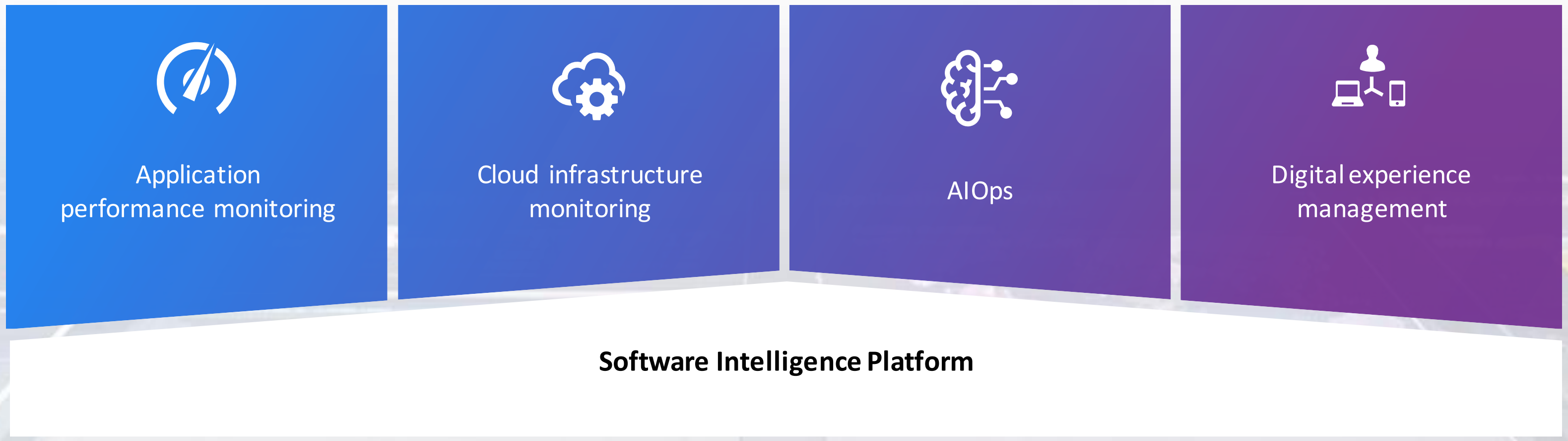
Your office could be here

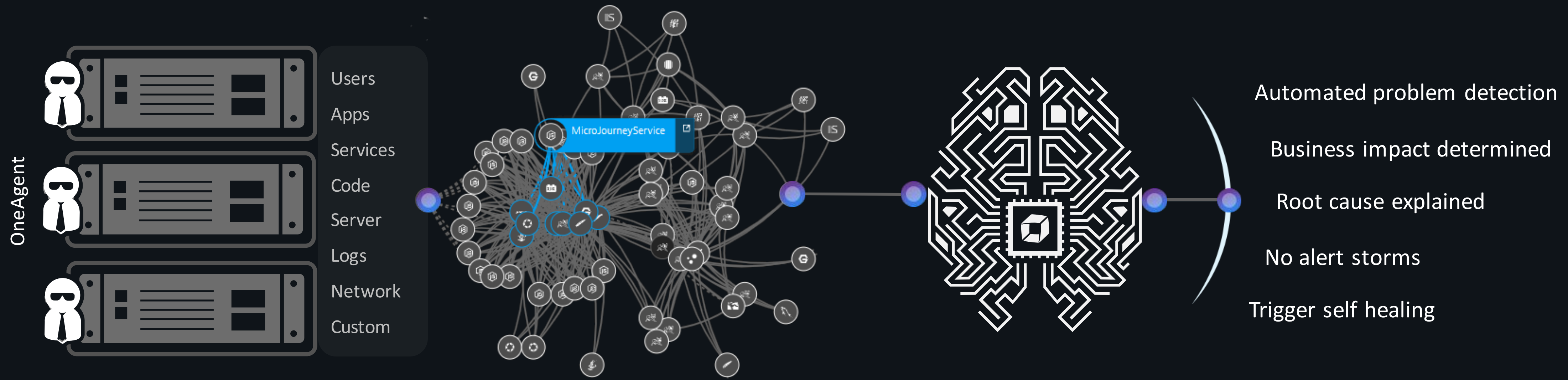
Software intelligence built for the enterprise cloud

Go beyond APM with the Dynatrace all-in-one platform


Application performance monitoring
Cloud infrastructure monitoring
AIOps
Digital experience management
Software Intelligence Platform

Better data makes Dynatrace A.I. and massive automation possible

High fidelity data | Mapped end-to-end | Deterministic AI | Answers + Action



Completely automated

Dynatrace - Software intelligence built for the enterprise cloud

Application performance management

Microservices & containers	Code-level	Database monitoring
Lifecycle analytics	Transaction tracing	Monolith and mainframe

Cloud infrastructure monitoring

Cloud monitoring	Virtualization monitoring	Network monitoring
Container monitoring	Server monitoring	Log monitoring

AIOps

Continuous auto-discovery	Anomaly detection	Prediction
Automatic topology	Root cause analysis	Third party data & metrics

Digital experience management

Real user monitoring	Mobile app monitoring	RUM for SaaS vendors
Session replay	Synthetic monitoring	Digital experience insights

All-in-one software intelligence platform.

The southern hemisphere of Go

About operational monitoring of Go applications

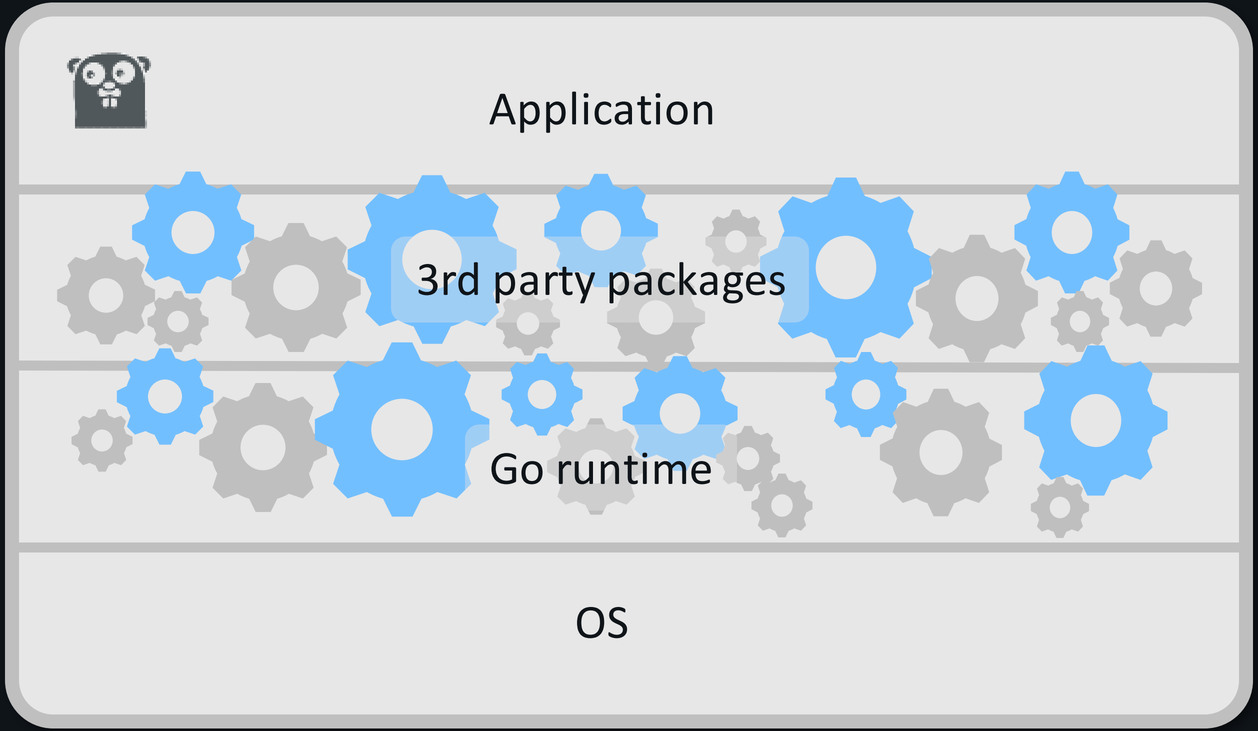
Dec 11th, 2019

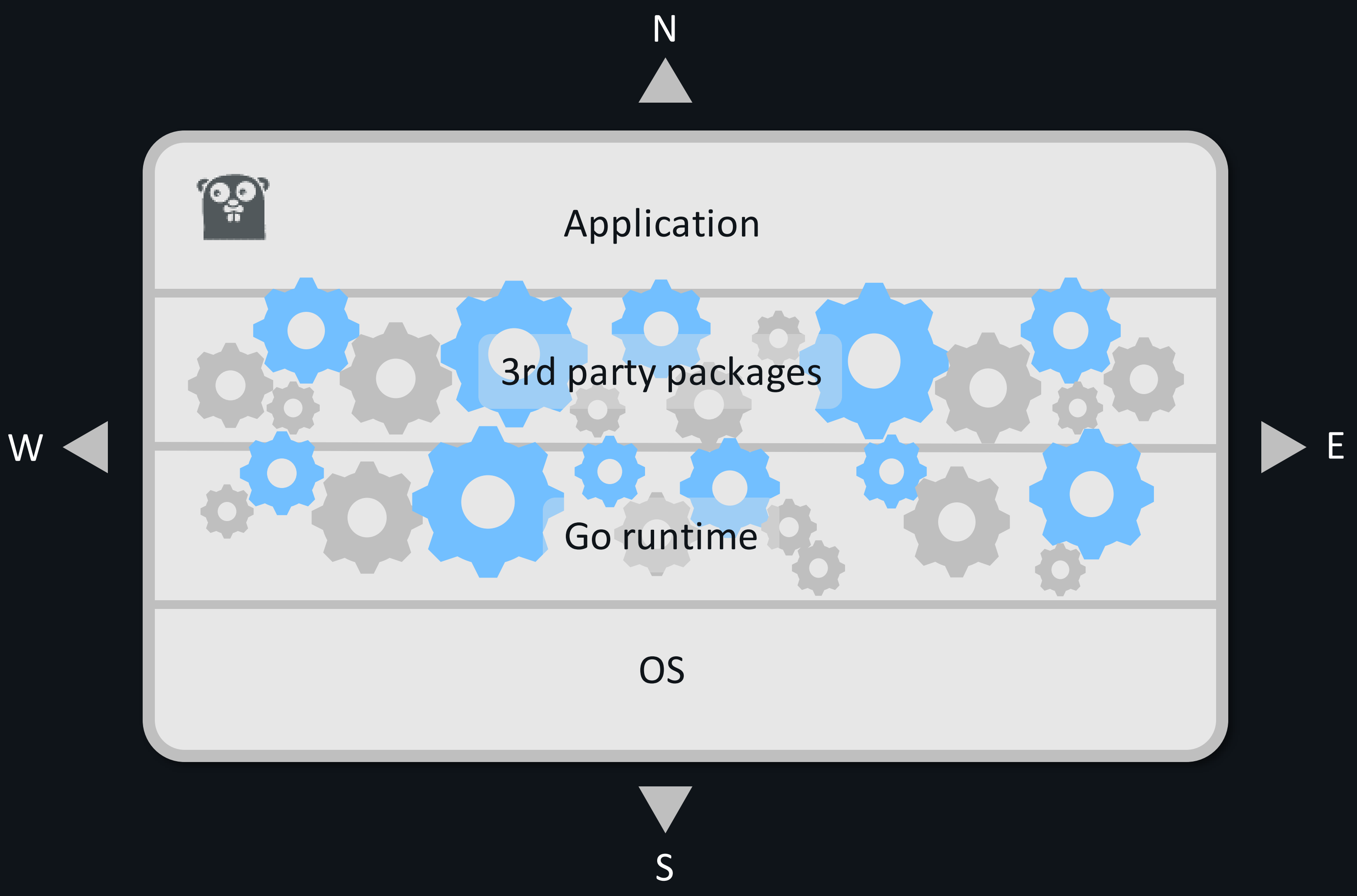
Michael Obermueller

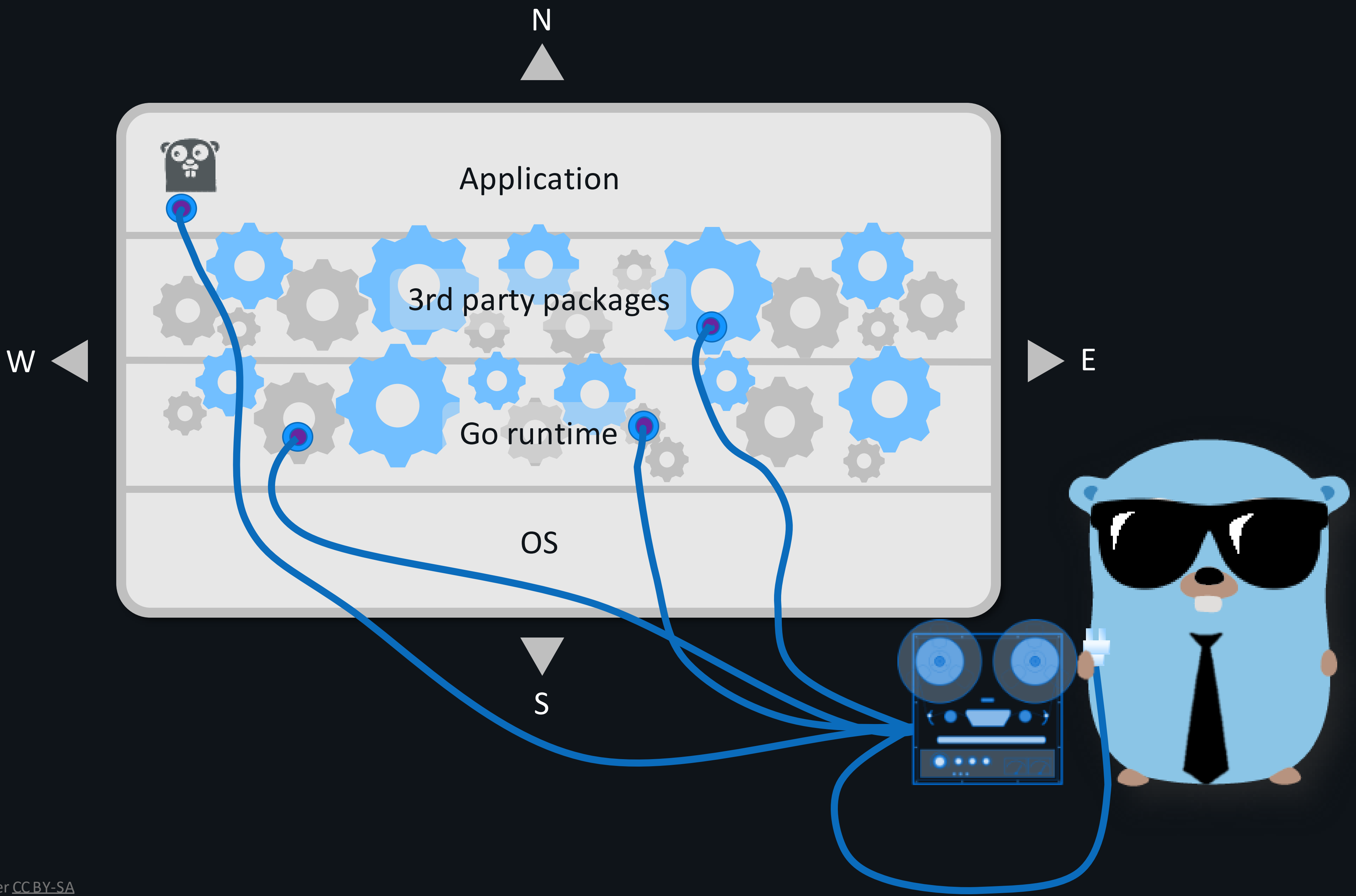
Gernot Reisinger

dynatrace

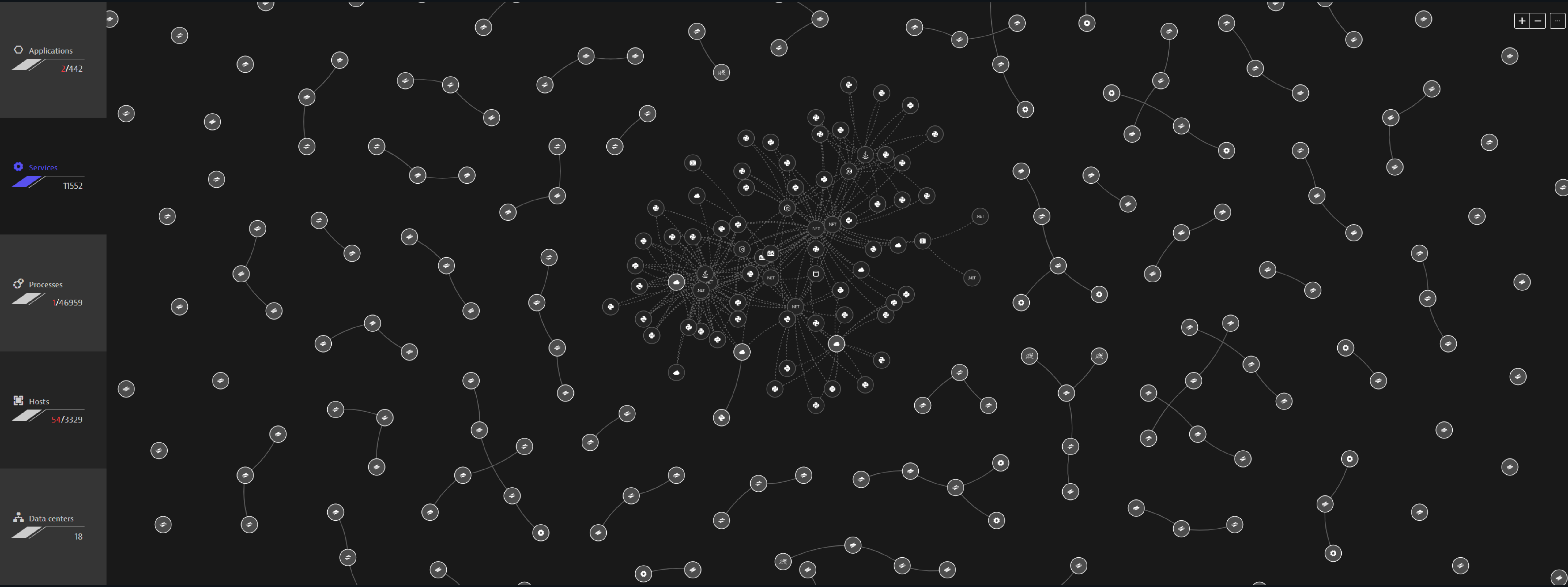
Application


Application
3rd party packages
Go runtime
OS

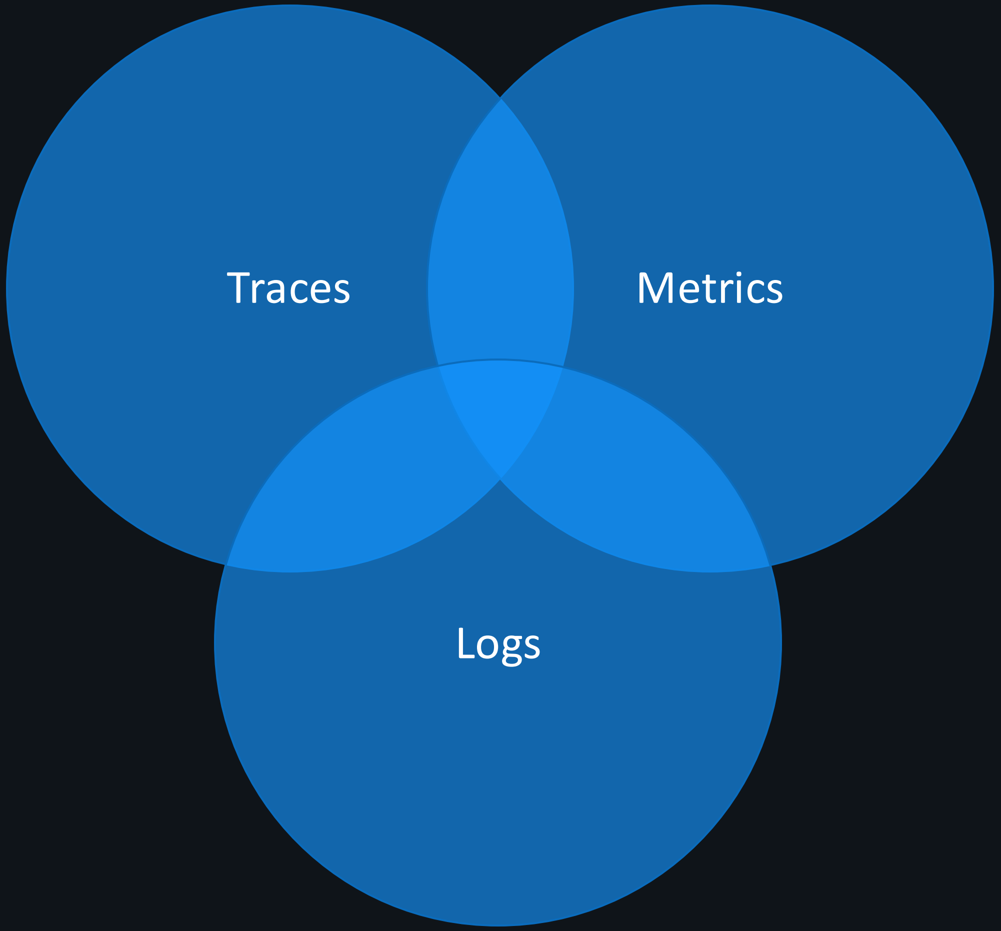

N
W
E
S
Application
3rd party packages
Go runtime
OS


N
Application
3rd party packages
W
E
Go runtime
OS
S

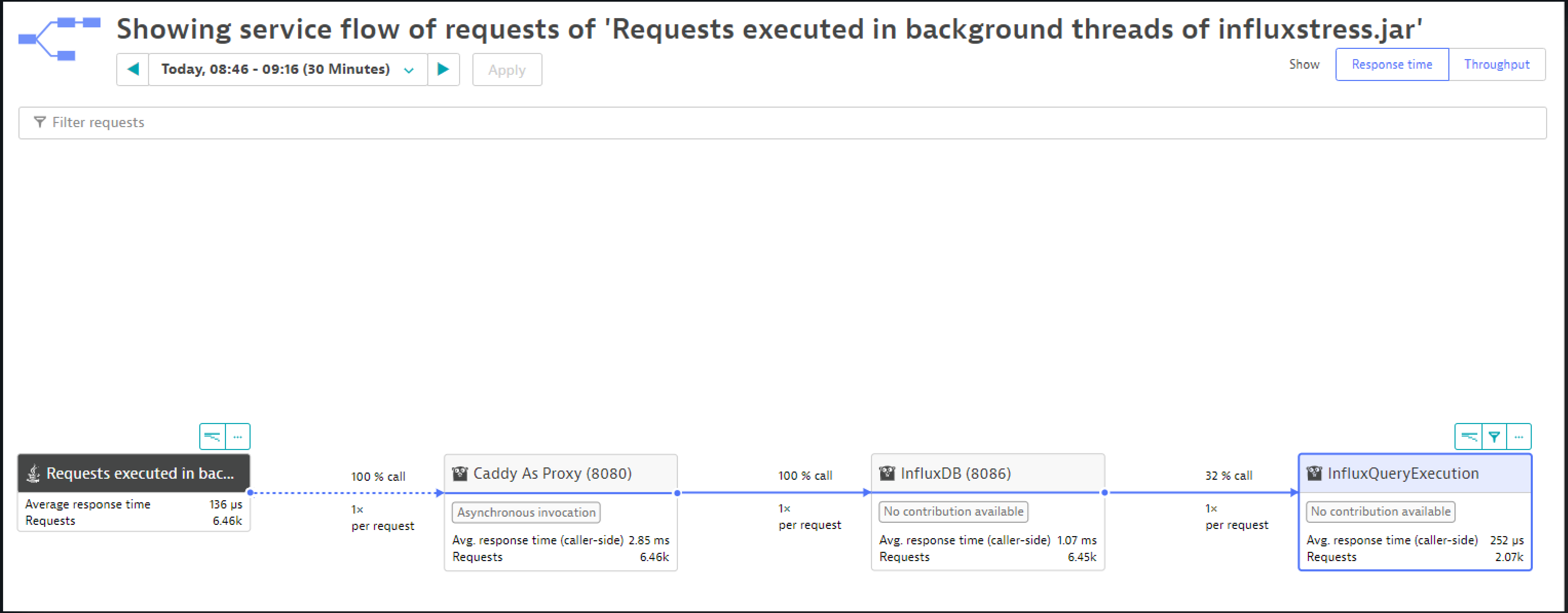
Operational monitoring


Applications
2/442
Services
11552
Processes
1/46959
Hosts
54/3329
Data centers
18
.NET

Observability


Traces
Metrics
Logs

Traces



Showing service flow of requests of 'Requests executed in background threads of influxstress.jar'

Today, 08:46 - 09:16 (30 Minutes)

Apply

Show Response time Throughput

Filter requests

Requests executed in bac...
Average response time 136 µs
Requests 6.46k

100 % call
1× per request

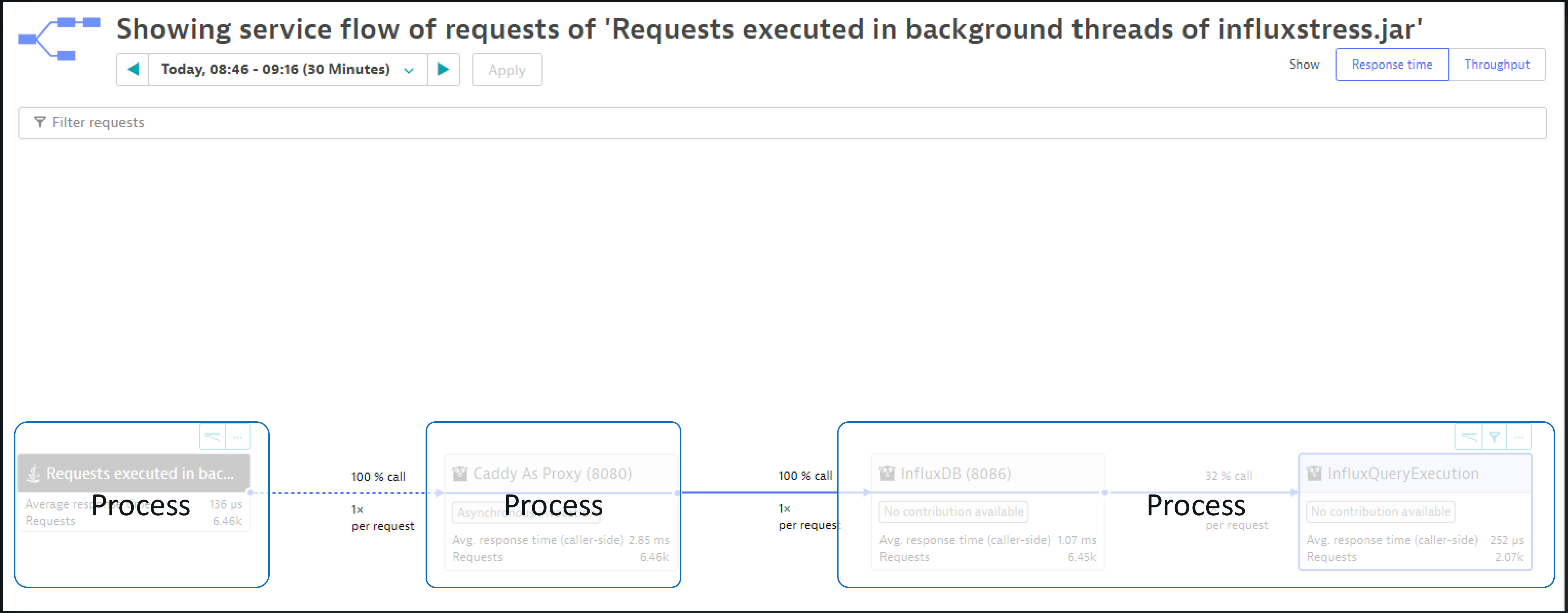
Caddy As Proxy (8080)
Asynchronous invocation
Avg. response time (caller-side) 2.85 ms
Requests 6.46k

100 % call
1× per request

InfluxDB (8086)
No contribution available
Avg. response time (caller-side) 1.07 ms
Requests 6.45k

32 % call
1× per request

InfluxQueryExecution
No contribution available
Avg. response time (caller-side) 252 µs
Requests 2.07k


Showing service flow of requests of 'Requests executed in background threads of influxstress.jar'

Today, 08:46 - 09:16 (30 Minutes)

Filter requests

Requests executed in bac... — Average response time 136 µs, Requests 6.46k

100 % call, 1× per request

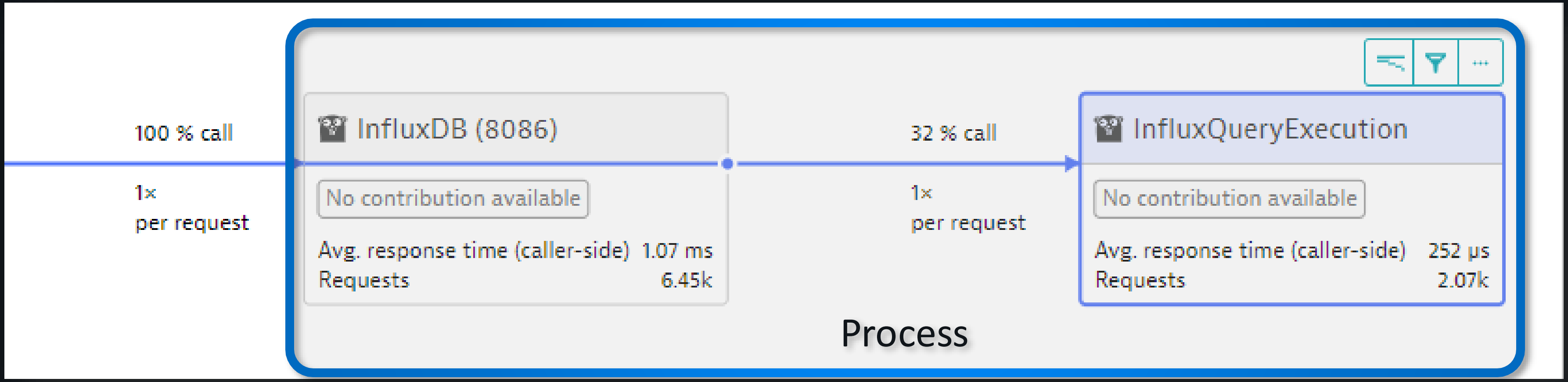
Caddy As Proxy (8080) — Asynchronous; Avg. response time (caller-side) 2.85 ms, Requests 6.46k

100 % call, 1× per request

InfluxDB (8086) — No contribution available; Avg. response time (caller-side) 1.07 ms, Requests 6.45k

32 % call, per request

InfluxQueryExecution — No contribution available; Avg. response time (caller-side) 252 µs, Requests 2.07k



100 % call

1×
per request

InfluxDB (8086)

No contribution available

Avg. response time (caller-side) 1.07 ms
Requests 6.45k

32 % call

1×
per request

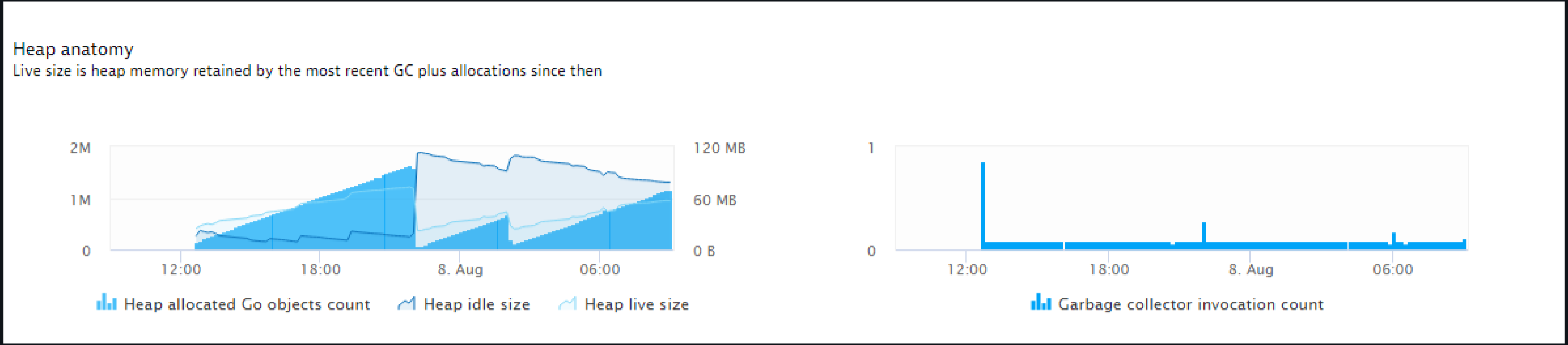
InfluxQueryExecution

No contribution available

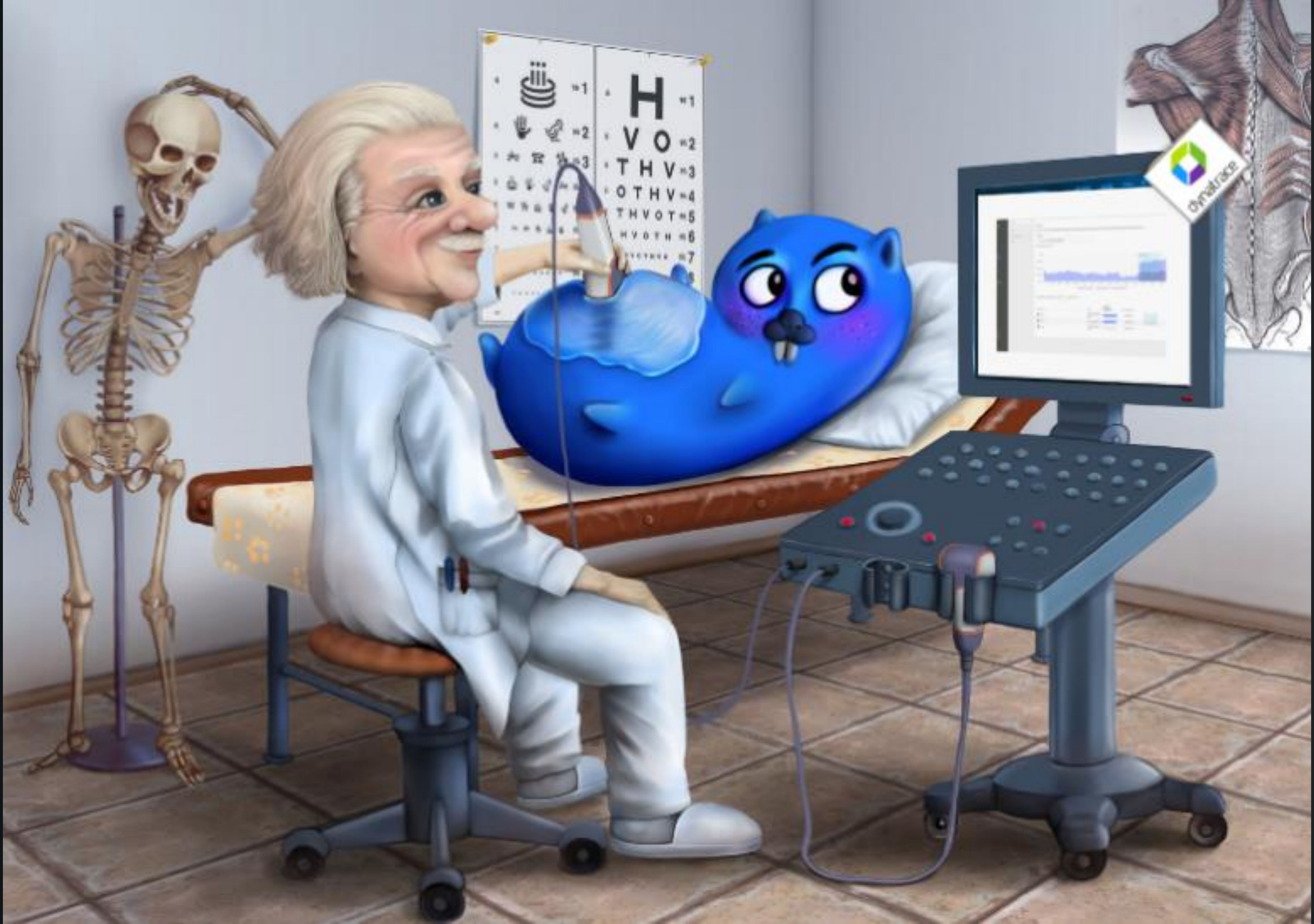
Avg. response time (caller-side) 252 µs
Requests 2.07k

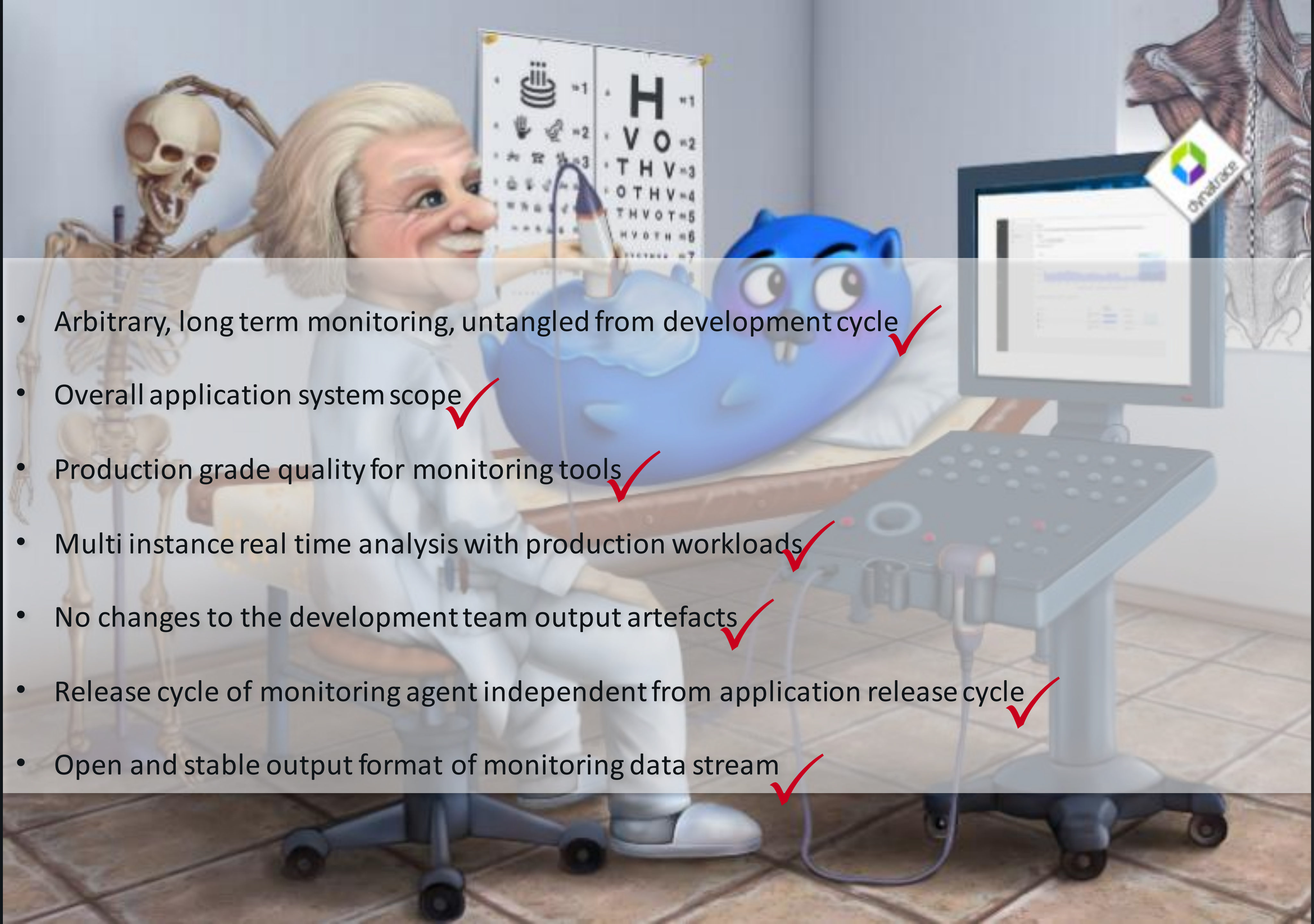
Process

Metrics


Heap anatomy
Live size is heap memory retained by the most recent GC plus allocations since then

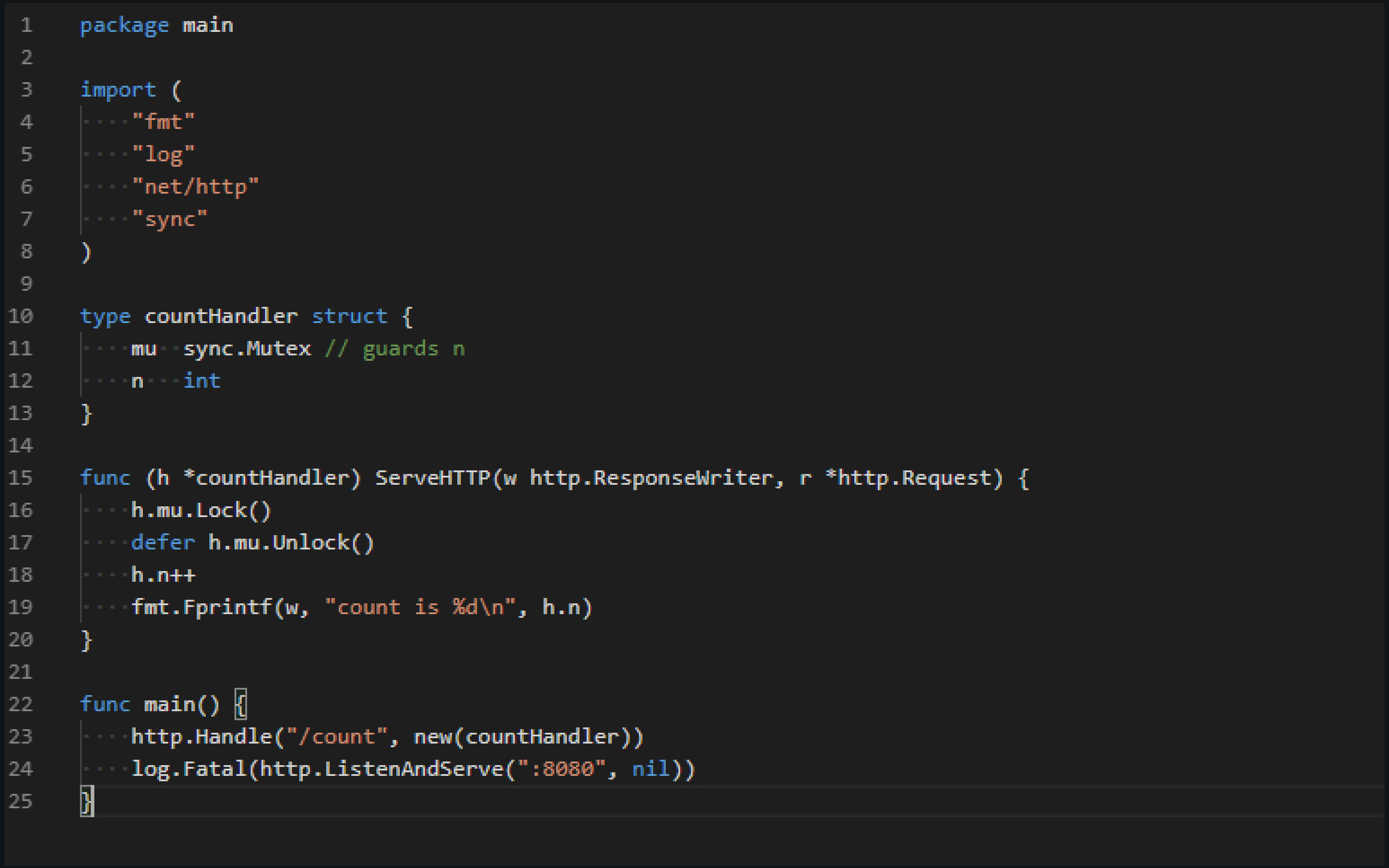
Operational monitoring	Development diagnostics
Arbitrary, long term monitoring, untangled from development cycle	Event driven, typically within issue analysis
Overall application system scope	Application building block scope (e.g. microservice)
Production grade quality for monitoring tools	No specific requirements
Multi instance real time analysis with production workloads	Offline analysis with mostly synthetic workloads
No changes to the development team output artefacts	Code changes, special build modes
Release cycle of monitoring agent independent from application release cycle	Monitoring agent release cycle bound to development cycle
Open and stable output format of monitoring data stream	Data persisted to files; version dependent format


H
V O
T H V
O T H V
T H V O T
dynatrace

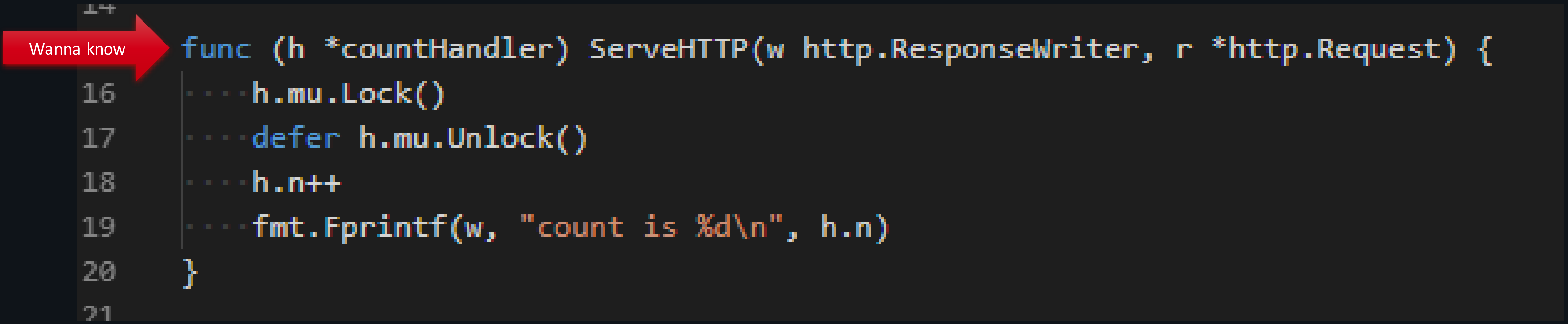


- Arbitrary, long term monitoring, untangled from development cycle ✓
- Overall application system scope ✓
- Production grade quality for monitoring tools ✓
- Multi instance real time analysis with production workloads ✓
- No changes to the development team output artefacts ✓
- Release cycle of monitoring agent independent from application release cycle ✓
- Open and stable output format of monitoring data stream ✓

Under the hoods of a Go monitoring agent


```go
package main

import (
    "fmt"
    "log"
    "net/http"
    "sync"
)

type countHandler struct {
    mu  sync.Mutex // guards n
    n   int
}

func (h *countHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    h.mu.Lock()
    defer h.mu.Unlock()
    h.n++
    fmt.Fprintf(w, "count is %d\n", h.n)
}

func main() {
    http.Handle("/count", new(countHandler))
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

https://golang.org/pkg/net/http/#example_Handle



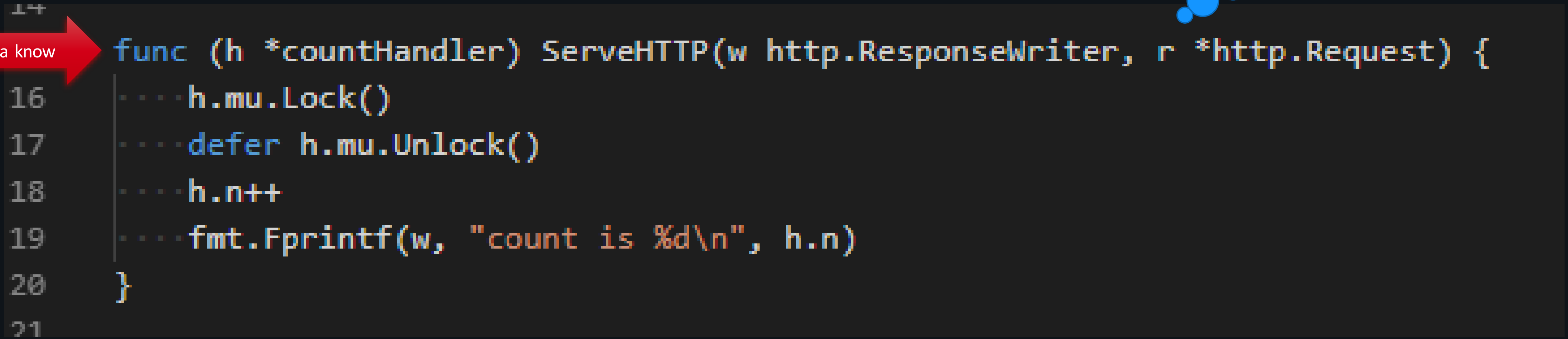
Wanna know

```go
func (h *countHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    h.mu.Lock()
    defer h.mu.Unlock()
    h.n++
    fmt.Fprintf(w, "count is %d\n", h.n)
}
```

https://golang.org/pkg/net/http/#example_Handle

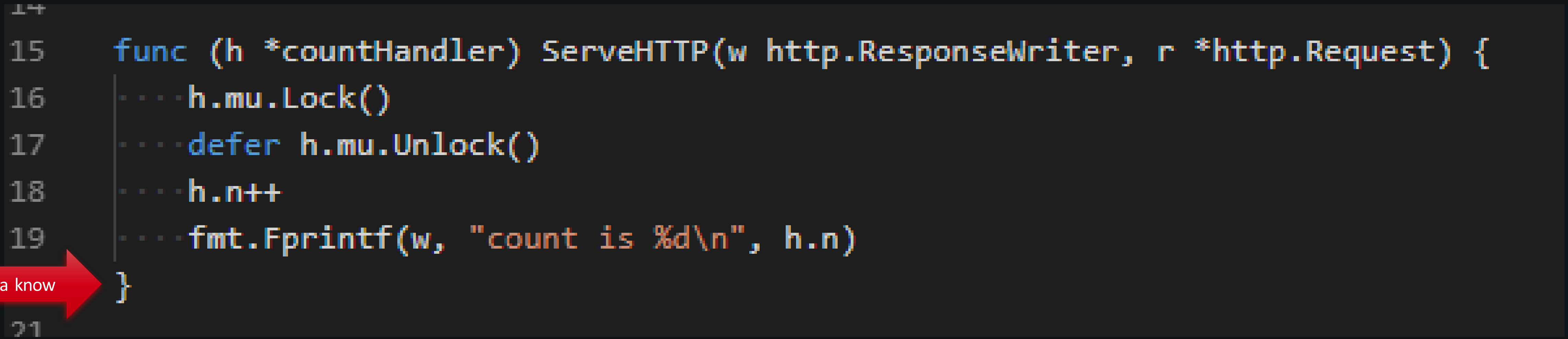
Headers
Remote address
RequestURI
Method
Time stamps

Wanna know



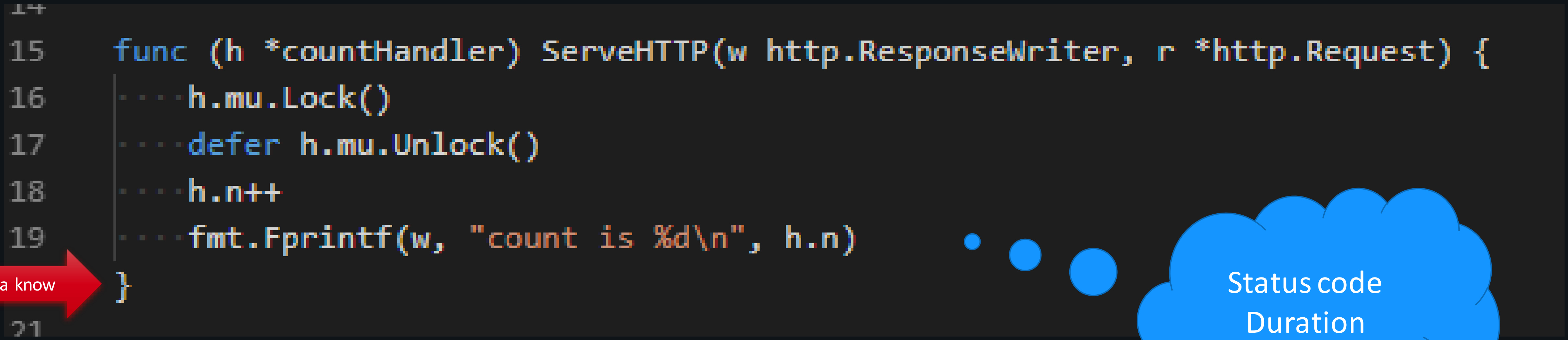
```go
func (h *countHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    h.mu.Lock()
    defer h.mu.Unlock()
    h.n++
    fmt.Fprintf(w, "count is %d\n", h.n)
}
```

https://golang.org/pkg/net/http/#example_Handle



```go
func (h *countHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    h.mu.Lock()
    defer h.mu.Unlock()
    h.n++
    fmt.Fprintf(w, "count is %d\n", h.n)
}
```

Wanna know



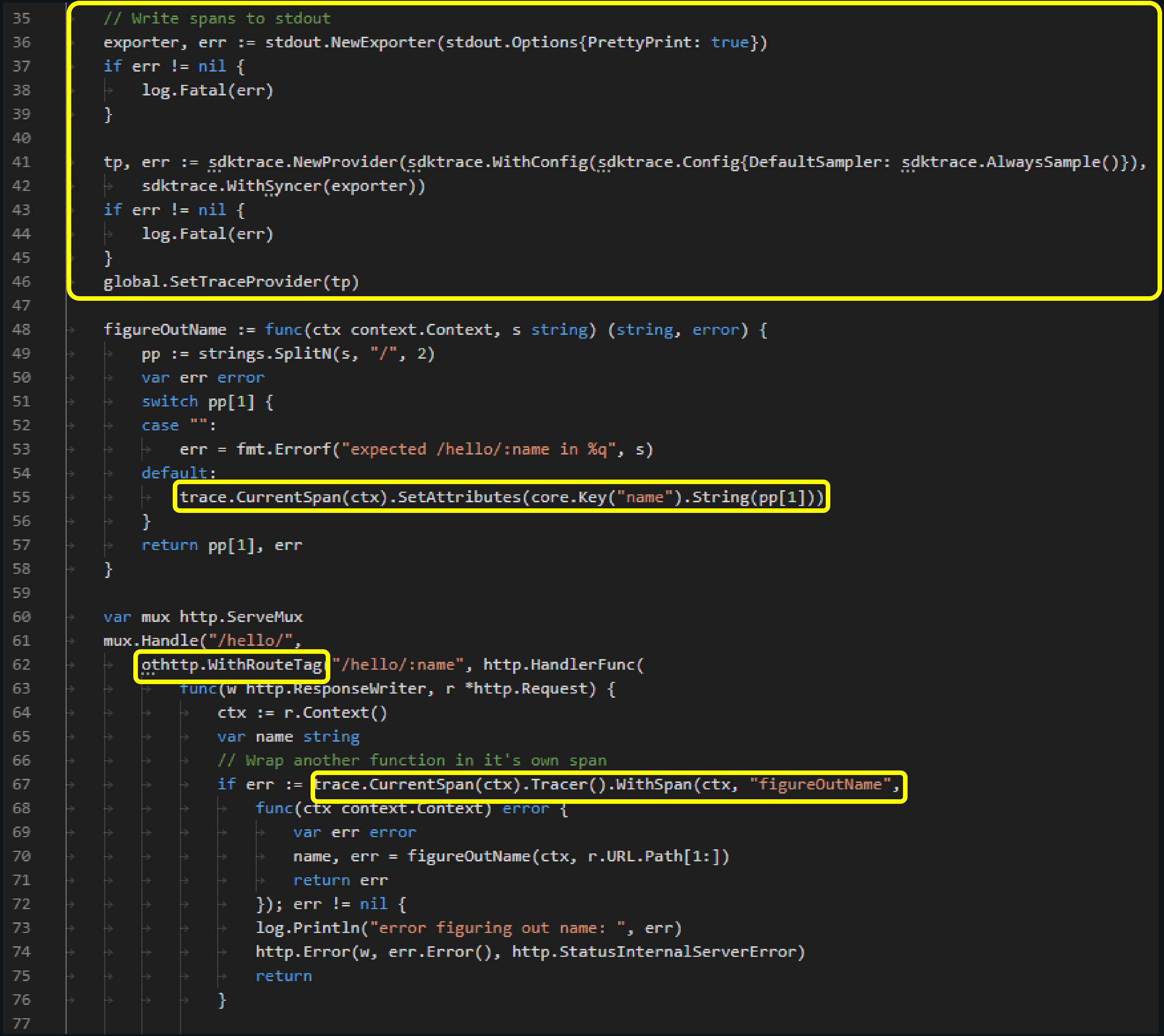
```go
func (h *countHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    h.mu.Lock()
    defer h.mu.Unlock()
    h.n++
    fmt.Fprintf(w, "count is %d\n", h.n)
}
```

Wanna know

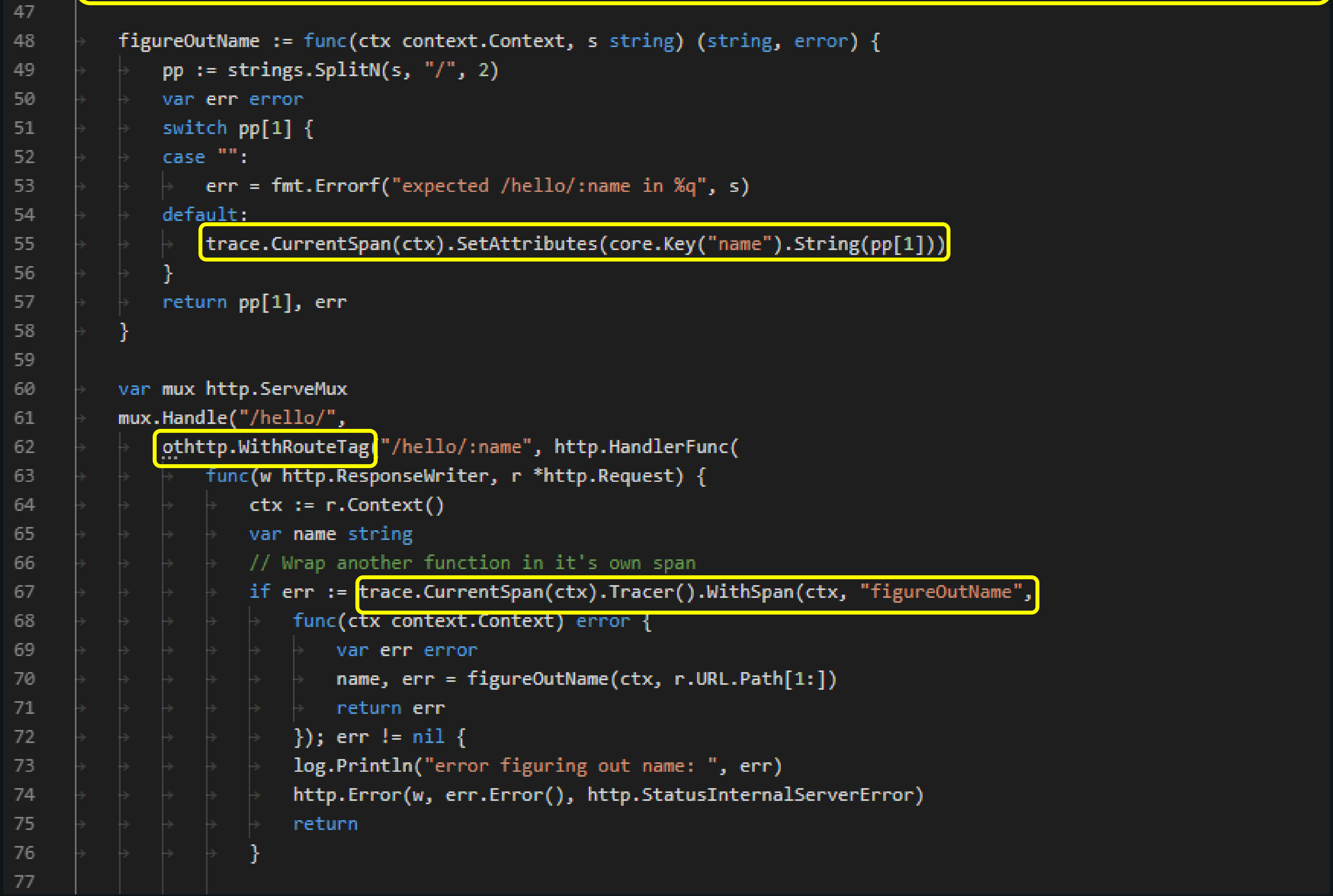
Status code
Duration
CPU timings

https://golang.org/pkg/net/http/#example_Handle

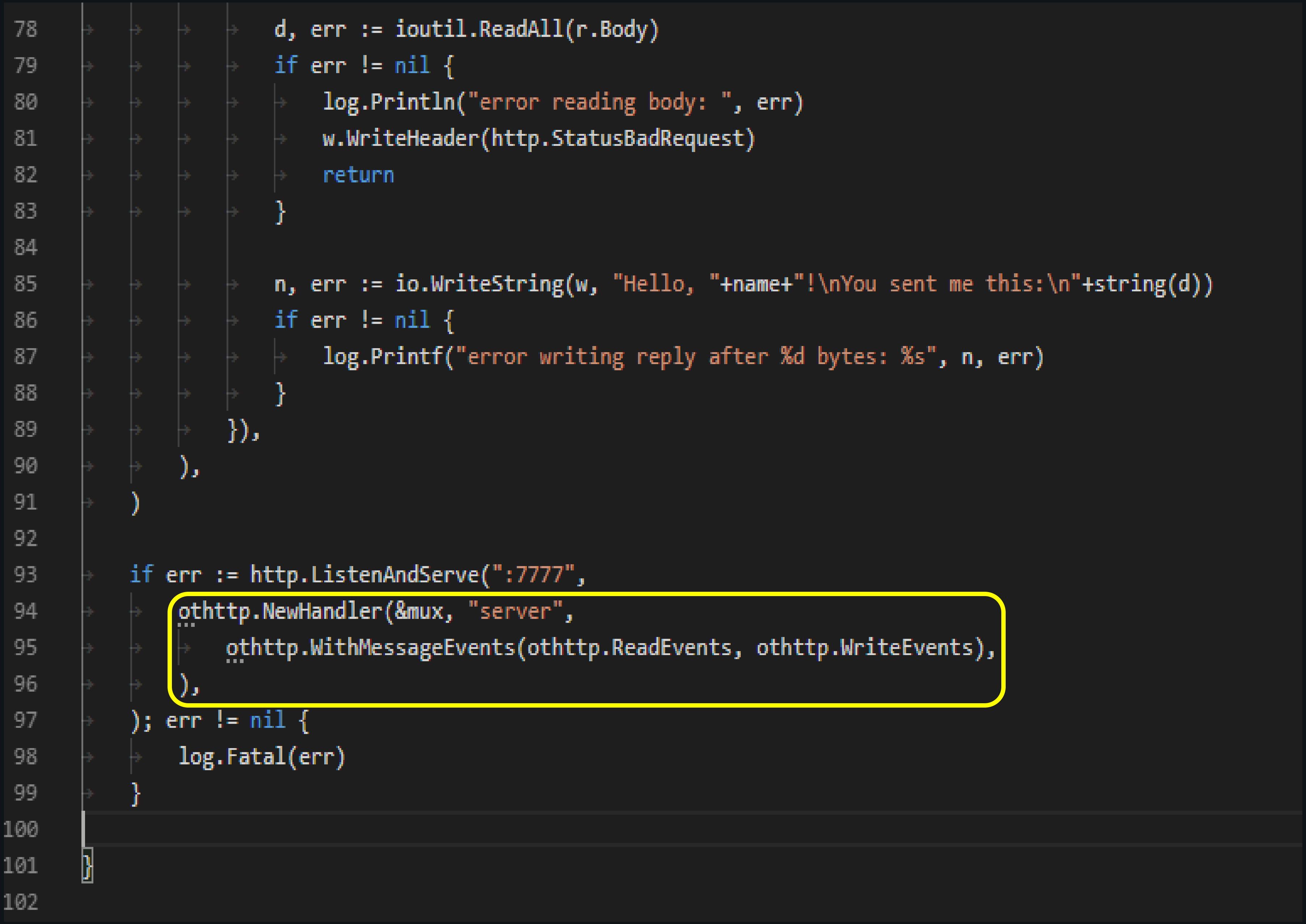
Manual instrumentation



```go
    // Write spans to stdout
    exporter, err := stdout.NewExporter(stdout.Options{PrettyPrint: true})
    if err != nil {
        log.Fatal(err)
    }

    tp, err := sdktrace.NewProvider(sdktrace.WithConfig(sdktrace.Config{DefaultSampler: sdktrace.AlwaysSample()}),
        sdktrace.WithSyncer(exporter))
    if err != nil {
        log.Fatal(err)
    }
    global.SetTraceProvider(tp)

    figureOutName := func(ctx context.Context, s string) (string, error) {
        pp := strings.SplitN(s, "/", 2)
        var err error
        switch pp[1] {
        case "":
            err = fmt.Errorf("expected /hello/:name in %q", s)
        default:
            trace.CurrentSpan(ctx).SetAttributes(core.Key("name").String(pp[1]))
        }
        return pp[1], err
    }

    var mux http.ServeMux
    mux.Handle("/hello/",
        othttp.WithRouteTag("/hello/:name", http.HandlerFunc(
            func(w http.ResponseWriter, r *http.Request) {
                ctx := r.Context()
                var name string
                // Wrap another function in it's own span
                if err := trace.CurrentSpan(ctx).Tracer().WithSpan(ctx, "figureOutName",
                    func(ctx context.Context) error {
                        var err error
                        name, err = figureOutName(ctx, r.URL.Path[1:])
                        return err
                    }); err != nil {
                    log.Println("error figuring out name: ", err)
                    http.Error(w, err.Error(), http.StatusInternalServerError)
                    return
                }

```

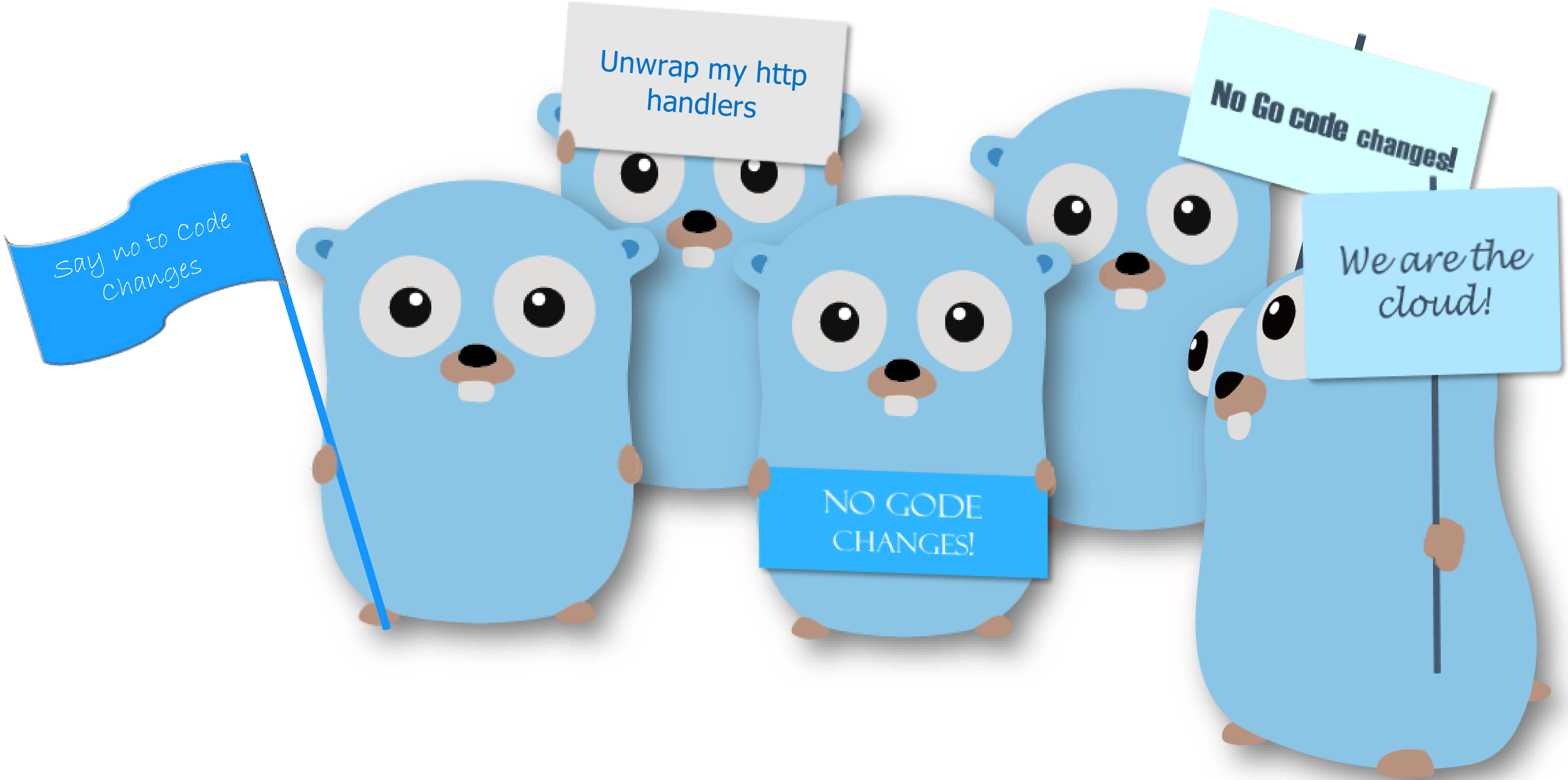


```go

figureOutName := func(ctx context.Context, s string) (string, error) {
    pp := strings.SplitN(s, "/", 2)
    var err error
    switch pp[1] {
    case "":
        err = fmt.Errorf("expected /hello/:name in %q", s)
    default:
        trace.CurrentSpan(ctx).SetAttributes(core.Key("name").String(pp[1]))
    }
    return pp[1], err
}

var mux http.ServeMux
mux.Handle("/hello/",
    othttp.WithRouteTag("/hello/:name", http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
            ctx := r.Context()
            var name string
            // Wrap another function in it's own span
            if err := trace.CurrentSpan(ctx).Tracer().WithSpan(ctx, "figureOutName",
                func(ctx context.Context) error {
                    var err error
                    name, err = figureOutName(ctx, r.URL.Path[1:])
                    return err
                }); err != nil {
                log.Println("error figuring out name: ", err)
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
            }

```



```go
				d, err := ioutil.ReadAll(r.Body)
				if err != nil {
					log.Println("error reading body: ", err)
					w.WriteHeader(http.StatusBadRequest)
					return
				}

				n, err := io.WriteString(w, "Hello, "+name+"!\nYou sent me this:\n"+string(d))
				if err != nil {
					log.Printf("error writing reply after %d bytes: %s", n, err)
				}
			}),
		),
	)

	if err := http.ListenAndServe(":7777",
		othttp.NewHandler(&mux, "server",
			othttp.WithMessageEvents(othttp.ReadEvents, othttp.WriteEvents),
		),
	); err != nil {
		log.Fatal(err)
	}

}

```

https://godoc.org/go.opentelemetry.io/otel/plugin/othttp#example-NewHandler

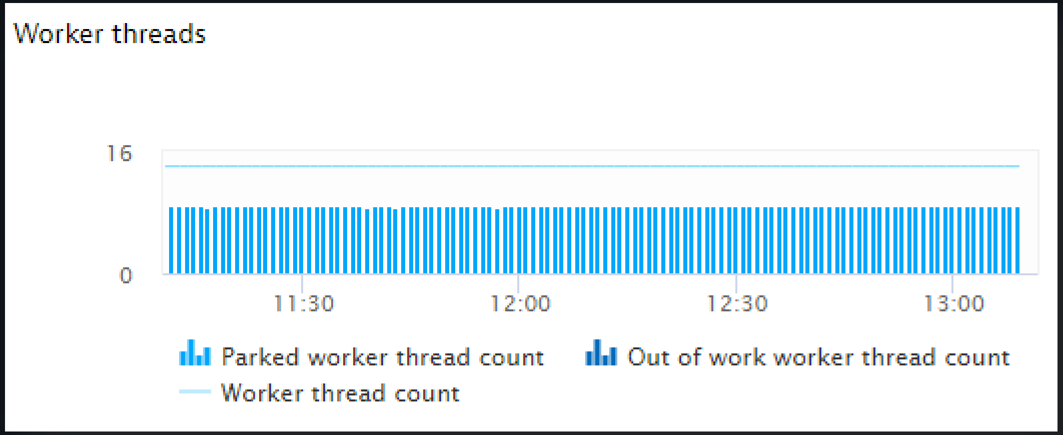

Say no to Code changes
Unwrap my http handlers
NO CODE CHANGES!
No Go code changes!
We are the cloud!

Behind the curtain …

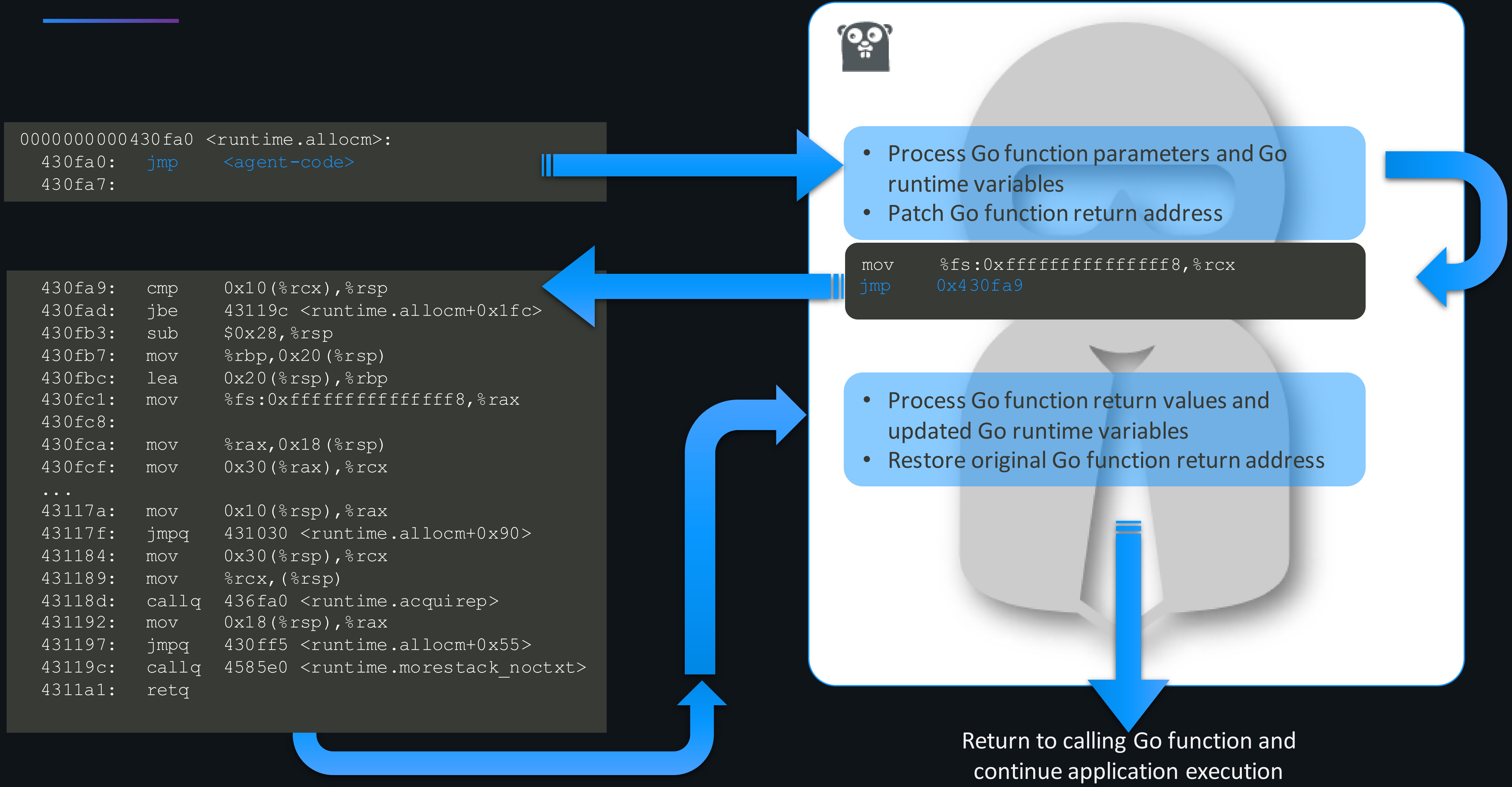
```
0000000000430fa0 <runtime.allocm>:
  430fa0:   mov    %fs:0xfffffffffffffff8,%rcx
  430fa7:

  430fa9:   cmp    0x10(%rcx),%rsp
  430fad:   jbe    43119c <runtime.allocm+0x1fc>
  430fb3:   sub    $0x28,%rsp
  430fb7:   mov    %rbp,0x20(%rsp)
  430fbc:   lea    0x20(%rsp),%rbp
  430fc1:   mov    %fs:0xfffffffffffffff8,%rax
  430fc8:
  430fca:   mov    %rax,0x18(%rsp)
  430fcf:   mov    0x30(%rax),%rcx
  ...
  43117a:   mov    0x10(%rsp),%rax
  43117f:   jmpq   431030 <runtime.allocm+0x90>
  431184:   mov    0x30(%rsp),%rcx
  431189:   mov    %rcx,(%rsp)
  43118d:   callq  436fa0 <runtime.acquirep>
  431192:   mov    0x18(%rsp),%rax
  431197:   jmpq   430ff5 <runtime.allocm+0x55>
  43119c:   callq  4585e0 <runtime.morestack_noctxt>
  4311a1:   retq
```

`runtime.allocm` is a Go runtime internal function used in the process to create new worker threads to execute Goroutines.

- Source for “Worker thread count” metric data



Behind the curtain ...



```
0000000000430fa0 <runtime.allocm>:
  430fa0:   jmp    <agent-code>
  430fa7:
```

```
  430fa9:   cmp    0x10(%rcx),%rsp
  430fad:   jbe    43119c <runtime.allocm+0x1fc>
  430fb3:   sub    $0x28,%rsp
  430fb7:   mov    %rbp,0x20(%rsp)
  430fbc:   lea    0x20(%rsp),%rbp
  430fc1:   mov    %fs:0xfffffffffffffff8,%rax
  430fc8:
  430fca:   mov    %rax,0x18(%rsp)
  430fcf:   mov    0x30(%rax),%rcx
  ...
  43117a:   mov    0x10(%rsp),%rax
  43117f:   jmpq   431030 <runtime.allocm+0x90>
  431184:   mov    0x30(%rsp),%rcx
  431189:   mov    %rcx,(%rsp)
  43118d:   callq  436fa0 <runtime.acquirep>
  431192:   mov    0x18(%rsp),%rax
  431197:   jmpq   430ff5 <runtime.allocm+0x55>
  43119c:   callq  4585e0 <runtime.morestack_noctxt>
  4311a1:   retq
```

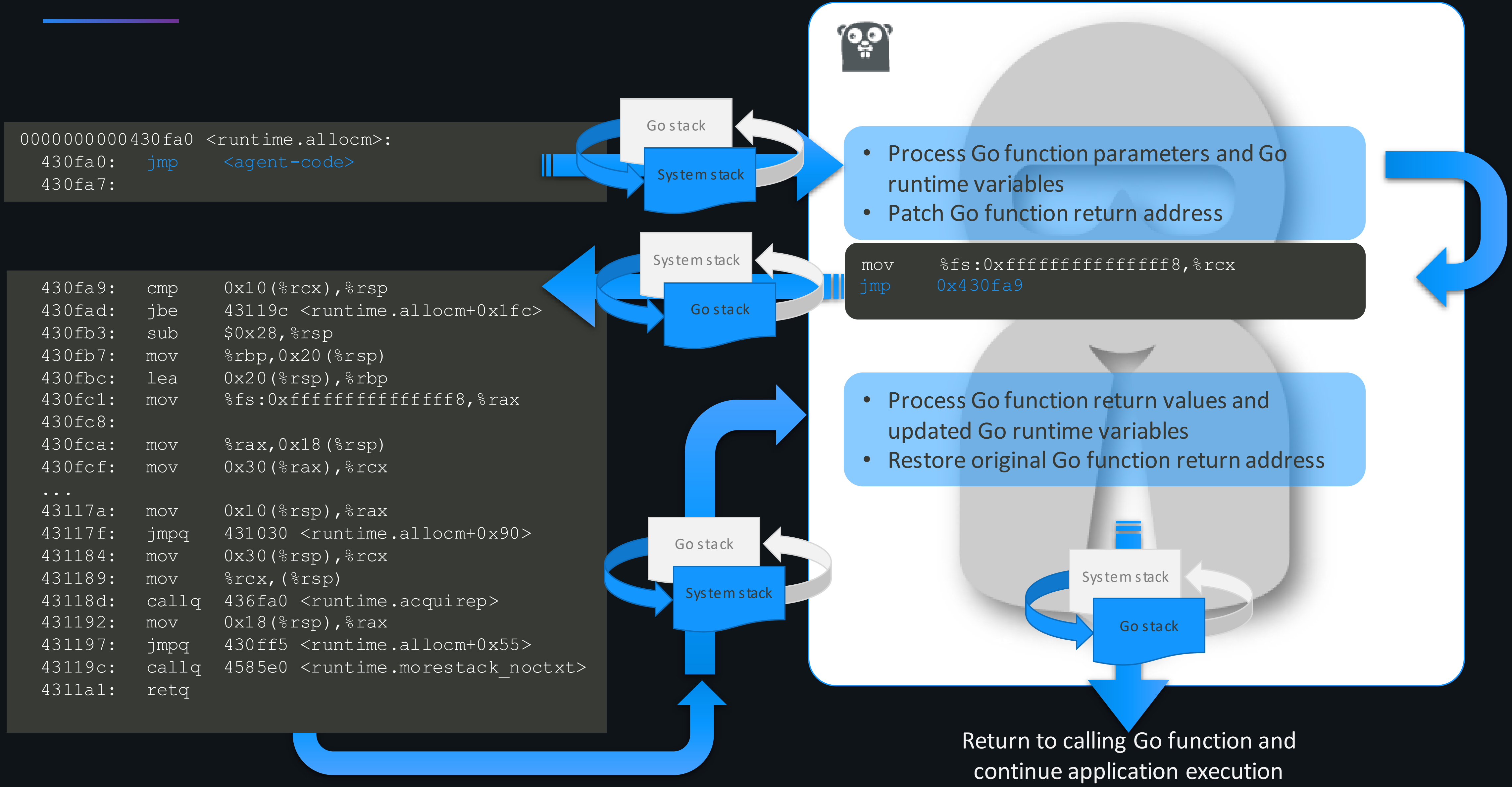
- Process Go function parameters and Go runtime variables
- Patch Go function return address

```
mov    %fs:0xfffffffffffffff8,%rcx
jmp    0x430fa9
```

- Process Go function return values and updated Go runtime variables
- Restore original Go function return address

Return to calling Go function and continue application execution

Behind the curtain ...



```
0000000000430fa0 <runtime.allocm>:
  430fa0:   jmp    <agent-code>
  430fa7:
```

- Process Go function parameters and Go runtime variables
- Patch Go function return address

```
mov      %fs:0xfffffffffffffff8,%rcx
jmp      0x430fa9
```

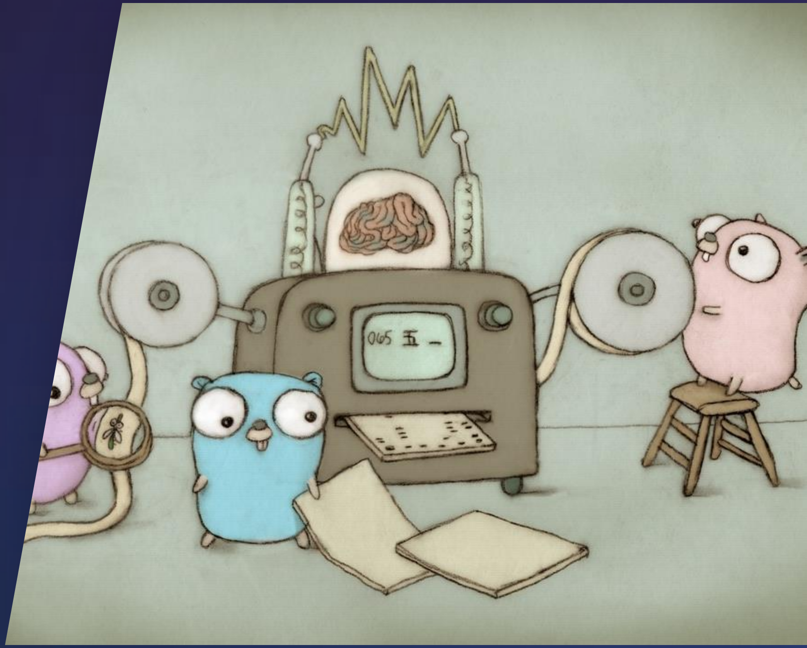
```
  430fa9:   cmp    0x10(%rcx),%rsp
  430fad:   jbe    43119c <runtime.allocm+0x1fc>
  430fb3:   sub    $0x28,%rsp
  430fb7:   mov    %rbp,0x20(%rsp)
  430fbc:   lea    0x20(%rsp),%rbp
  430fc1:   mov    %fs:0xfffffffffffffff8,%rax
  430fc8:
  430fca:   mov    %rax,0x18(%rsp)
  430fcf:   mov    0x30(%rax),%rcx
  ...
  43117a:   mov    0x10(%rsp),%rax
  43117f:   jmpq   431030 <runtime.allocm+0x90>
  431184:   mov    0x30(%rsp),%rcx
  431189:   mov    %rcx,(%rsp)
  43118d:   callq  436fa0 <runtime.acquirep>
  431192:   mov    0x18(%rsp),%rax
  431197:   jmpq   430ff5 <runtime.allocm+0x55>
  43119c:   callq  4585e0 <runtime.morestack_noctxt>
  4311a1:   retq
```

- Process Go function return values and updated Go runtime variables
- Restore original Go function return address

Return to calling Go function and continue application execution

Golang Proposals

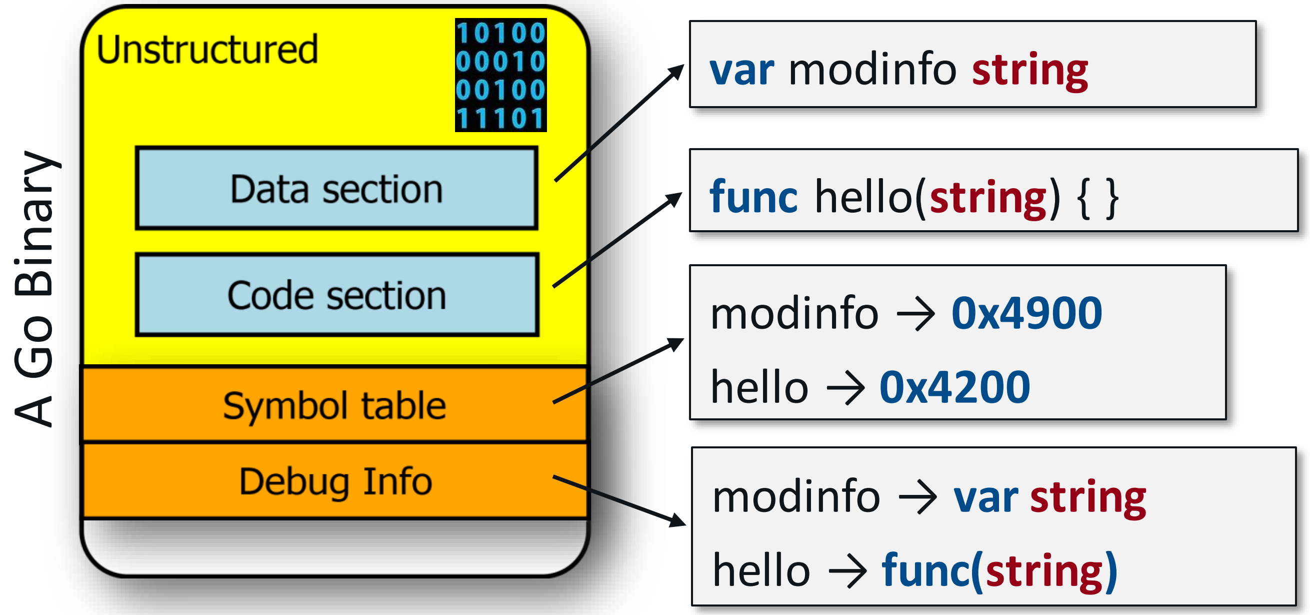
Newsflash from department of proposals



dynatrace

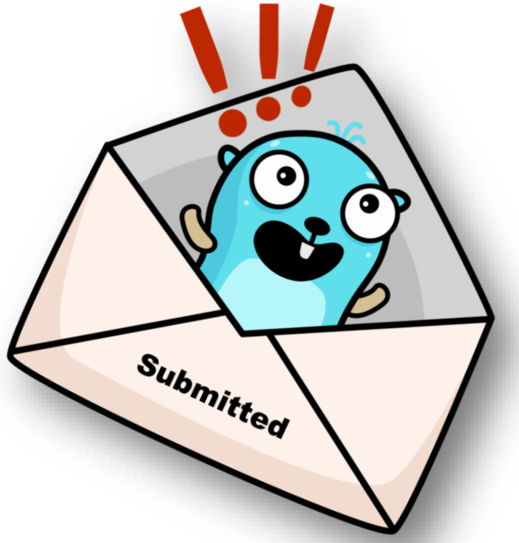
Proposal: Build meta information

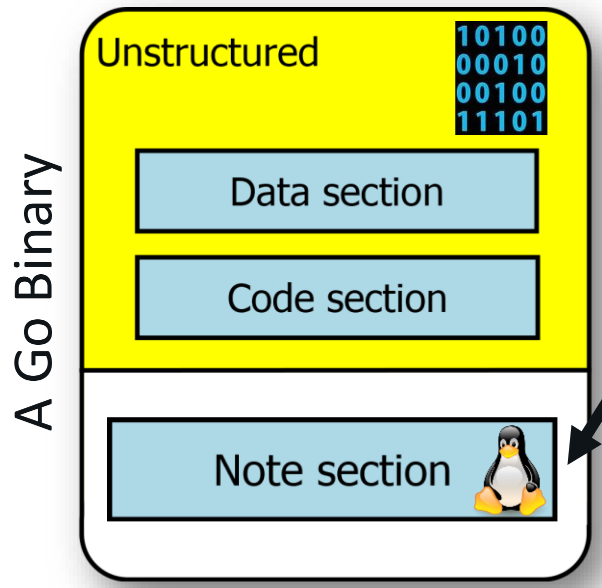
- Goal: Extract meta information from Go binaries without execution
- Current situation:
  - Read meta data from **global variables**, e.g. Go modules package list:
    - `modinfo: "github.com/grpc/grpc-go\tv1.22\th1:fHOK...\n`
  - Parse symbol/debug information (may not be available)



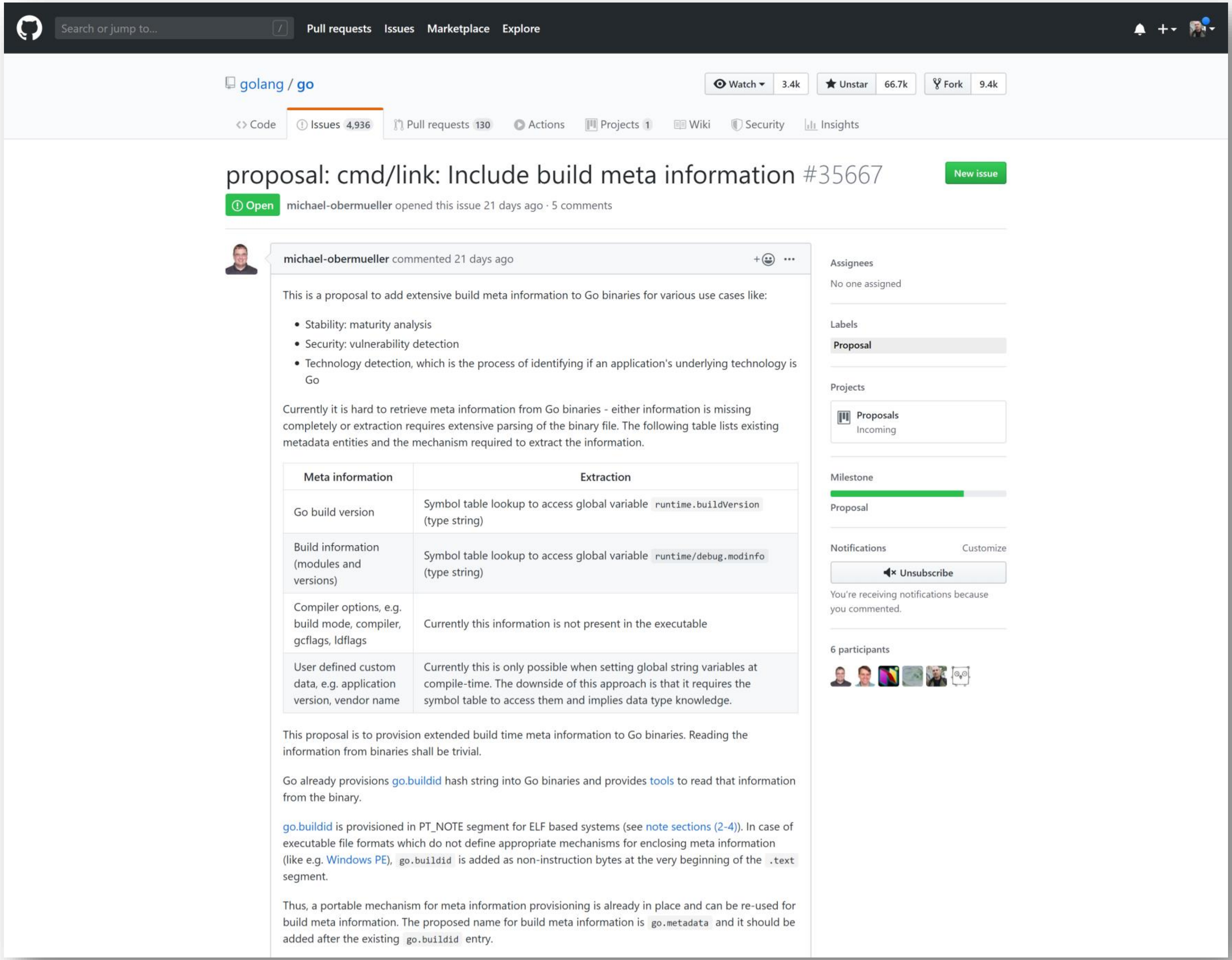
Proposal: Build meta information

- Add extensive build meta information to Go binaries
- Reading information shall be trivial
- Submitted: Github Golang issue 35667





```
{
  "version": "go1.13.4",
  "compileropts": {
    "os": "linux",
    "arch": "amd64",
  },
  "deps": [
    {
      "path": "github.com/grpc/grpc-go",
      "version": "v1.22",
      "sum": "h1:iURUrRGxPUNPd="
    }
  ]
}
```



proposal: cmd/link: Include build meta information #35667

Open michael-obermueller opened this issue 21 days ago · 5 comments

michael-obermueller commented 21 days ago

This is a proposal to add extensive build meta information to Go binaries for various use cases like:

- Stability: maturity analysis
- Security: vulnerability detection
- Technology detection, which is the process of identifying if an application's underlying technology is Go

Currently it is hard to retrieve meta information from Go binaries - either information is missing completely or extraction requires extensive parsing of the binary file. The following table lists existing metadata entities and the mechanism required to extract the information.

Meta information	Extraction
Go build version	Symbol table lookup to access global variable `runtime.buildVersion` (type string)
Build information (modules and versions)	Symbol table lookup to access global variable `runtime/debug.modinfo` (type string)
Compiler options, e.g. build mode, compiler, gcflags, ldflags	Currently this information is not present in the executable
User defined custom data, e.g. application version, vendor name	Currently this is only possible when setting global string variables at compile-time. The downside of this approach is that it requires the symbol table to access them and implies data type knowledge.

This proposal is to provision extended build time meta information to Go binaries. Reading the information from binaries shall be trivial.

Go already provisions go.buildid hash string into Go binaries and provides tools to read that information from the binary.

go.buildid is provisioned in PT_NOTE segment for ELF based systems (see note sections (2-4)). In case of executable file formats which do not define appropriate mechanisms for enclosing meta information (like e.g. Windows PE), `go.buildid` is added as non-instruction bytes at the very beginning of the `.text` segment.

Thus, a portable mechanism for meta information provisioning is already in place and can be re-used for build meta information. The proposed name for build meta information is `go.metadata` and it should be added after the existing `go.buildid` entry.

Assignees
No one assigned

Labels
Proposal

Projects
Proposals
Incoming

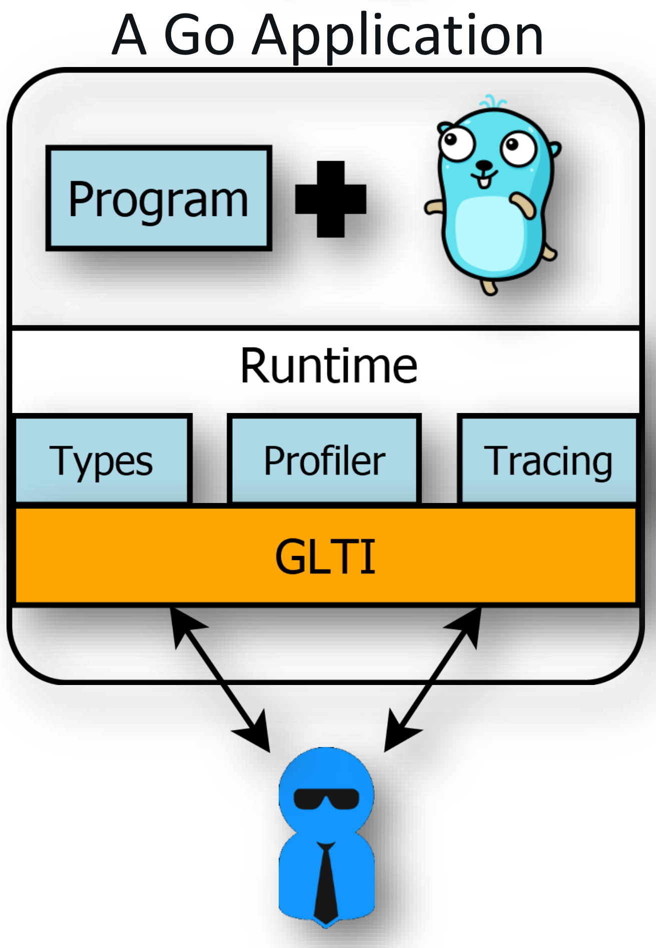
Milestone
Proposal

Notifications Customize
Unsubscribe
You're receiving notifications because you commented.

6 participants

Proposal: Golang Tool Interface (GLTI)

- Goal: Introduce **operational monitoring** without code changes
- Open access to **existing** application meta information and tooling
- **Defined** agent loading procedure
  - Load monitoring code at runtime initialization
  - Go plugins, native shared objects
- Beneficial for many tools



dynatrace

dynatrace.com

https://github.com/golang-samples/gopher-vector/blob/master/gopher-front.png and CC BY-SA