

How to Instrument Go Without Changing a Single Line of Code

Hannah Kim, Kemal Akkoyun

FOSDEM 2026

Actually...

Does any of this matter anymore?

AI writes all our code now anyway...

WHY do we care?

The Observability Promise

What we want:

- Understand system behavior
- Debug production issues fast
- Prevent outages before they happen

What we get:

- Distributed complexity
- Partial visibility
- "It works on my machine"

The Instrumentation Tax

Every new service requires:

- Import the SDK
- Initialize the tracer
- Wrap every handler
- Propagate context everywhere
- Handle shutdown gracefully

Multiply by 100 microservices...

A Story

Where instrumentation gets in the way:

- **Vendor lock-in:** Committed to one APM? Good luck switching
- **Code pollution:** Business logic buried under telemetry
- **Inconsistent coverage:** Some services instrumented, some not
- **Performance anxiety:** "Is this span worth the overhead?"

The Dream

What if we could have observability without changing code?

- No SDK imports
- No wrapper functions
- No context propagation boilerplate
- Just... observability

What is instrumentation?

your application

your application → your backend

your application → your backend

???

???

**your application → your
backend**

LOGS

(what happened)

???

???

**your application → your
backend**

???

???

LOGS

(what happened)

METRICS

(how much/fast things happened)

**your application → your
backend**

???

???

LOGS

(what happened)

METRICS

(how much/fast things happened)

TRACES

(how things happened)

WHAT IS AUTO-INSTRUMENTATION? 🤔

Manual Instrumentation (Before)

```
func handleRequest(w http.ResponseWriter, r *http.Request) {  
    // Just business logic  
    result := processData(r.Body)  
    json.NewEncoder(w).Encode(result)  
}
```

Manual Instrumentation (After)

```
func handleRequest(w http.ResponseWriter, r *http.Request) {
    ctx, span := tracer.Start(r.Context(), "handleRequest")
    defer span.End()

    span.SetAttributes(
        attribute.String("http.method", r.Method),
        attribute.String("http.url", r.URL.Path),
    )

    result := processData(ctx, r.Body)
    json.NewEncoder(w).Encode(result)
}
```

+15 lines per handler

What is auto-instrumentation?

1. I want to know more about my code
2. I need to instrument it, but I'm too lazy to do it myself
3. ???

What is auto-instrumentation?

1. I want to know more about my code
2. I need to instrument it, but I'm too lazy to do it myself
3. ???

Profit



What is auto-instrumentation?

auto-instrumentation: instrumenting your code (collecting signals + data) without manual code changes

What is auto-instrumentation?

RUN TIME

- Happens at runtime (surprised???)
- Sometimes causes source code changes
- Meh with compiled languages like Go, C++, etc.

auto-instrumentation: instrumenting your code (collecting signals + data) without manual code changes

What is auto-instrumentation?

RUN TIME

- Happens at runtime (surprised???)
- Sometimes causes source code changes
- Meh with compiler languages like Go

COMPILE TIME

- Happens at... compile time
- (Before run time)
- Works great with compiler languages like Go

auto-instrumentation: instrumenting your code (collecting signals + data) without manual code changes

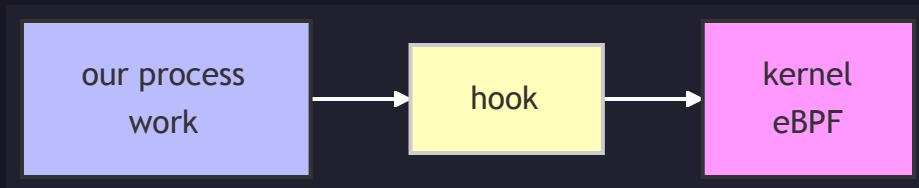
Runtime Approaches

- **eBPF**: extended Berkeley packet filter
 - `uprobe` hooks
 - `kprobe` hooks
 - `USDT` (Userland Statically Defined Tracing) hooks
- **Library injection (LD_PRELOAD)**

How eBPF Works



How eBPF Works

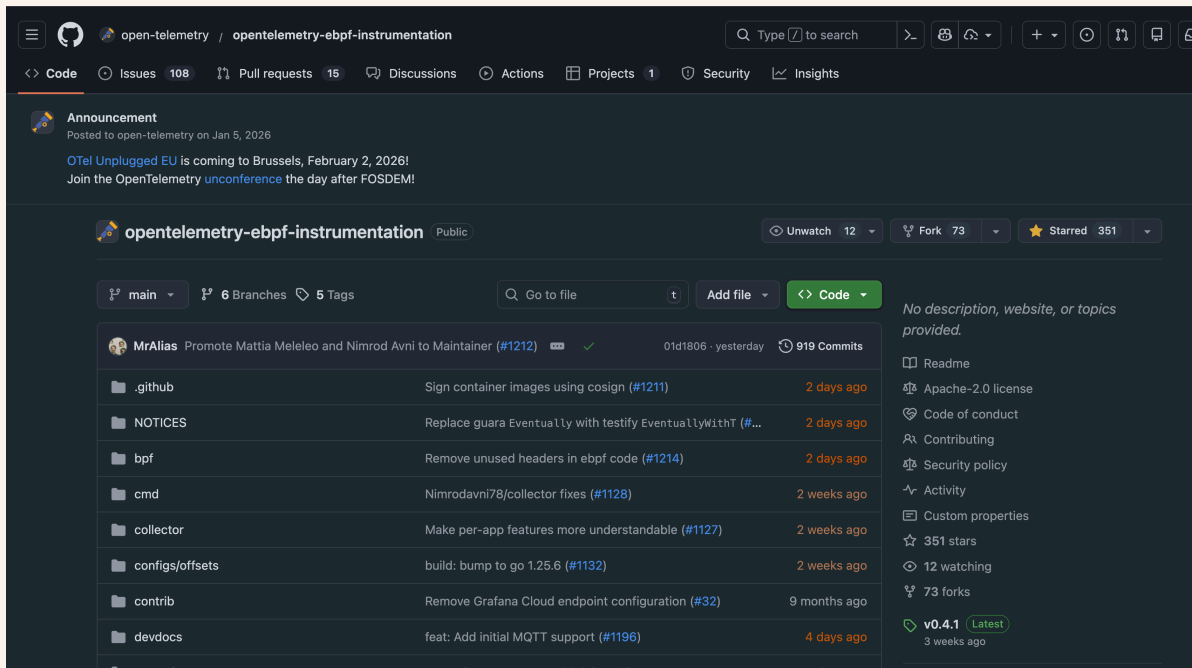


eBPF Auto-Instrumentation

```
# No code changes – just deploy a sidecar
apiVersion: v1
kind: Pod
spec:
  containers:
    - name: app
      image: myapp:latest
    - name: otel-auto
      image: otel/autoinstrumentation-go:latest
      securityContext:
        privileged: true # Required for eBPF
```

Hooks into Go runtime via uprobes

OpenTelemetry eBPF Instrumentation (OBI)



The screenshot shows the GitHub repository page for `opentelemetry-ebpf-instrumentation`. The repository is public and has 108 issues, 15 pull requests, 1 project, and 1 security issue. It has 12 watchers, 73 forks, and 351 stars. The repository is currently on the `main` branch, with 6 other branches and 5 tags. The repository is managed by `MrAlias`, who promoted Mattia Meleleo and Nimrod Avni to maintainers. The repository has 919 commits. The repository is currently on the `main` branch, with 6 other branches and 5 tags. The repository is managed by `MrAlias`, who promoted Mattia Meleleo and Nimrod Avni to maintainers. The repository has 919 commits. The repository is currently on the `main` branch, with 6 other branches and 5 tags. The repository is managed by `MrAlias`, who promoted Mattia Meleleo and Nimrod Avni to maintainers. The repository has 919 commits.

File	Description	Time
<code>.github</code>	Sign container images using cosign (#1211)	2 days ago
<code>NOTICES</code>	Replace guara Eventually with testify EventuallyWithT (#1212)	2 days ago
<code>bpf</code>	Remove unused headers in ebpf code (#1214)	2 days ago
<code>cmd</code>	Nimrodavni78/collector fixes (#1128)	2 weeks ago
<code>collector</code>	Make per-app features more understandable (#1127)	2 weeks ago
<code>configs/offsets</code>	build: bump to go 1.25.6 (#1132)	2 weeks ago
<code>contrib</code>	Remove Grafana Cloud endpoint configuration (#32)	9 months ago
<code>devdocs</code>	feat: Add initial MQTT support (#1196)	4 days ago
<code>examples</code>	Upgrade prometheus to 1.20.0 (#1176)	5 days ago

Metadata:

- Readme
- Apache-2.0 license
- Code of conduct
- Contributing
- Security policy
- Activity
- Custom properties
- 351 stars
- 12 watching
- 73 forks
- v0.4.1 (Latest) 3 weeks ago

What is OBI?

What is OBI?

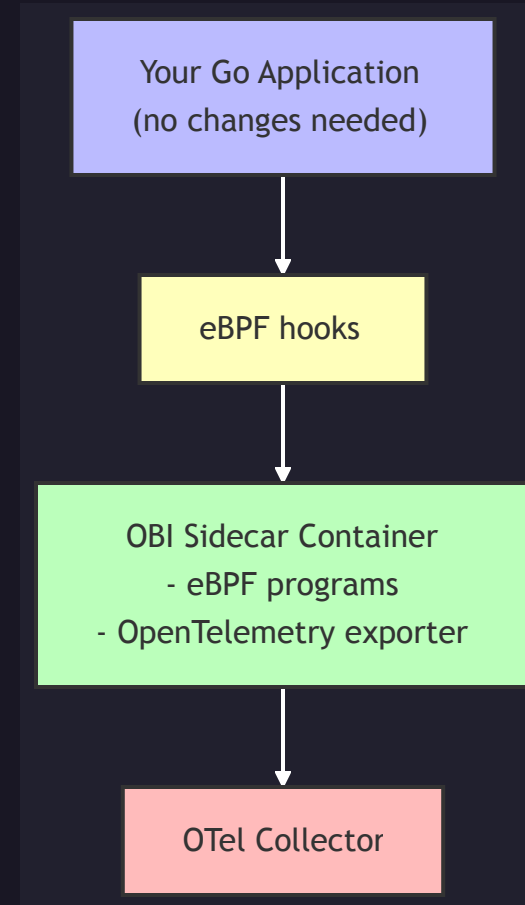
OBI (OpenTelemetry eBPF Instrumentation) is a runtime instrumentation approach that:

- Uses eBPF for network-level tracing
- **Multi-language:** Go, Java, .NET, Python, Node.js, Ruby, Rust
- **Protocol coverage:** HTTP/S, gRPC, TLS visibility
- Production-ready and vendor-neutral
- Requires administrative privileges (root access)

What is OBI?

OBI (OpenTelemetry eBPF Instrumentation) is a runtime instrumentation approach that:

- Uses eBPF for network-level tracing
- **Multi-language:** Go, Java, .NET, Python, Node.js, Ruby, Rust
- **Protocol coverage:** HTTP/S, gRPC, TLS visibility
- Production-ready and vendor-neutral
- Requires administrative privileges (root access)



OBI Configuration

```
# obi-config.yaml
open_port: 8080
service:
  name: fosdem-obi
log_level: debug

otel_traces_export:
  endpoint: http://otel-collector:4318

prometheus_export:
  port: 9090
  path: /metrics

meter_provider:
  features:
    - application
```

OBI (OpenTelemetry eBPF Instrumentation)

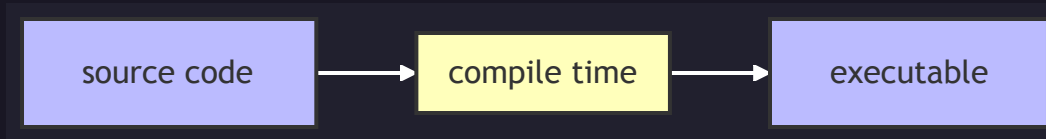
```
# Run alongside your application
docker run --privileged \
  --pid=container:myapp \
  -e OTEL_EXPORTER_OTLP_ENDPOINT=http://collector:4318 \
  otel/ebpf-instrumentation:latest
```

Attaches to running process - no restart needed

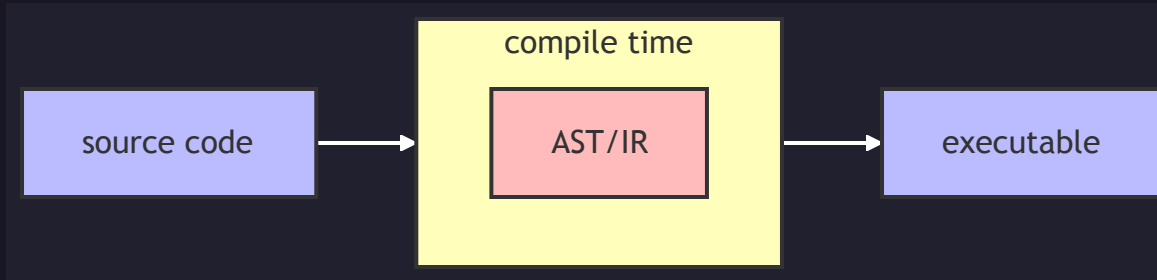
Compile Time Approaches

- OpenTelemetry Compile Time Instrumentation SIG
- Datadog Orchestrion

Compile Time Flow



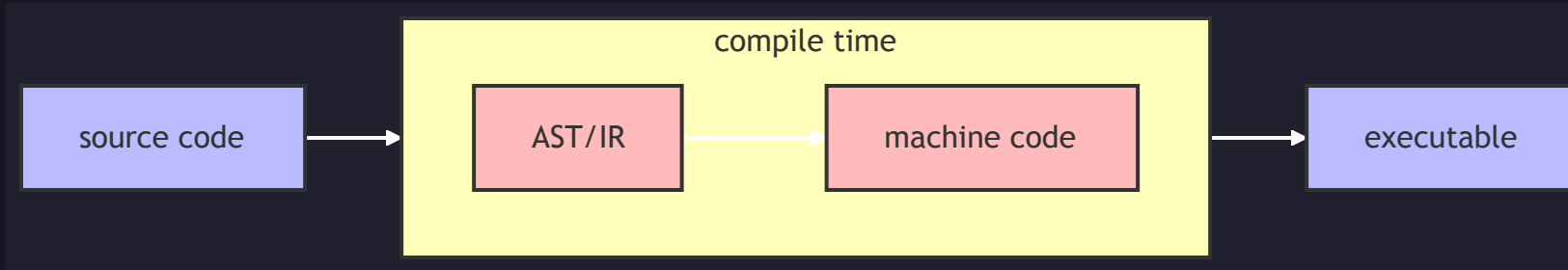
Compile Time Flow



AST: abstract syntax tree

IR: intermediate representation

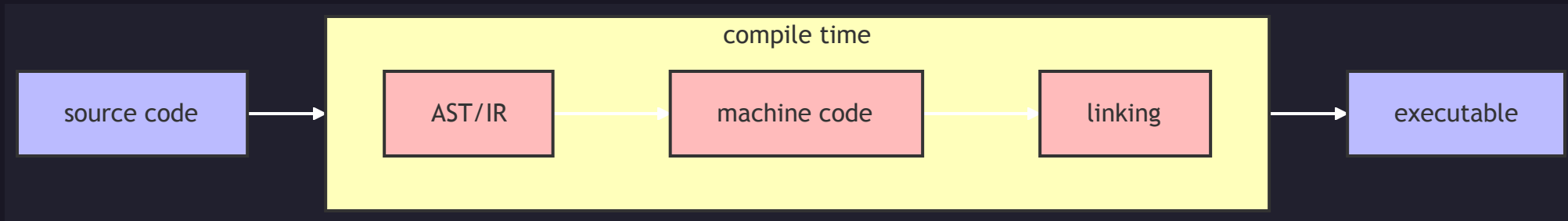
Compile Time Flow



AST: abstract syntax tree

IR: intermediate representation

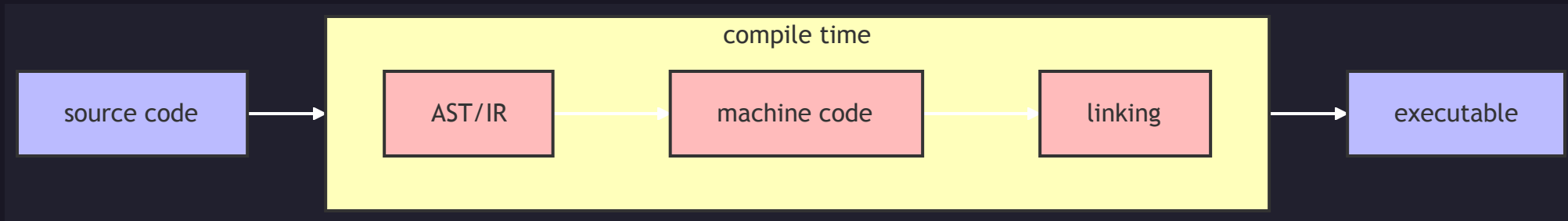
Compile Time Flow



AST: abstract syntax tree

IR: intermediate representation

Compile Time Flow



```
go run -toolexec 'orchestrian toolexec' .
```

AST: abstract syntax tree

IR: intermediate representation

What is Orchestrion?

Orchestrion is a compile-time instrumentation approach that:

- Uses Go's `-toolexec` to rewrite AST during compilation
- Aspect-oriented: declarative join points + advice templates
- **Vendor-neutral**: Supports OpenTelemetry (not Datadog-specific)
- Can instrument stdlib and dependencies automatically

Jan 2025: Datadog + Alibaba merging into unified OTel compile-time solution

Orchestrion Configuration

```
# orchestrion.yaml
aspects:
  - id: make spans
    join-point:
      all-of:
        - package-name: main
        - function-body:
            function:
              - name: main
    advice:
      - prepend-statements:
          imports:
            otel: go.opentelemetry.io/otel
            context: context
          template: |-
            tracer := otel.Tracer()
            _, span := tracer.Start(context.Background, "orchestrion.handler")
```

Orchestrion (Compile-Time)

```
# Build with instrumentation injected  
go build -toolexec 'orchestrion toolexec' -o myapp .  
  
# Or  
orchestrion go build -o myapp .
```

AST transformation during compilation

Benchmarking

Methodology

- **Environment:** Docker-based observability stack
- **Load Generator:** Archetypes (idle, throughput, latency, enterprise)
- **Metrics:** CPU, Memory, Latency (p50/p95/p99), Error rate
- **Application:** Same Go HTTP server across all scenarios

▮ Detailed methodology in our FOSDEM Software Performance Devroom talk

Scenarios Tested

1. **Default** - No instrumentation (baseline)
2. **Manual** - OpenTelemetry SDK with explicit spans
3. **OBI** - OpenTelemetry eBPF Instrumentation
4. **eBPF Auto** - OTel Auto-Instrumentation
5. **Orchestrion** - Compile-time code injection (OTel SDK)

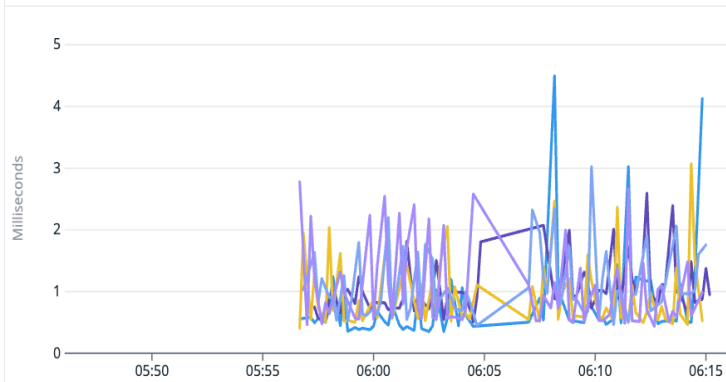
Environment Setup

- Docker Compose stack with:
 - Go application container
 - OTel Collector
 - Jaeger (traces)
 - Prometheus (metrics)
- Identical hardware allocation per scenario
- 5-minute sustained load tests

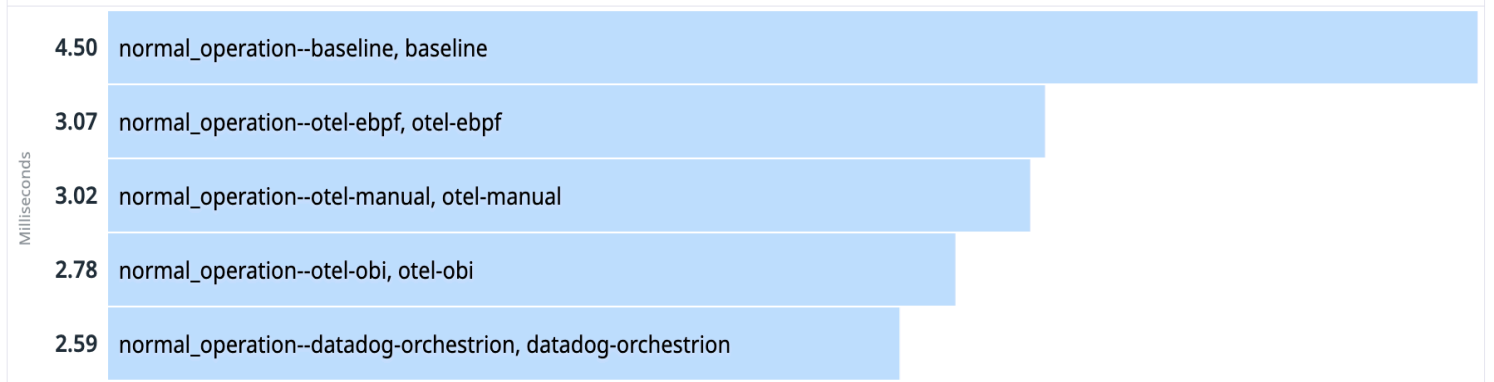
How do they compare?

Latency

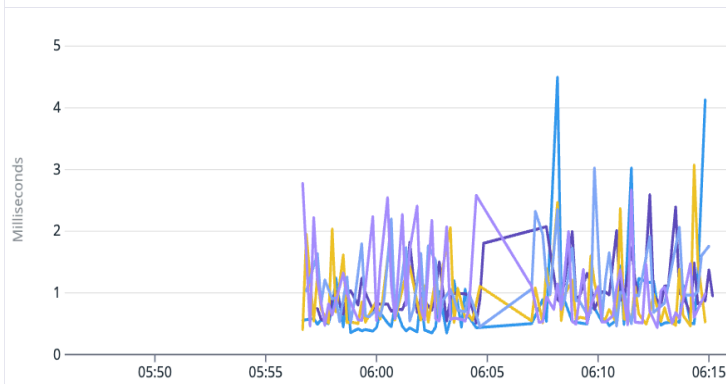
Avg Latency



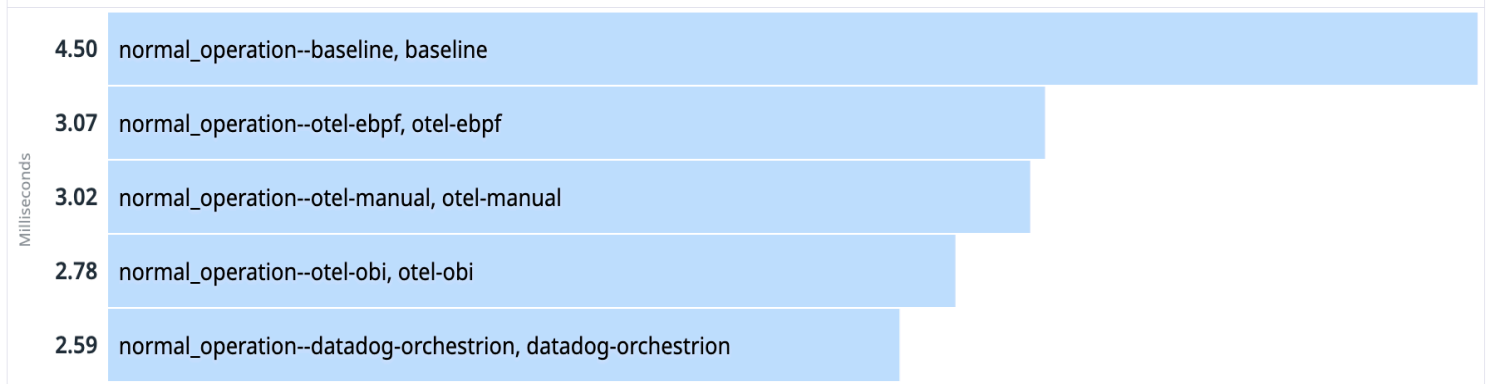
Latest Avg Latency



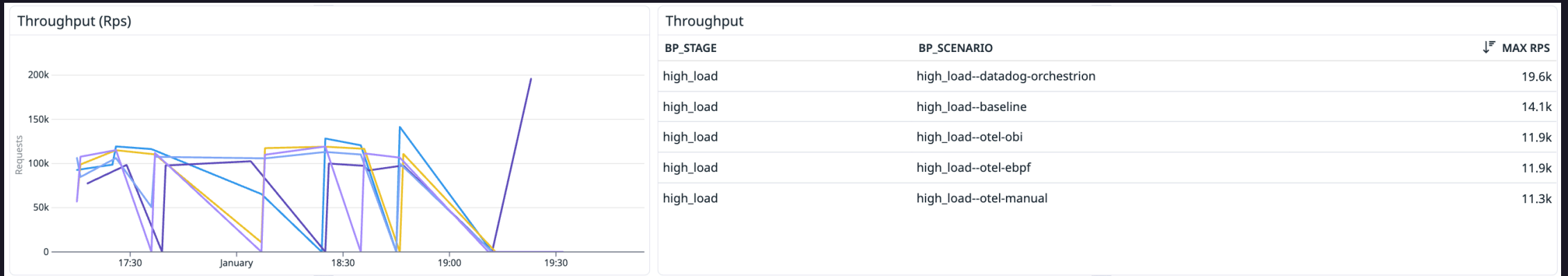
Max Latency



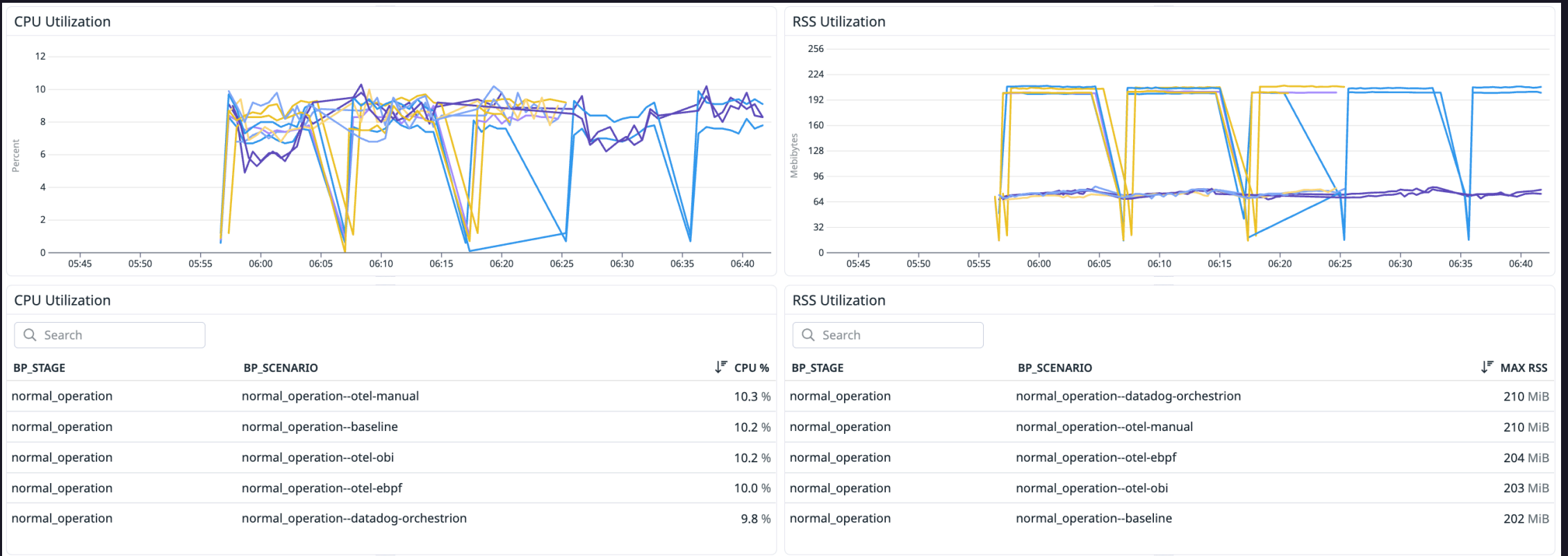
Latest Max Latency



Throughput



CPU / Memory



Host Metrics

Memory Usage						CPU Usage				
<input type="text" value="Search"/>						<input type="text" value="Search"/>				
BP_STAGE	BP_SCENARIO	↓ USED	CACHED	FREE	SHARED	BP_STAGE	BP_SCENARIO	IDLE CPU	↓ SYSTEM CPU	USER CPU
high_load	high_load--datadog-orchestrion	15.5 GiB	12.83 GiB	362 GiB	5.60 MiB	high_load	high_load--datadog-orchestrion	91.9 %	1.38 %	6.69 %
normal_operation	normal_operation--datadog-orchestrion	15.5 GiB	12.78 GiB	362 GiB	5.70 MiB	high_load	high_load--otel-ebpf	93.4 %	1.38 %	5.13 %
cleanup	cleanup--datadog-orchestrion	13.8 GiB	11.06 GiB	364 GiB	5.50 MiB	high_load	high_load--otel-obi	94.4 %	1.19 %	4.35 %
high_load	high_load--otel-ebpf	11.5 GiB	8.79 GiB	366 GiB	3.07 MiB	high_load	high_load--baseline	95.2 %	1.12 %	3.54 %
high_load	high_load--otel-obi	11.3 GiB	8.58 GiB	366 GiB	3.07 MiB	high_load	high_load--otel-manual	94.2 %	1.01 %	4.73 %
high_load	high_load--baseline	11.3 GiB	8.73 GiB	366 GiB	3.07 MiB	normal_operation	normal_operation--otel-ebpf	98.9 %	0.33 %	0.69 %
high_load	high_load--otel-manual	11.3 GiB	8.68 GiB	366 GiB	3.04 MiB	normal_operation	normal_operation--otel-obi	98.9 %	0.32 %	0.74 %
normal_operation	normal_operation--otel-ebpf	11.2 GiB	8.63 GiB	366 GiB	3.15 MiB	normal_operation	normal_operation--datadog-orchestrion	99.0 %	0.29 %	0.71 %
normal_operation	normal_operation--otel-manual	11.2 GiB	8.56 GiB	366 GiB	3.14 MiB	normal_operation	normal_operation--baseline	99.0 %	0.29 %	0.65 %

Approach	CPU	Memory (RSS)	Max Latency	Max Throughput
Baseline	10.2%	202 MiB	4.50 ms	3.1k req/sec
Manual	10.3% (+0.1%)	210 MiB (+8 MiB)	3.02 ms (-1.48 ms)	13.97k req/sec (+10.87k req/sec)
Auto (eBPF)	10% (-0.3%)	204 MiB (+2 MiB)	3.07 ms (-1.43 ms)	4.57k req/sec (+1.47k req/sec)
Auto (toolchain)	9.8% (-0.4%)	210 (+8 MiB)	2.59 ms (-1.91 ms)	27.8k req/sec (+24.7k req/sec)

Who wins?

Comparison Matrix

Approach	Performance	Stability	Security	Portability
Auto (eBPF)				
Auto (toolchain)				

Comparison Matrix

Approach	Performance	Stability	Security	Portability
Auto (eBPF)	⚠			
Auto (toolchain)	⚠			

Comparison Matrix

Approach	Performance	Stability	Security	Portability
Auto (eBPF)	⚠	⚠		
Auto (toolchain)	⚠	✅		

Comparison Matrix

Approach	Performance	Stability	Security	Portability
Auto (eBPF)	⚠️	⚠️	⚠️	
Auto (toolchain)	⚠️	✅	✅	

Comparison Matrix

Approach	Performance	Stability	Security	Portability
Auto (eBPF)	⚠	⚠	⚠	⚠
Auto (toolchain)	⚠	✅	✅	⚠

The Winner?

Approach	Performance	Stability	Security	Portability
Auto (eBPF)	⚠	⚠	⚠	⚠
Auto (toolchain)	⚠	✅	✅	⚠

It depends on your use case!

eBPF/OBI: Great for RUNTIME FLEXIBILITY

Orchestrion: Great for STABILITY AND SECURITY

The Future: Proof of Concepts

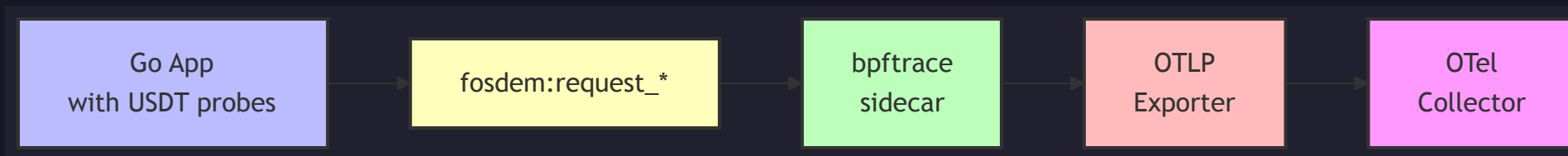
PoC: USDT + eBPF

User Statically-Defined Tracing

```
// Compile-time probes – zero overhead when disabled
probe.Fire("fosdem:request_start", requestID, timestamp)
// ... handle request ...
probe.Fire("fosdem:request_end", requestID, timestamp, duration)
```

- Uses `salp` library (Go bindings to libstapsdt)
- Attached dynamically via bpftrace sidecar
- **Zero runtime cost** when not tracing

PoC: USDT Architecture



USDT Ecosystem

libstapsdt enables runtime USDT probes for dynamic languages:

Language	Library	Status
C/C++	sys/sdt.h	Native support
Python	python-stapsdt	Production ready
Node.js	node-stapsdt	Production ready
Go	salp	Experimental
Ruby	ruby-stapsdt	In development

- DTrace 2.0.4+ supports `stapsdt` provider on Linux
- Works with bpftrace, bcc, perf, and SystemTap

PoC: Native USDT in Go Fork

```
import "runtime/trace/usdt"

func handleRequest(w http.ResponseWriter, r *http.Request) {
    usdt.Probe("myapp", "request_start")
    defer usdt.Probe1("myapp", "request_end", int32(w.StatusCode))
    // ... handle request
}
```

Stdlib auto-instrumented: `net/http`, `database/sql`, `crypto/tls`, `net`

PoC: Native USDT - Tooling

```
# List probes in binary
$ go tool usdt list ./myserver
PROVIDER    NAME                ADDRESS             ARGUMENTS
net_http    server_request_start 0x63296c           8@%rsi -8@%r8

# Generate bpftrace script
$ go tool usdt bpftrace ./myserver > trace.bt
$ sudo bpftrace trace.bt
```

github.com/kakkoyun/go/tree/poc_usdt

PoC: Frida Dynamic Instrumentation

Runtime function hooking via ptrace

```
// Attach to running Go binary
Interceptor.attach(Module.findExportByName(null,
    "net/http.serverHandler.ServeHTTP"), {
    onEnter: function(args) {
        send({ method: readGoString(args[1]) });
    }
});
```

- No code changes, works with any existing binary
- Requires `-gcflags="all=-N -l"` (disable optimizations)

Frida: Challenges with Go

Quarkslab's blog [Categories](#) [Tags](#) [Authors](#) [Q](#)


Table of contents

- Introduction
- Why Hook Golang Progra...
- What Is Program Hooking ...
- Hooking Go Using C and ...
 - Representing a Golang ...
 - Locating the Right Place...
 - Loading the Guest
 - Inserting the JUMP Stub...
 - Hook Insertion — ABI S...
 - Hook insertion — Stack ...
 - Hook Insertion — Mitiga...
 - Limitations of the Above...
- Why implement runtime h...
- Conclusion
- Resources

Let's Go into the rabbit hole (part 1) — the challenges of dynamically hooking Golang programs

Posted Tue 03 October 2023
Authors Mihail Kirov, Damien Aumaître
Category Containers
Tags container, cloud, Linux, go, golang, 2023

Golang is the most used programming language for developing cloud technologies. Tools such as *Kubernetes*, *Docker*, *Containerd* and *gVisor* are written in Go. Despite the fact that the code of these programs is open source, there is no way to analyze and extend their behavior dynamically without recompiling their code. Is this due to the complex internals of the language? In this blog post, we'll look into the challenges of developing and inserting runtime hooks in *Golang* programs.



Based on [Quarkslab research](#)

PoC: Flight Recording (Future Vision)

Always-on distributed tracing with Go's runtime

- Ring buffer with bounded memory
- W3C Trace Context propagation
- GODEBUG-based: `tracehttp=1`, `tracesql=1`, `tracenet=1`

```
import "runtime/trace/flight"  
  
flight.Enable(flight.HTTP | flight.SQL | flight.Net)  
defer flight.Flush() // Export on error/crash
```

PoC: Flight Recorder Go Fork

golang / go

Search Type to search

Code Issues 5k+ Pull requests 375 Discussions Actions Projects 5 Wiki Security Insights

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#). [Learn more about diff comparisons here](#).

base repository: golang/go base: master head repository: kakkoyun/go compare: poc_flight_recorder Swap

✓ **Able to merge.** These branches can be automatically merged.

Add a title

runtime/trace: add event filtering and distributed tracing support to FlightRecorder

Add a description

Write Preview

This change introduces event filtering capabilities and distributed tracing support for the runtime/trace package, enabling selective tracing of different event categories and W3C Trace Context propagation.

New features:

- EventFilter bitmask API for selective tracing (FilterCore, FilterHTTP, FilterSQL, FilterTLS, FilterNet, FilterCustom)
- GODEBUG environment variables (tracehttp, tracesql, tracetls, tracenet) for runtime configuration
- W3C Trace Context support with TraceContext struct for distributed tracing correlation
- HTTPSpan for HTTP client/server request tracing with version and error tracking
- SQLSpan for database query tracing with error codes
- TLSSpan for TLS handshake tracing with version and cipher suite info
- DNSSpan and ConnectSpan for network-level tracing
- URL and SQL sanitizers for sensitive data protection

Final thoughts

1. Instrumentation is helpful and important
2. Auto-instrumentation is EASY
3. What are YOU going to do next?

Thanks

Hannah Kim

hannahkm.github.io

linkedin.com/in/hannah-kim24

Kemal Akkoyun

@kakkoyun

github.com/kakkoyun

linkedin.com/in/kakkoyun



References

- [OpenTelemetry eBPF Instrumentation \(OBI\)](#).
- [OpenTelemetry Go Auto-Instrumentation](#)
- [Datadog Orchestrion](#)
- [salp - Go USDT library](#).
- [libstapsdt](#)
- [Go fork with USDT support](#)
- [Go fork with Flight Recorder](#)
- [bpftrace](#)
- [Quarkslab: Dynamically hooking Go programs](#)
- [USDT Tracing Across Runtimes \(Oracle\)](#).