

# Go compile time instrumentation

Przemysław Delewski

February 18, 2025

Warsaw, Poland

# Agenda

1. Short bio
2. Observability domain
3. OpenTelemetry
4. History of OpenTelemetry go  
compile time instrumentation
5. Demo
6. Some code walkthrough
7. Current status and future
8. Questions

# Short bio

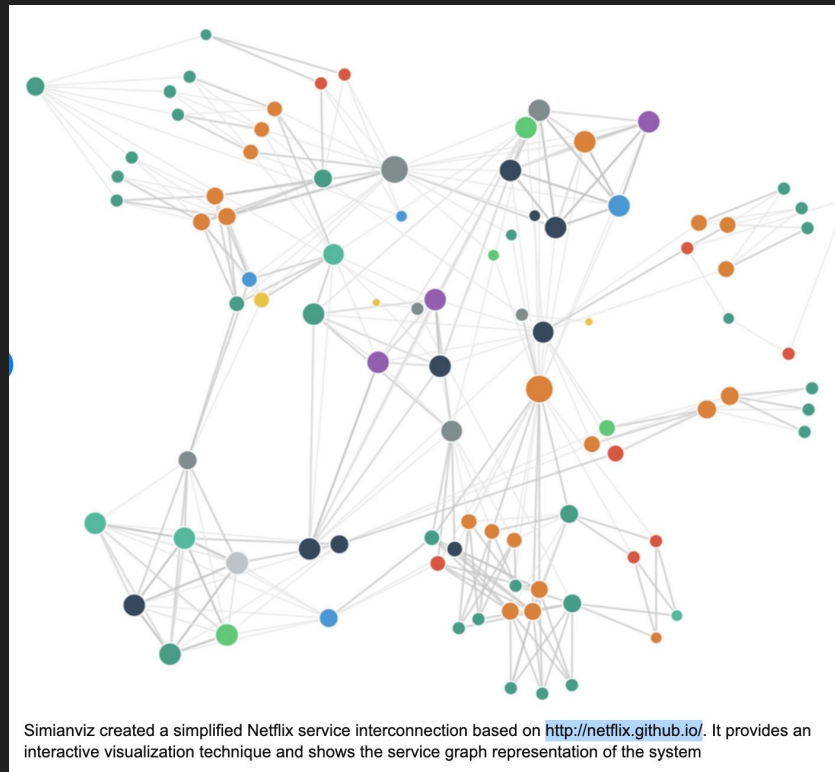
I'm a founding engineer at Quesma, where we work on a data gateway. Before that, I spent over a decade working on observability at Sumo Logic and Dynatrace in different roles, with a short break in navigation software at TomTom where I had the opportunity to work on brand new navigation stack. I'm also founding member of the OpenTelemetry Go compile-time instrumentation project

<https://github.com/open-telemetry/community/blob/main/projects/go-compile-instrumentation.md>



# Software complexity

Today's software, especially distributed systems, can be extremely complex. Understanding what is happening is crucial for solving problems.



# Observability - tool for complexity

[Observability](#) is the ability to understand the internal state of a system by examining its outputs. In the context of software, this means being able to understand the internal state of a system by examining its telemetry data, which includes traces, metrics, and logs.

To make a system observable, it must be [instrumented](#). That is, the code must emit [traces](#), [metrics](#), or [logs](#). The instrumented data must then be sent to an observability backend.

OpenTelemetry

# What is OpenTelemetry

1. Open source implementation of observability concepts
2. Framework and toolkit (set of libraries) designed to create, manage and consume telemetry signals

**OpenTelemetry** was officially introduced in **2019** as a **merger** of two existing observability projects:

1. **OpenTracing** (by the Cloud Native Computing Foundation - CNCF)
2. **OpenCensus** (originally developed by Google)



# Telemetry signals

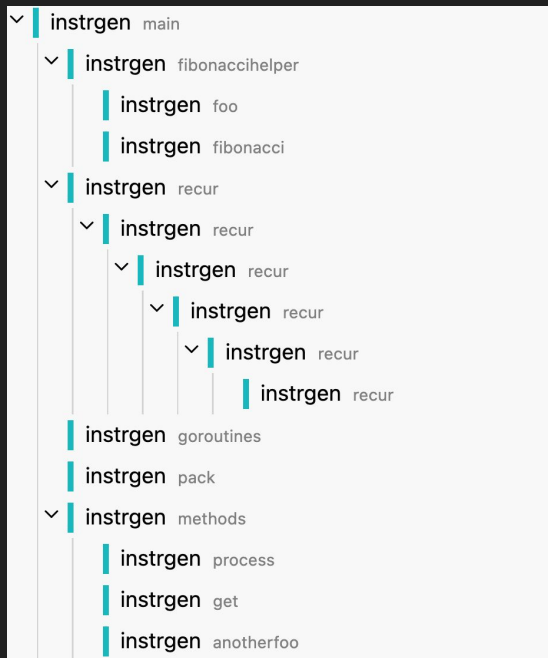
- Logs (additional info about behaviour)
- Metrics (measuring)
- Traces





# Trace

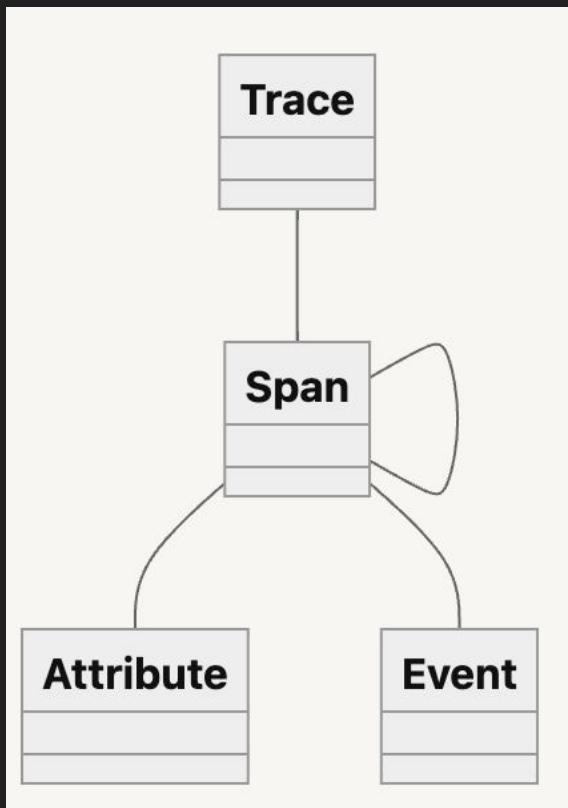
A trace represents the execution path of a request across multiple services.



```
{
  "name": "hello",
  "context": {
    "trace_id": "5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "051581bf3cb55c13"
  },
  "parent_id": null,
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114687Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "Guten Tag!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```

# Tracing concepts

- Trace represents an execution path
- Span represents step in an execution path
- Attribute and Event represents additional metadata information carried together with span



# History of OpenTelemetry go instrumentation

# Beginnings of go instrumentation

Everything started with two proposals:

- <https://github.com/open-telemetry/opentelemetry-go-instrumentation/issues/3>
- <https://github.com/open-telemetry/opentelemetry-go-instrumentation/issues/2>

Presented on the same GO SIG

Two repos:

- <https://github.com/open-telemetry/opentelemetry-go-contrib/>
- <https://github.com/open-telemetry/opentelemetry-go-instrumentation>

## Donation proposal: OpenTelemetry source level automatic Go instrumentation #3

Closed



pdelewski opened on Jul 8, 2022 · edited by pdelewski

Edits ...

### Description of project

This project adds OpenTelemetry instrumentation (<https://github.com/open-telemetry/opentelemetry-go>) to Go applications by automatically modifying their source code in similar way as compiler. It can instrument any golang project. It depends only on standard libraries and is platform agnostic.

Current github repo:

<https://github.com/SumoLogic-Labs/autotel>

## Donation proposal: OpenTelemetry Go Automatic Instrumentation #2

Closed



edenNFed opened on May 18, 2022 · edited by edenNFed

Edits ...

### Description of donated project

This project adds OpenTelemetry instrumentation to Go applications without having to modify their source code. Instrumentation is done by using [eBPF](#) uprobes.

Automatic instrumentation is available for a wide range of Go applications: Go version 1.12 and above, in addition to supporting stripped binaries (compiled with `go build -ldflags "-s -w"`)

Instrumented libraries follow the [OpenTelemetry specification](#) and semantic conventions to produce standard OpenTelemetry data.

GitHub repos:

<https://github.com/keyval-dev/opentelemetry-go-instrumentation>

<https://github.com/keyval-dev/offsets-tracker> (used by this project, can also be donated if needed)

Go features

# Go provides excellent tooling for code analysis and manipulation

- All tools needed for source level manipulation are part of standard library

```
go/types  
go/constant  
go/parser  
go/ast  
go/scanner  
go/token
```

# Go AST Traversal

```
func main() {  
    // parse file  
    fset := token.NewFileSet()  
    node, err := parser.ParseFile(fset, filename: "main.go", src: nil, parser.ParseComments)  
    if err != nil {  
        log.Fatal(err)  
    }  
  
    ast.Inspect(node, func(n ast.Node) bool {  
        fn, ok := n.(*ast.FuncDecl)  
        if ok {  
            fmt.Printf(format: "function declaration found on line %d: \n\t%s\n",  
                fset.Position(fn.Pos()).Line, fn.Name.Name)  
            fmt.Println()  
        }  
        return true  
    })  
}
```

# Instrumentation



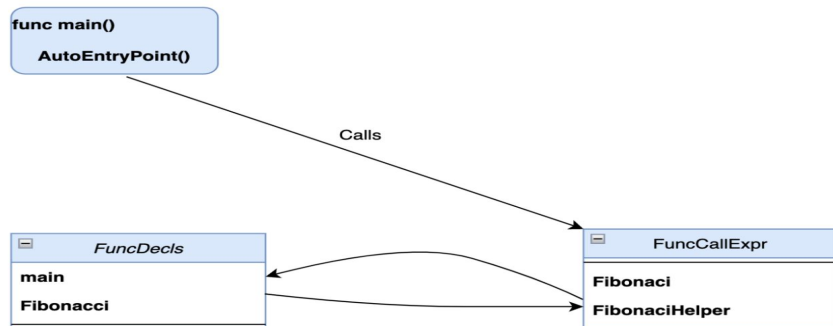
# Instrgen first instrumentation approach

## OpenTelemetry Go Source Automatic Instrumentation - How it works

`instrgen` adds OpenTelemetry instrumentation to source code by directly modifying it. It uses the AST (Abstract Syntax Tree) representation of the code to determine its operational flow and injects necessary OpenTelemetry functionality into the AST.

The AST modification algorithm is the following:

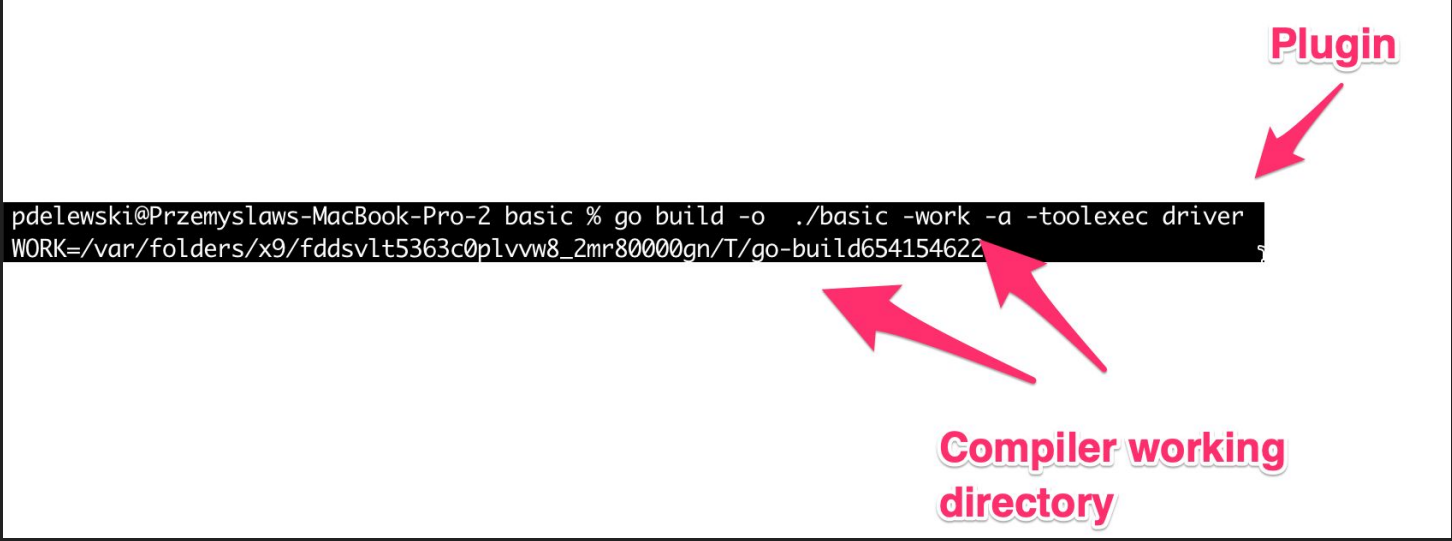
1. Search for the entry point: a function definition with `AutoEntryPoint()`.
2. Build the call graph. Traverse all calls from the entry point through all function definitions.
3. Inject OpenTelemetry instrumentation into functions bodies.
4. Context propagation. Adding an additional context parameter to all function declarations and function call expressions that are visible (it will not add a context argument to call expressions if they are not reachable from the entry point).



! Problematic from context propagation perspective, might give incorrect results and compilation failures.

# Instrgen second approach

- Utilizing toolexec (<https://github.com/open-telemetry/opentelemetry-go-contrib/pull/4058>)



A terminal window showing a Go build command. Three red arrows point from text labels to parts of the command: one from 'Plugin' to '-toolexec', and two from 'Compiler working directory' to the '-work' flag and the directory path.

```
pdelewski@Przemyslaw's-MacBook-Pro-2 basic % go build -o ./basic -work -a -toolexec driver  
WORK=/var/folders/x9/fddsvlt5363c0plvw8_2mr80000gn/T/go-build654154622
```


Plugin

Compiler working  
directory

# Runtime hooking

```
go.dev/src/runtime/proc.go

5026         if mainStarted {
5027             wakep()
5028         }
5029     })
5030 }
5031
5032 // Create a new g in state _Grunnable (or _Gwaiting if parked is true), starting at fn.
5033 // callerpc is the address of the go statement that created this. The caller is responsible
5034 // for adding the new g to the scheduler. If parked is true, waitreason must be non-zero.
5035 func newproc1(fn *funcval, callerg *g, callerpc uintptr, parked bool, waitreason waitReason) *g {
5036     if fn == nil {
5037         fatal("go of nil func value")
5038     }
5039
5040     mp := acquirem() // disable preemption because we hold M and P in local vars.
5041     pp := mp.p.ptr()
5042     newg := gfget(pp)
5043     if newg == nil {
5044         newg = malg(stackMin)
5045         casgstatus(newg, _Gidle, _Gdead)
5046         allgadd(newg) // publishes with a g->status of Gdead so GC scanner doesn't look at uninitialized stack.
5047     }
5048     if newg.stack.hi == 0 {
5049         throw("newproc1: newg missing stack")
5050     }
5051
5052     if readgstatus(newg) != _Gdead {
5053         throw("newproc1: new g is not Gdead")
5054     }
5055 }
```



# Compilation process internals

<https://raw.githubusercontent.com/pdelewski/toolexec/refs/heads/dump-args/main.go>

```
/opt/homebrew/Cellar/go/1.22.5/libexec/pkg/tool/darwin_arm64/compile
-o
/var/folders/x9/fddsvlt5363c0plvw8_2mr80000gn/T/go-build2085540145/b001/_pkg_.a
-trimpath
/var/folders/x9/fddsvlt5363c0plvw8_2mr80000gn/T/go-build2085540145/b001=>
-p
main
-lang=go1.19
-complete
-buildid
1ho7jWjcAkI15K5xGJoi/1ho7jWjcAkI15K5xGJoi
-goversion
go1.22.5
-c=4
-shared
-nolocalimports
-importcfg
/var/folders/x9/fddsvlt5363c0plvw8_2mr80000gn/T/go-build2085540145/b001/importcfg
-pack
/Users/pdelewski/Projects/toolexec/main.go
```

**Package name** (points to `main`)

**Working directory** (points to `/var/folders/x9/fddsvlt5363c0plvw8_2mr80000gn/T/go-build2085540145/b001=>`)

**Source files** (points to `/Users/pdelewski/Projects/toolexec/main.go`)

```
package main

import (
    "fmt"
    "golang.org/x/sys/unix"
    "os"
    "os/exec"
)

func executePass(args []string) { 1 usage
    path := args[0]
    args = args[1:]
    cmd := exec.Command(path, args...)
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    if e := cmd.Run(); e != nil {
        fmt.Println(e)
    }
}

func lockFile(file *os.File) error { 1 usage
    return unix.Flock(int(file.Fd()), unix.LOCK_EX) // Exclusive lock
}

func unlockFile(file *os.File) error { 1 usage
    return unix.Flock(int(file.Fd()), unix.LOCK_UN) // Unlock
}

func compile(args []string, f *os.File) { 1 usage
    for _, a := range args {
        lockFile(f)
        f.WriteString(a)
        f.WriteString(" ")
        unlockFile(f)
    }
    executePass(args[0:])
}

func main() {
    f, _ := os.OpenFile("args", os.O_APPEND|os.O_CREATE|os.O_WRONLY, perm: 0644)
    args := os.Args[1:]
    compile(args, f)
}
```

Demo

# Code walkthrough (PackageRewriter)

```
// PackageRewriter interface does actual input package
// rewriting according to specific criteria.
type PackageRewriter interface { 1usage
    // ID Dumps rewriter id.
    Id() string
    // Inject tells whether package should be rewritten.
    Inject(pkg string, filepath string) bool
    // ReplaceSource decides whether input sources should be replaced
    // or all rewriting work should be done in temporary location.
    ReplaceSource(pkg string, filePath string) bool
    // Rewrite does actual package rewriting.
    Rewrite(pkg string, file *ast.File, fset *token.FileSet, trace *os.File)
    // WriteExtraFiles generate additional files that will be linked
    // together to input package.
    // Additional files have to be returned as array of file names.
    WriteExtraFiles(pkg string, destPath string) []string
}
```

Current status and future

# Current status and future

Two new donations from alibaba and datadog:

- <https://github.com/open-telemetry/community/issues/1961>
- <https://github.com/open-telemetry/community/issues/2497>

For documentation look:

- <https://www.datadoghq.com/blog/go-instrumentation-orchestrion/>
- <https://github.com/alibaba/opentelemetry-go-auto-instrumentation>



# New SIG (Special Interests Group)

- <https://github.com/open-telemetry/community/pull/2490>

New repo

<https://github.com/open-telemetry/opentelemetry-go-compile-instrumentation/>

Blog post

<https://opentelemetry.io/blog/2025/go-compile-time-instrumentation/>

Thank you!  
Go compile time  
instrumentation

Przemysław Delewski <[pdelewski@quesma.com](mailto:pdelewski@quesma.com)>  
<[przemyslaw.delewski@gmail.com](mailto:przemyslaw.delewski@gmail.com)>

# Questions