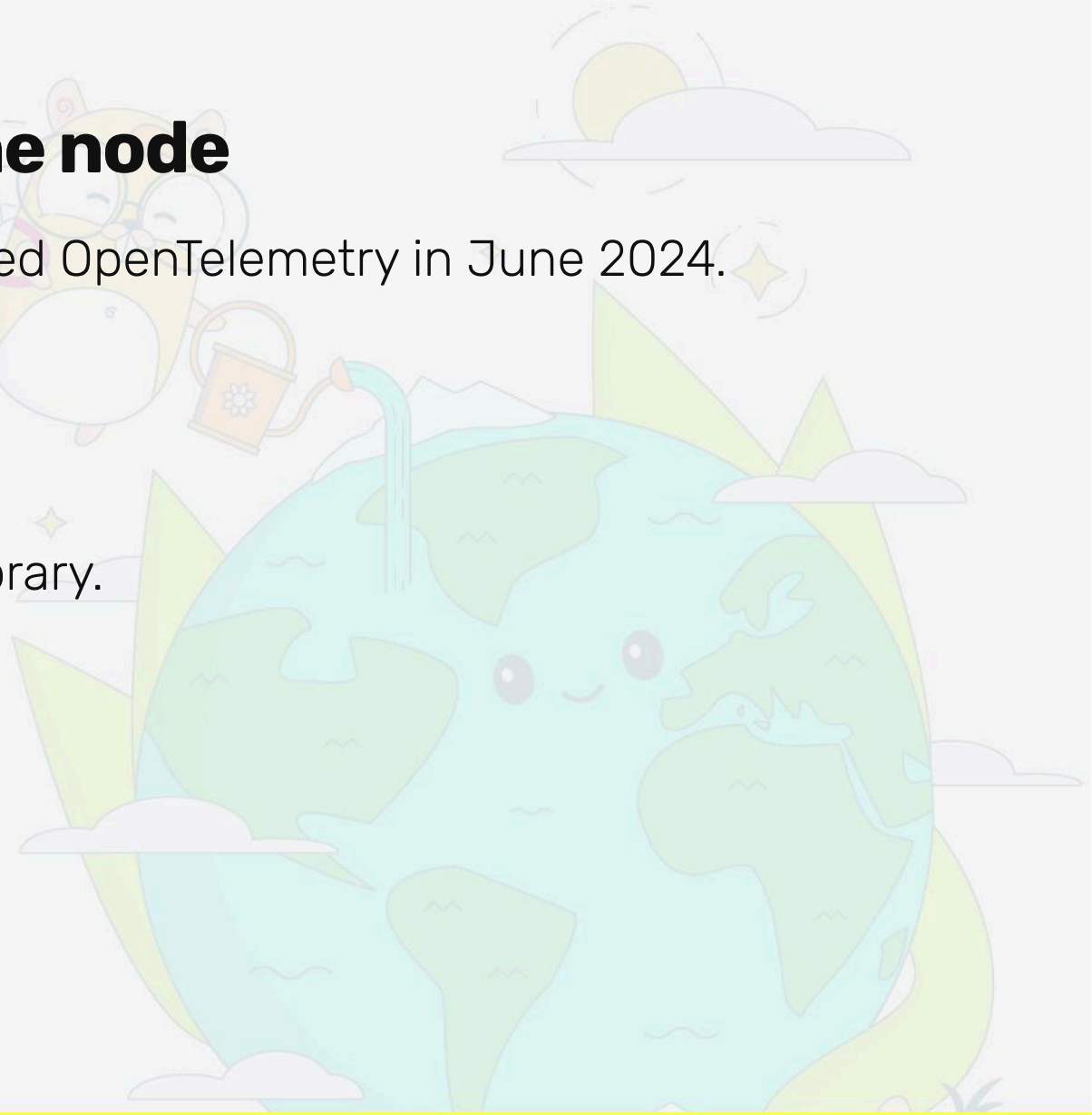
Profiles tell you where the CPU went



One profiler, every process on the node

Originated as Elastic Universal Profiling and joined OpenTelemetry in June 2024.

- Samples CPU from the kernel.
- Profiles every process on a node.
- Requires no application rebuild or injected library.
- Emits the OpenTelemetry Profiles signal.

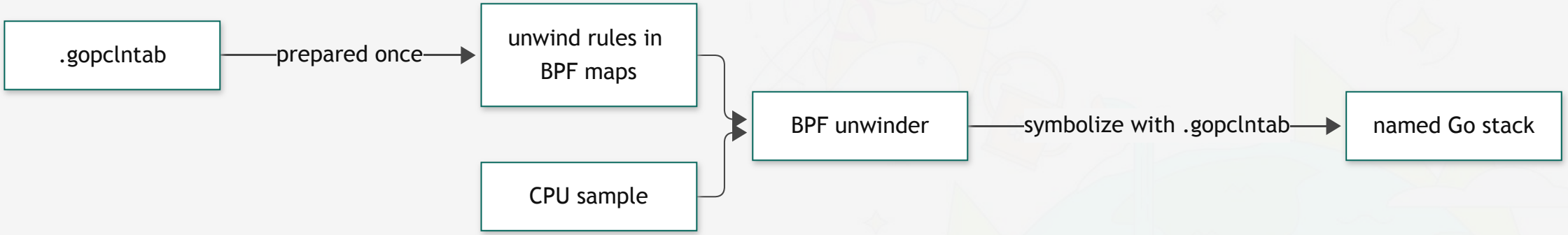


`PROFILES: ALPHA SPECIFICATION`

But how does it read a stripped binary?



Go metadata tells the profiler how to unwind



> "The information remains present even for fully static and stripped executables."
>
> OpenTelemetry eBPF Profiler documentation

Zero-touch profiling still has a deployment cost

APPLICATION	PROFILER AGENT
No source changes	`Linux node access`
No rebuild	`CAP_BPF` + `CAP_PERFMON` or root
No in-process agent	`DaemonSet / Collector distribution`
Stripped binaries supported	`CPU profiling is the confirmed core signal`

The signal is profiles. Choose the profiler by its coverage and deployment cost.

Traces and profiles. Apart...





each one points at the other

Live debugging: probe a running process

Traces
longer-term goal

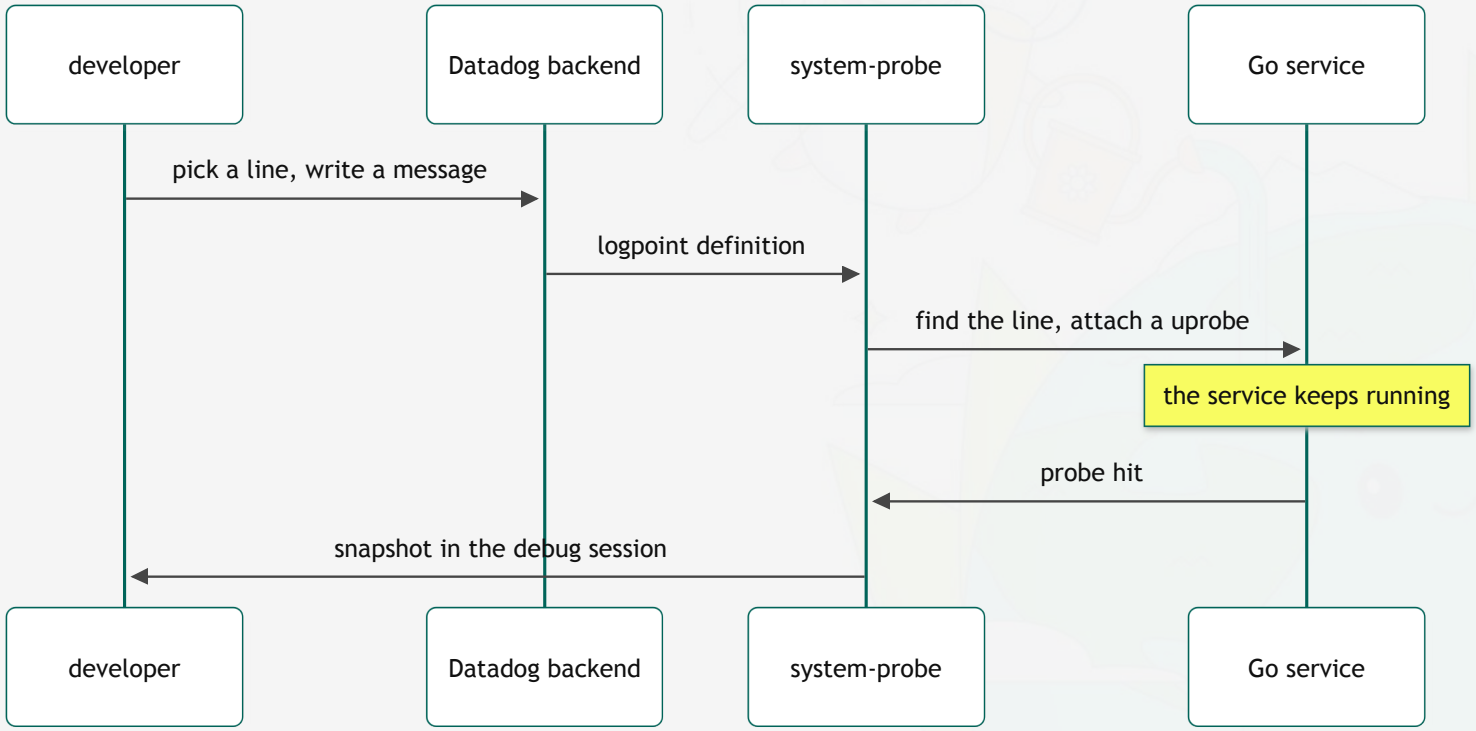
Metrics
goal

Logs
logpoints + snapshots

Profiles
not this logpoint

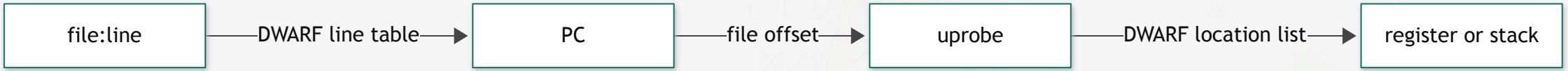
Ask for missing evidence while the service runs.

The probe goes to the running process



Simplified. Datadog Live Debugger: "non-breaking breakpoints that collect diagnostic data without pausing the application"

From a line to a machine address



The same uprobe as before. DWARF supplies the address and the variable locations.

Some variables are not there

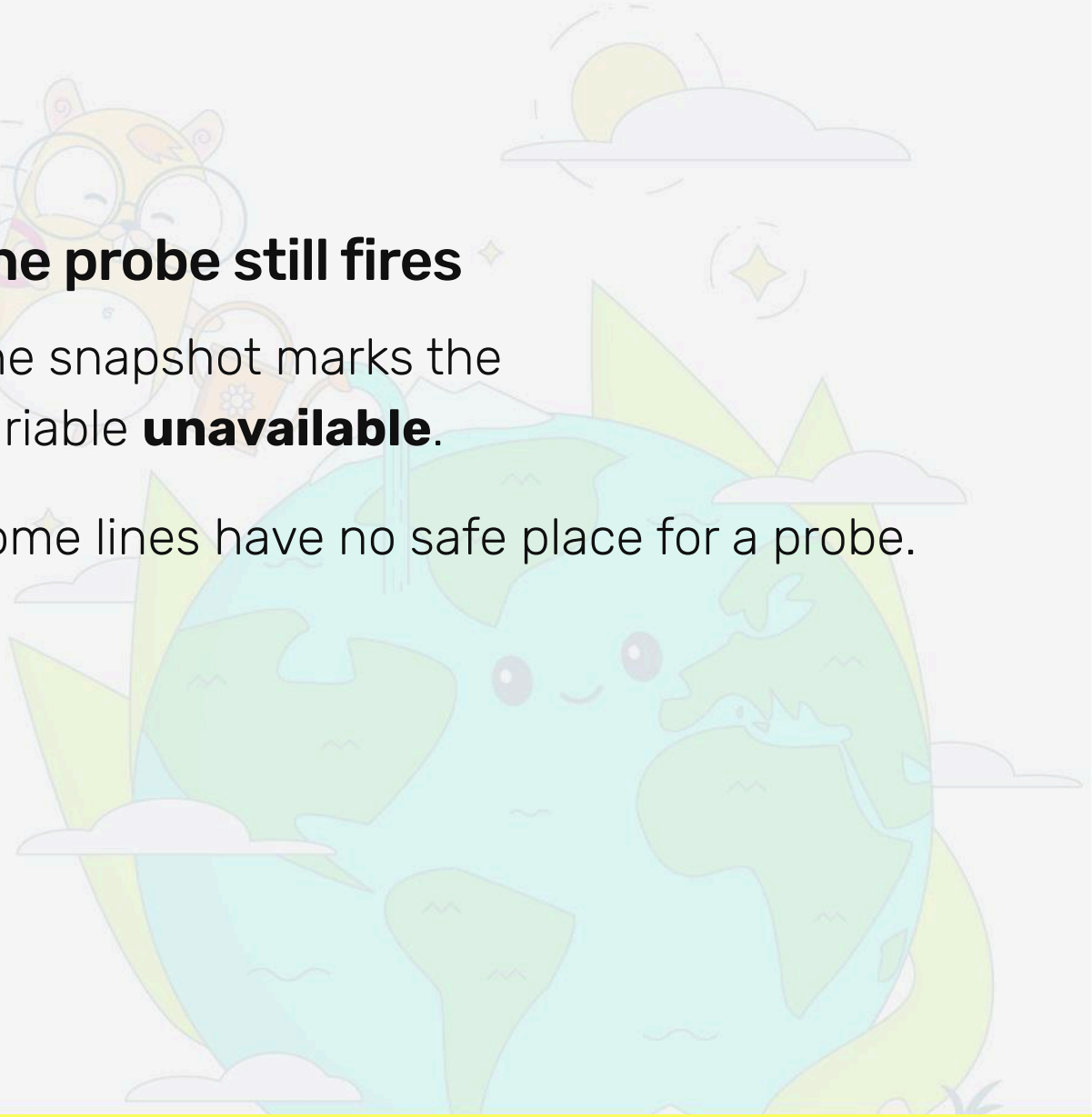
The compiler moved them

Inlining and register allocation leave some values with no location at that PC.

The probe still fires

The snapshot marks the variable **unavailable**.

Some lines have no safe place for a probe.



Bounded by design

CONTROL	PUBLIC LIMIT
Logpoint with variable capture	1 execution per second
Logpoint without capture	5000 per second per service instance
Lifetime	Expires with the debug session
Sensitive identifiers	Redacted before the data leaves the host

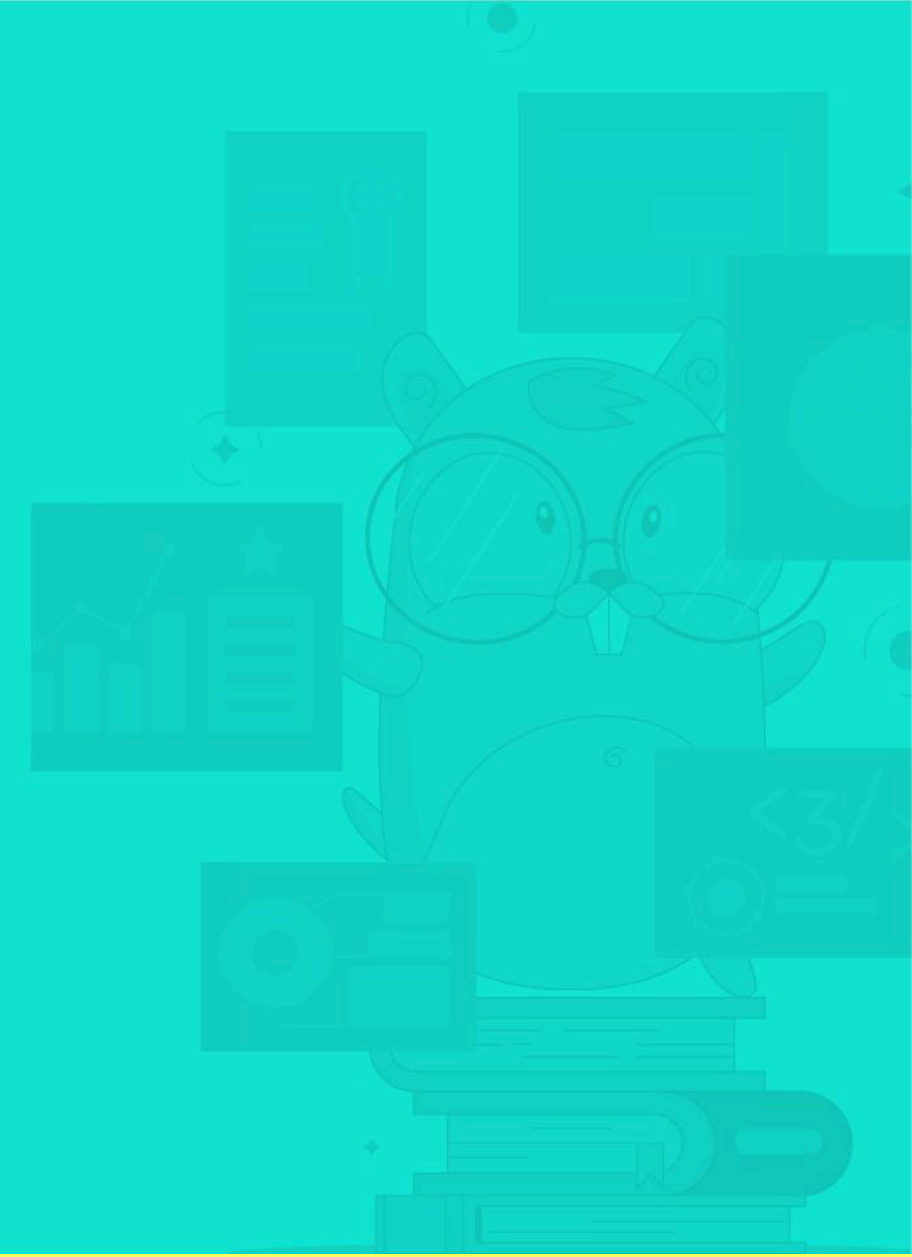
Go: Linux 5.17+, Datadog Agent 7.73+, Go SDK 2.9+ recommended • Live Debugger docs • Sensitive data scrubbing

Other ways, and their cost

	CODE CHANGE	STOPS THE PROGRAM	READS VARIABLES
Delve breakpoint	no	**yes**	yes
Flight recorder (Go 1.25)	**yes**	no	no, execution trace
Hand-written uprobe	no	no	by hand
Live Debugger logpoint	no	no	yes, bounded

07

Choose the signal and the layer



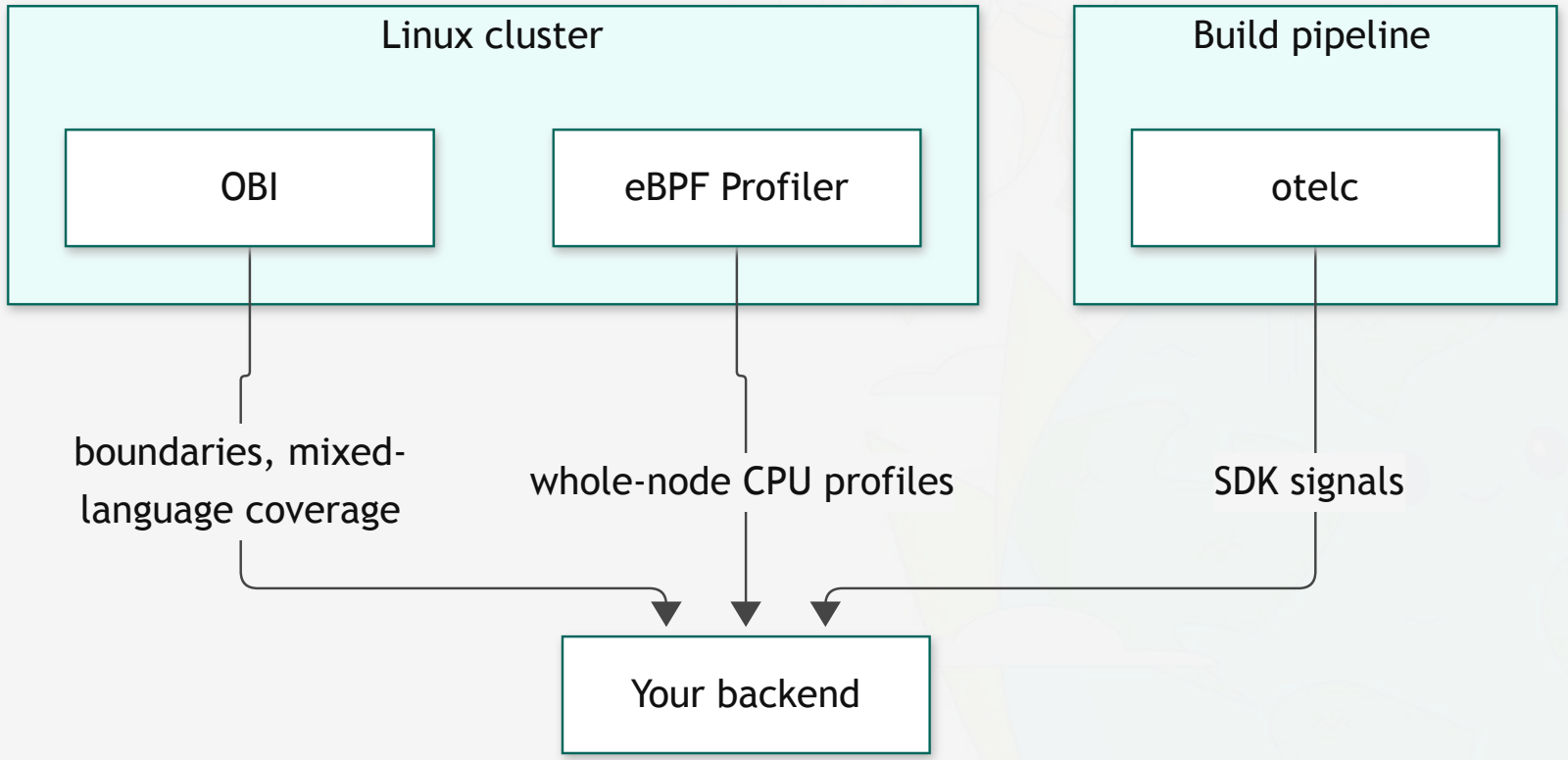
Similar signals, different coverage

APPROACH	TRACES	METRICS	LOGS	PROFILES
Build time	SDK	SDK	SDK	SDK
Injector (in development)	SDK	SDK	SDK	SDK
OBI	Yes	Yes	Correlation	No
eBPF Profiler	No	No	No	CPU
Live Debugger	Goal	Goal	Logpoints	No

When to reach for what

CONSTRAINT	PREFER
No rebuild window	OBI on supported Linux hosts
Non-Linux target	otelc
Mixed-language boundary coverage	OBI
Rich Go library semantics	otelc
No privileged runtime agent	otelc
Whole-node CPU profiles	eBPF Profiler
A missing variable in production	Live Debugger logpoint

A practical production combination



Add a layer when its signal earns the cost. Use live logpoints for missing evidence.

Asks for the Go team



I can find you in the hallway.

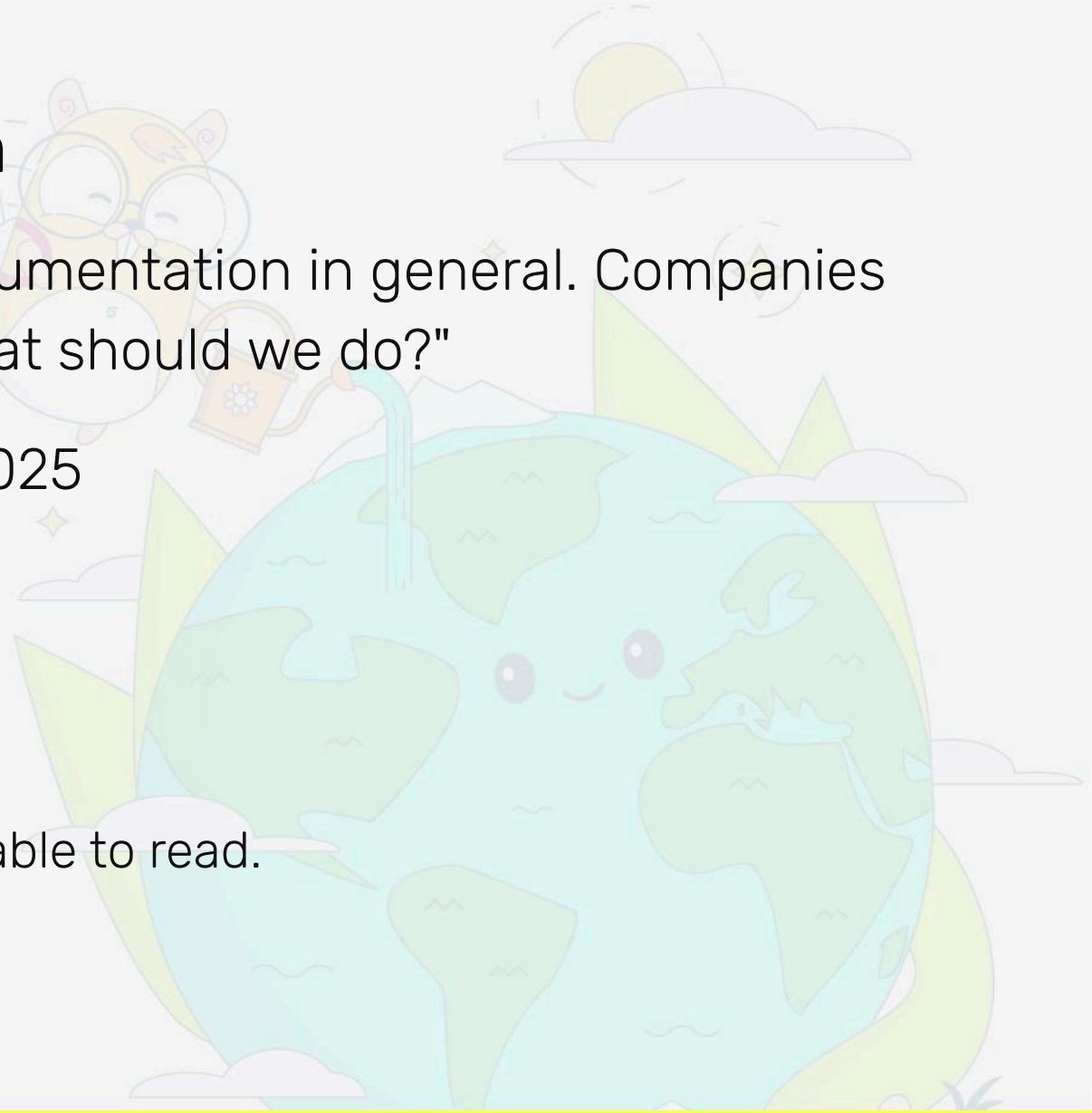
Pretty please

They already asked the question

> "We don't have a great way to add instrumentation in general. Companies will maintain patches to add hooks. What should we do?"
>
> Go runtime diagnostics sync, 31 July 2025

> "Maybe eBPF is the answer?"
>
> Go team, same meeting

Our answer: yes, if Go gives eBPF something stable to read.



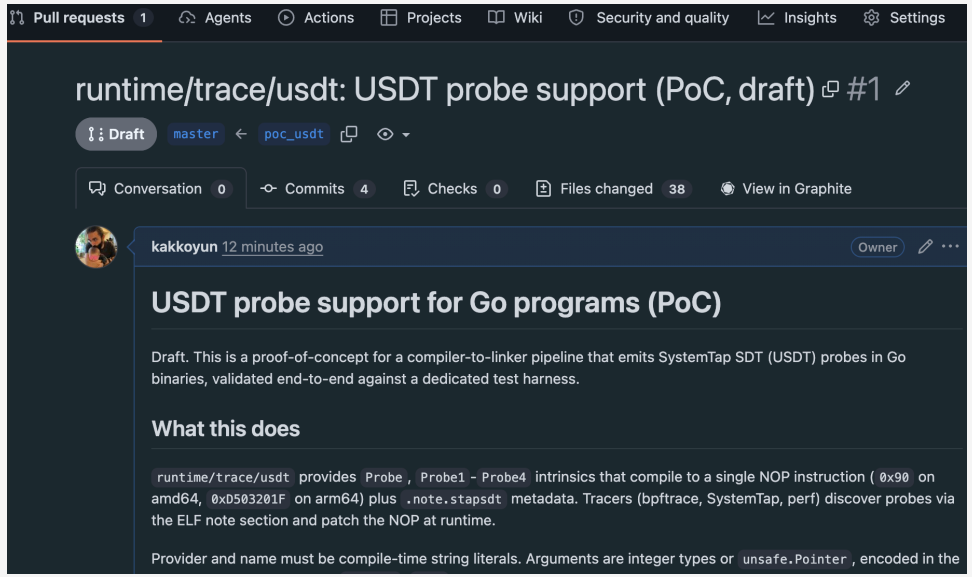
Ask 1: goroutine-local storage and context propagation

TODAY	ASK
Orchestrion adds `__dd_gls_v2` to `runtime.g`	One supported slot for trace context
otelc adds `otel_trace_context` to `runtime.g`	One layout that eBPF readers can trust
Each tool rewrites the runtime at build time	Rules for what a new goroutine inherits

Ask: give trace context a supported home. Two tools already build one.

Prior: golang/go#21355, #69478, #68785: "we've rejected anything that could be used as goroutine local storage"

Ask 2: USDT probes in Go



- A probe is a NOP plus an ELF note: a name and where each value lives.
- Go 1.27 emits none. Salp adds them through cgo.
- The eBPF profiler reads allocator USDT probes. The Go allocator has none.

Ask: let the toolchain emit SDT notes. Start with runtime sites.

`PROOF OF CONCEPT`

Ask 3: close the `-toolexec` gaps

- otelc and Orchestrion both build on `-toolexec`.
- A tool cannot add its own inputs to a package's build ID.
- golang/go#69887 lists the gaps, from build-ID influence to new build-graph edges.

Ask: close those gaps. A problem report, not a new build mode.

Ask 4: one supported start-up door

VARIABLE	VALUES AN AGENT MUST HANDLE
libc	`glibc, musl, none`
Link mode	`internal, external`
Build mode	`exe, PIE, static`
`plugin`	the exact toolchain and dependency versions

Ask: tell agent authors which door Go supports.

`CGO WITHOUT A C TOOLCHAIN: GOEXPERIMENT ACCEPTED (#81662)` `FIT FOR AGENTS STILL OPEN`

Four asks. I will find you in the hallway.

ASK	PRIOR ART
Goroutine-local storage and context propagation	#21355, #69478, #68785
USDT probes	#57175
`-toolexec` gaps	#69887
A start-up door	#28909, #12465
Bonus: stable runtime offsets	`moduledata`

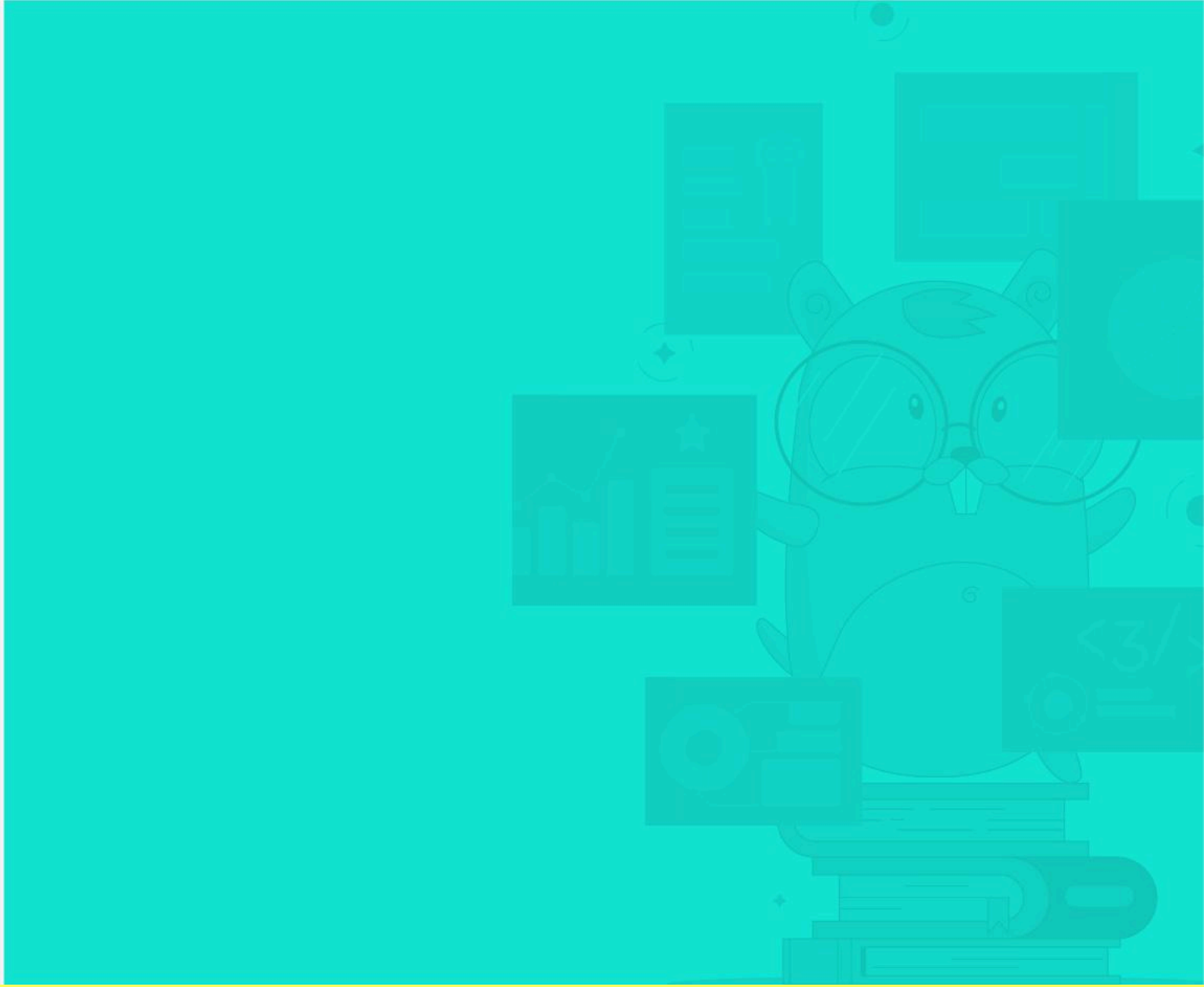
Where our tools go next

TOOL	TODAY	PLAN
Live Debugger	Logpoints and variable snapshots	Goal: metrics. Then traces, 12+ months
Go injector	In development	Go in Single-Step Instrumentation
eBPF Profiler	CPU profiles	Allocation profiling merged on `main`. Live heap in review

GOALS, NOT RELEASE COMMITMENTS

09

From question to evidence



The loop, without the redeploy

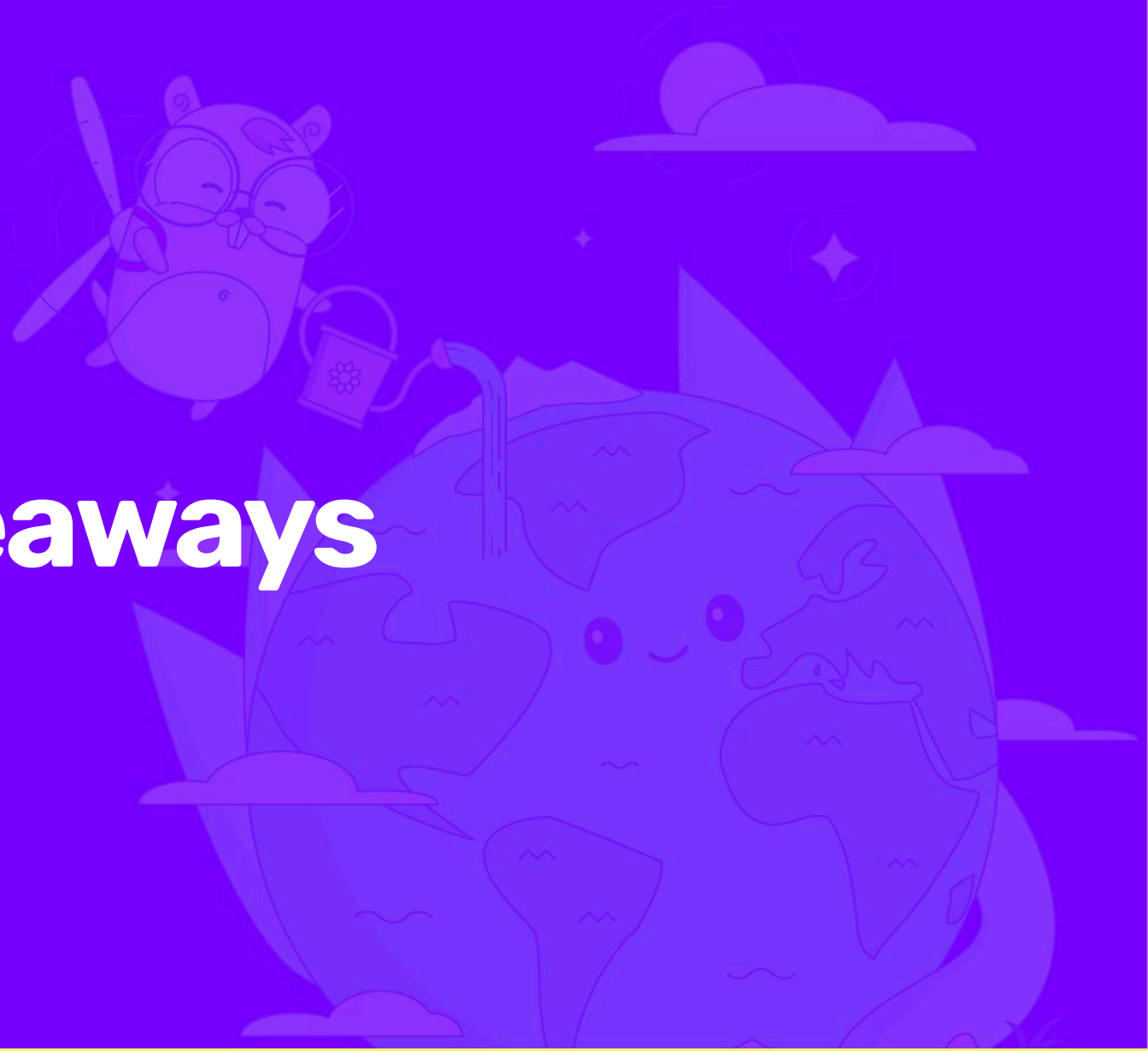
Describe the bug in words. The agent places the probe and reads *production evidence*.

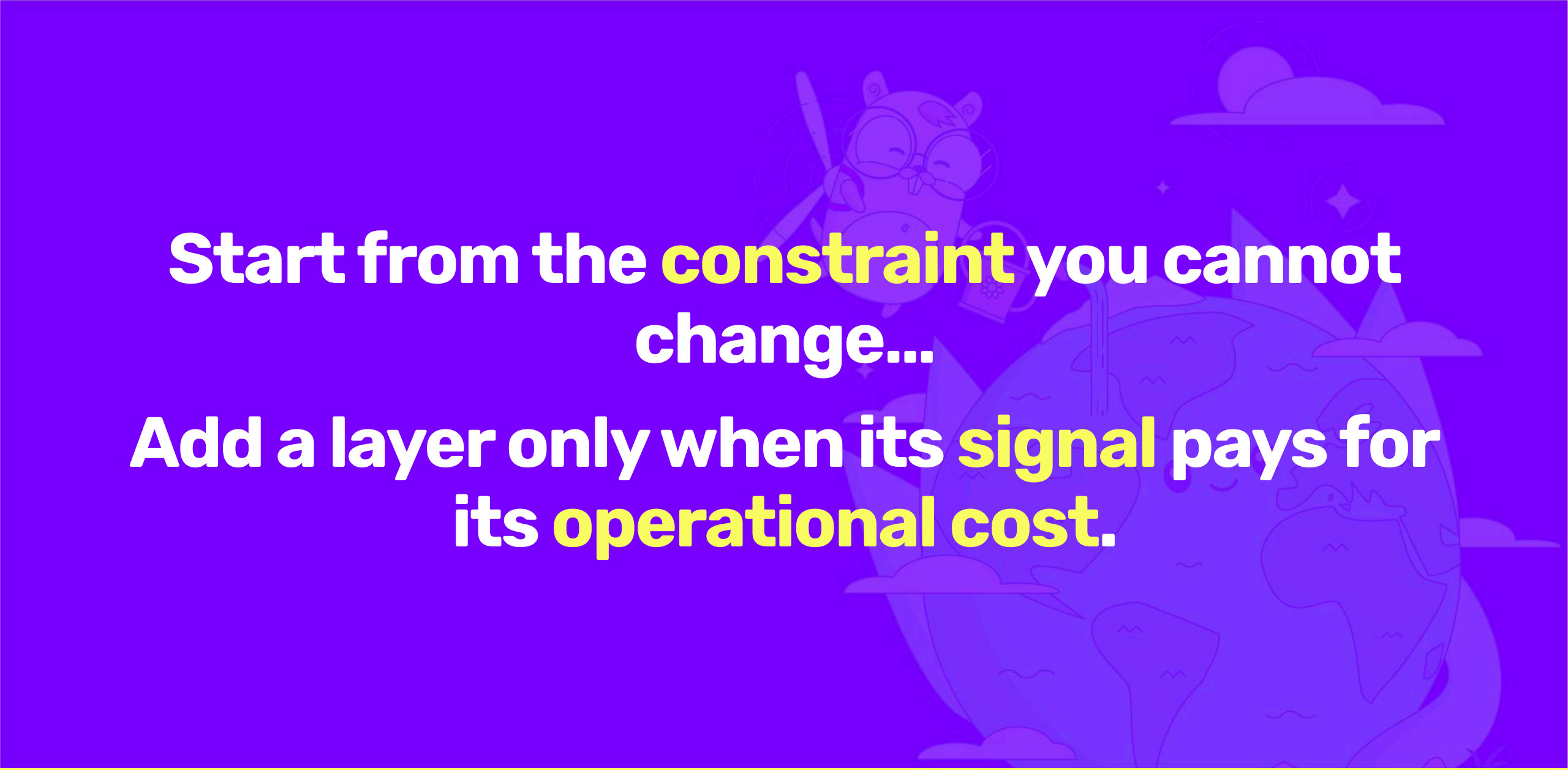
No source edit. No restart. No *redeploy*.

Bits Live Debugger does this today, in preview.

Same move as the opening joke, except now the agent has the *framework*.

Takeaways




Start from the constraint you cannot change...
Add a layer only when its signal pays for its operational cost.

Get involved in the SIGs



github.com/open-telemetry/community has every SIG's calendar, notes, and channels.

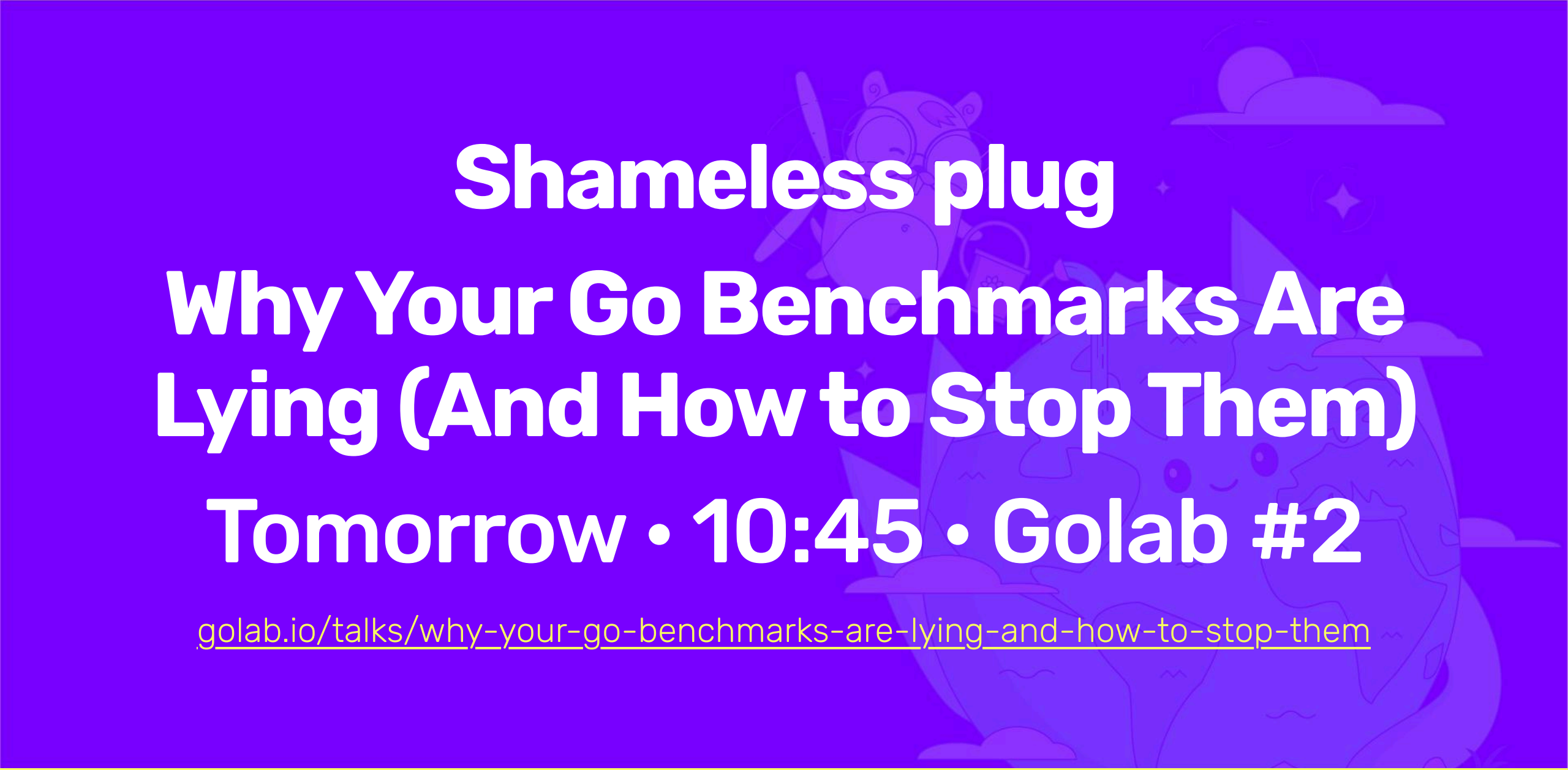
- Go Compile-Time Instrumentation (`otelc`)
- eBPF Instrumentation (OBI)
- Injector
- Profiling

Shameless plug

Why Your Go Benchmarks Are Lying (And How to Stop Them)

Tomorrow • 10:45 • Golab #2

golab.io/talks/why-your-go-benchmarks-are-lying-and-how-to-stop-them



Take it with you

Talk material

github.com/kakkoyun/golab-26

Both decks • research • earlier FOSDEM version

zeroins toolkit

github.com/kakkoyun/zeroins

`obi-integration` • `otelc-aspect`

`kubectl-obi` • `kubectl-profiler`



Scan for the talk repository

Grazie!

Find me in the hallway.

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 • Tools github.com/kakkoyun/zeroins

