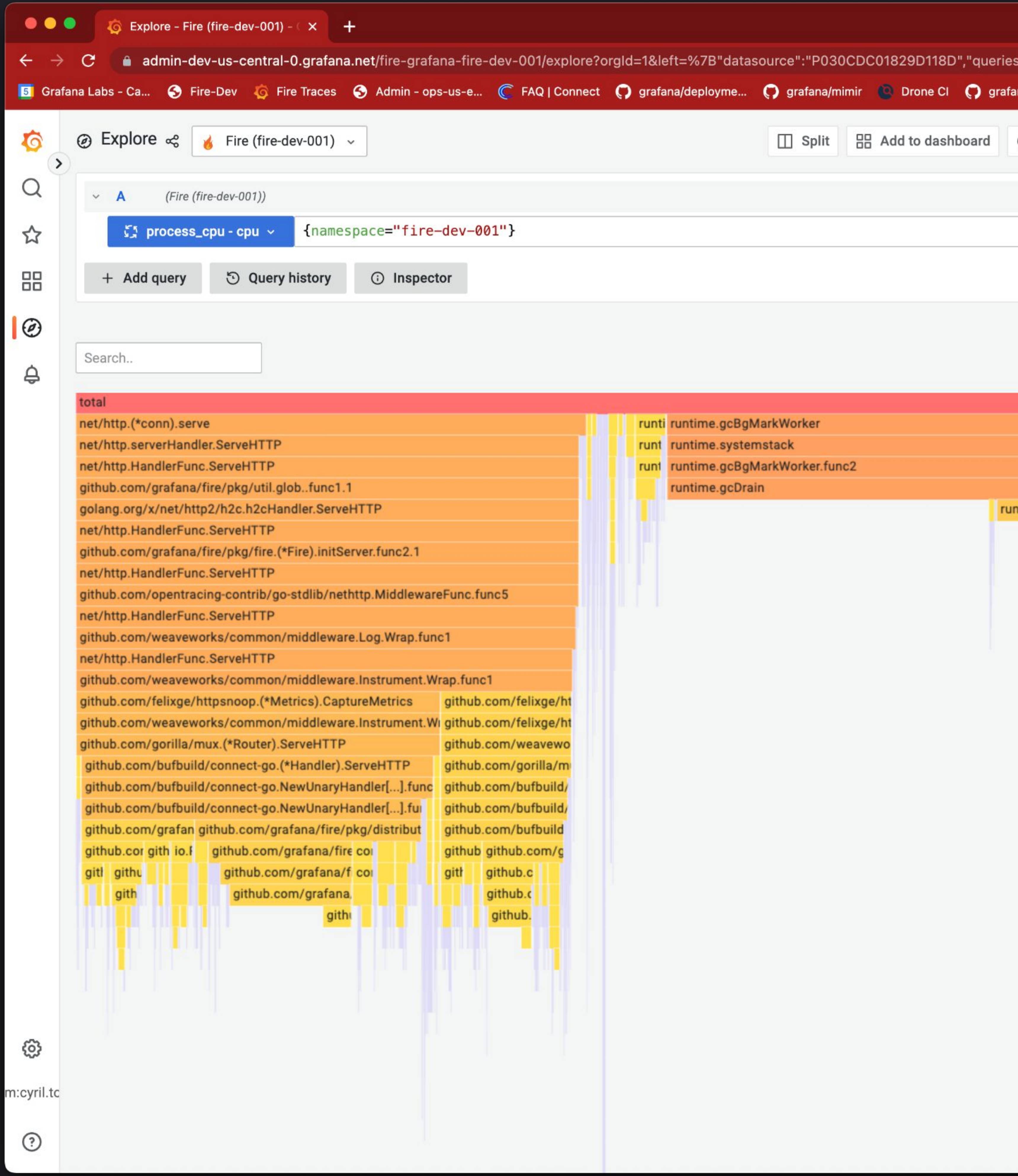
Adopting continuous-profiling:

Understand how your code utilizes cpu/memory



Introduction into continuous-profiling and how it can help you writing more efficient code

2023-02-05

FOSDEM 2023 - Monitoring and Observability devroom

Christian Simon

Software Engineer at Grafana Labs

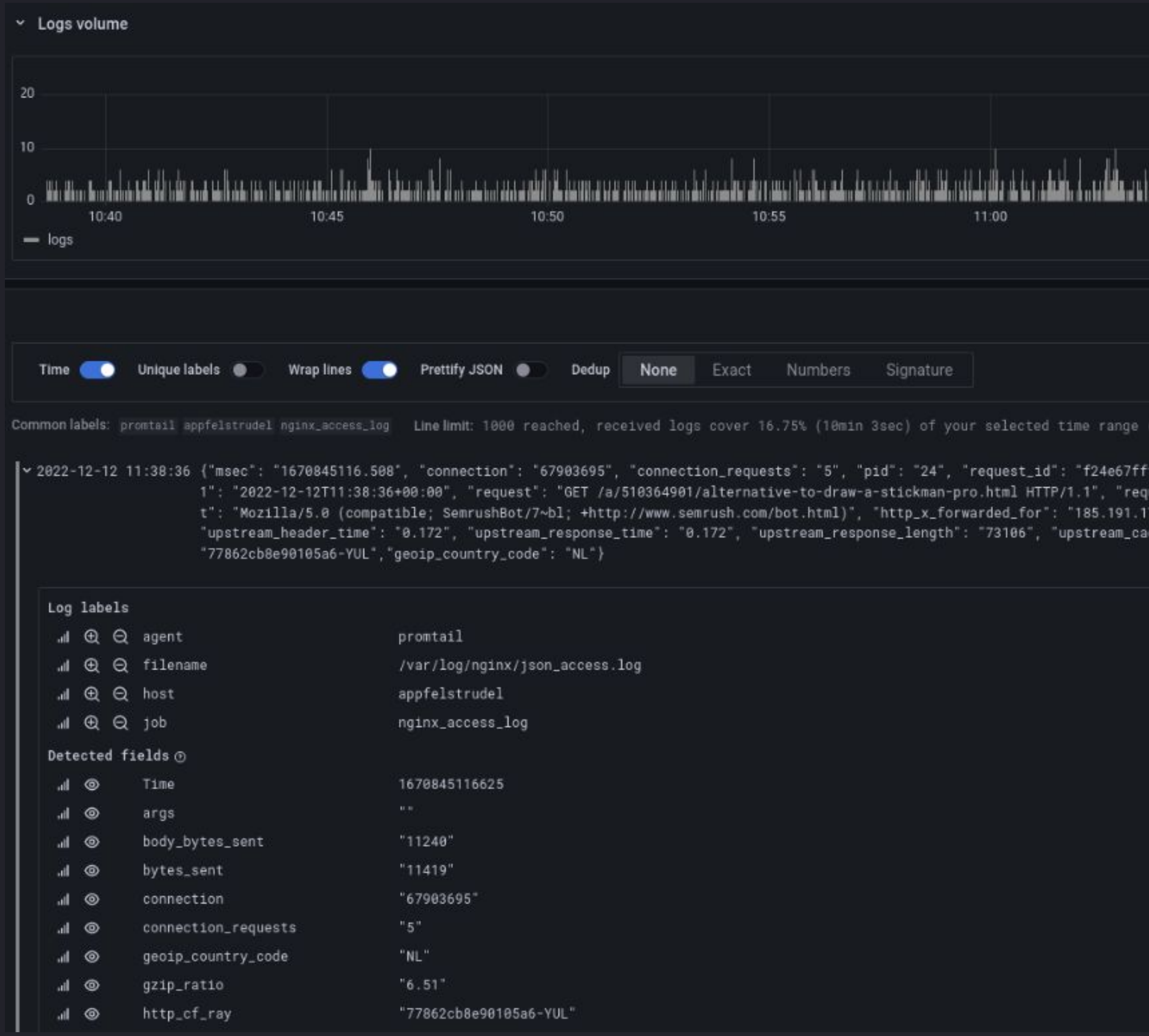
Working on observability databases (Loki, Mimir, Phlare)

Observability

- **Introspect your applications/infrastructure running in production**
- **Objective way of looking at state between teams**
- **Goals**
    - **Avoid negative user experience and ideally catch problems before they become user facing**
    - **Reduce the mean time to repair**
    - **Aid in root cause analysis**

Observability using Logs

- No specific instrumentation or application changes need, as most applications already support logging in some form
- Challenges
  - Aggregations across many log lines can become quite expensive
  - Different log formats and sometimes it might be hard to correlate information
  - Signal to noise ratio can hide important information



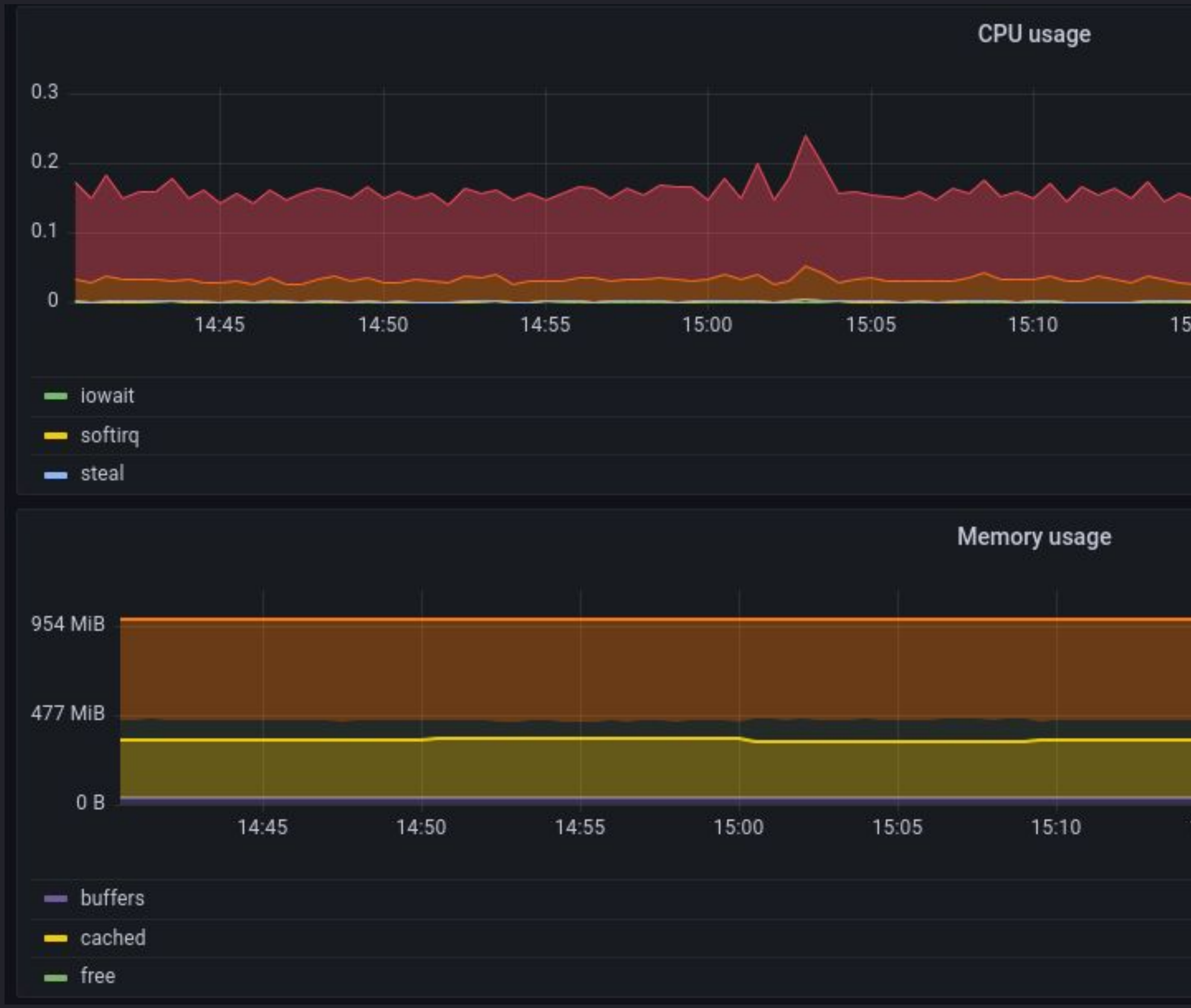
Common labels: promtail appfelstrudel nginx_access_log   Line limit: 1000 reached, received logs cover 16.75% (10min 3sec) of your selected time range

```
2022-12-12 11:38:36 {"msec": "1670845116.508", "connection": "67903695", "connection_requests": "5", "pid": "24", "request_id": "f24e67ff
                    1": "2022-12-12T11:38:36+00:00", "request": "GET /a/510364901/alternative-to-draw-a-stickman-pro.html HTTP/1.1", "req
                    t": "Mozilla/5.0 (compatible; SemrushBot/7~bl; +http://www.semrush.com/bot.html)", "http_x_forwarded_for": "185.191.1
                    "upstream_header_time": "0.172", "upstream_response_time": "0.172", "upstream_response_length": "73106", "upstream_ca
                    "77862cb8e90105a6-YUL","geoip_country_code": "NL"}
```

```
Log labels
agent                     promtail
filename                  /var/log/nginx/json_access.log
host                      appfelstrudel
job                       nginx_access_log
Detected fields
Time                      1670845116625
args                      ""
body_bytes_sent           "11240"
bytes_sent                "11419"
connection                "67903695"
connection_requests       "5"
geoip_country_code        "NL"
gzip_ratio                "6.51"
http_cf_ray               "77862cb8e90105a6-YUL"
```

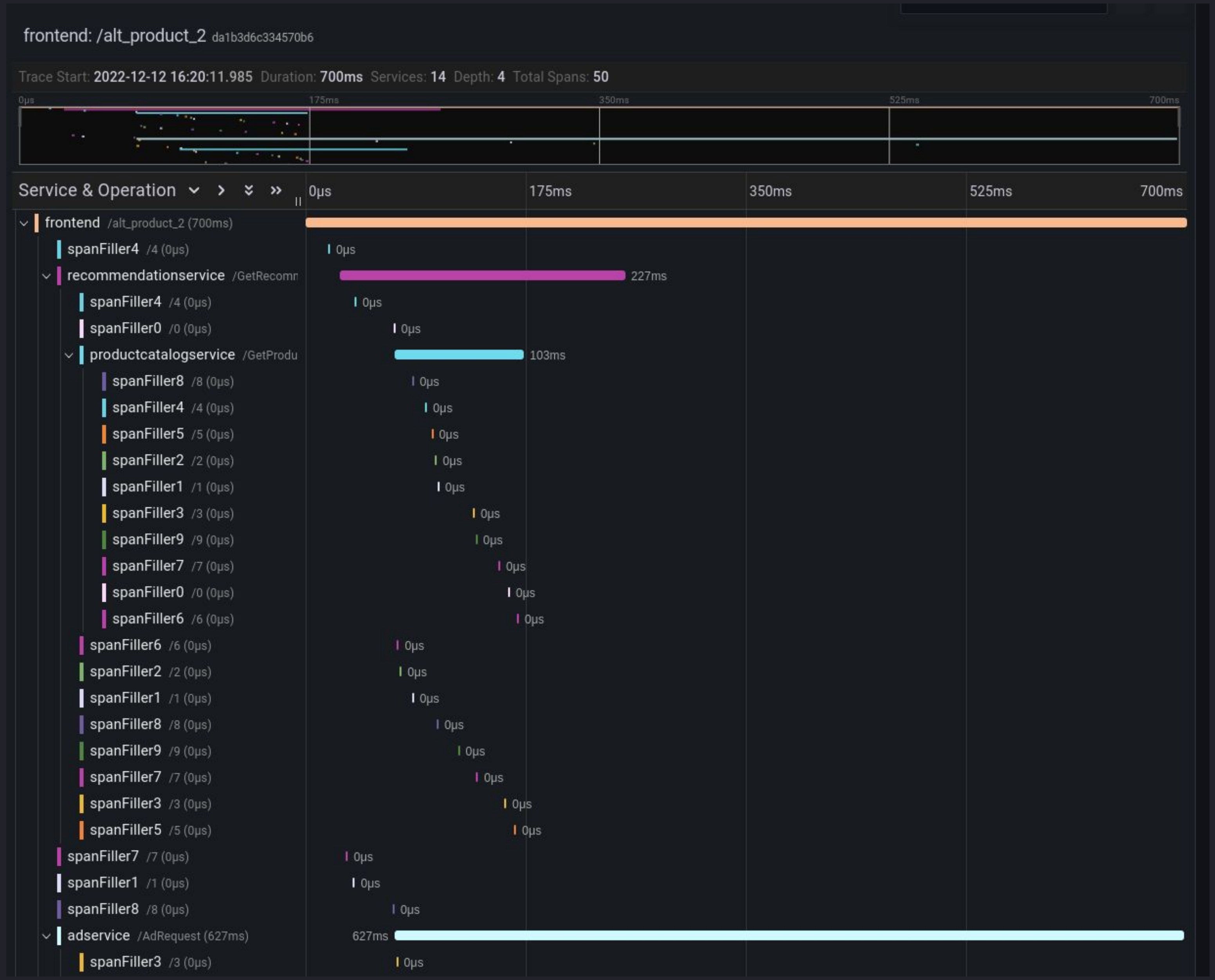
Observability using Metrics

- Numerical data measured over a time
- Typical metrics measured for web services:
    - RED method (Rate, Error rate and Duration of requests)
- A lot of values can be stored and aggregated efficiently
- Challenges
    - Applications require instrumentation, so it is important to know beforehand what to measure



Observability using Traces

- Introspect how requests, which are dependent on each other are flowing through a distributed system.
- Go from metrics to traces using exemplars
- Challenges
  - Not all requests might be sampled



frontend: /alt_product_2 da1b3d6c334570b6

Trace Start: **2022-12-12 16:20:11.985** Duration: **700ms** Services: **14** Depth: **4** Total Spans: **50**

- frontend /alt_product_2 (700ms)
  - spanFiller4 /4 (0µs) 0µs
  - recommendationservice /GetRecomm 227ms
    - spanFiller4 /4 (0µs) 0µs
    - spanFiller0 /0 (0µs) 0µs
    - productcatalogservice /GetProdu 103ms
      - spanFiller8 /8 (0µs) 0µs
      - spanFiller4 /4 (0µs) 0µs
      - spanFiller5 /5 (0µs) 0µs
      - spanFiller2 /2 (0µs) 0µs
      - spanFiller1 /1 (0µs) 0µs
      - spanFiller3 /3 (0µs) 0µs
      - spanFiller9 /9 (0µs) 0µs
      - spanFiller7 /7 (0µs) 0µs
      - spanFiller0 /0 (0µs) 0µs
      - spanFiller6 /6 (0µs) 0µs
    - spanFiller6 /6 (0µs) 0µs
    - spanFiller2 /2 (0µs) 0µs
    - spanFiller1 /1 (0µs) 0µs
    - spanFiller8 /8 (0µs) 0µs
    - spanFiller9 /9 (0µs) 0µs
    - spanFiller7 /7 (0µs) 0µs
    - spanFiller3 /3 (0µs) 0µs
    - spanFiller5 /5 (0µs) 0µs
  - spanFiller7 /7 (0µs) 0µs
  - spanFiller1 /1 (0µs) 0µs
  - spanFiller8 /8 (0µs) 0µs
  - adservice /AdRequest (627ms) 627ms
    - spanFiller3 /3 (0µs) 0µs

Let’s go through an example

- User raises a ticket because during check out they saw a timeout
- Looking up the trace ID in the logs reveals that tracing show the location service has been timing out
- Looking at the metrics for all location service replicas, we can see 5% of the requests time out
- Next steps
  - Scale up replicas for location service 💸💸💸💸
  - Optimise location service

Observability using Profiles

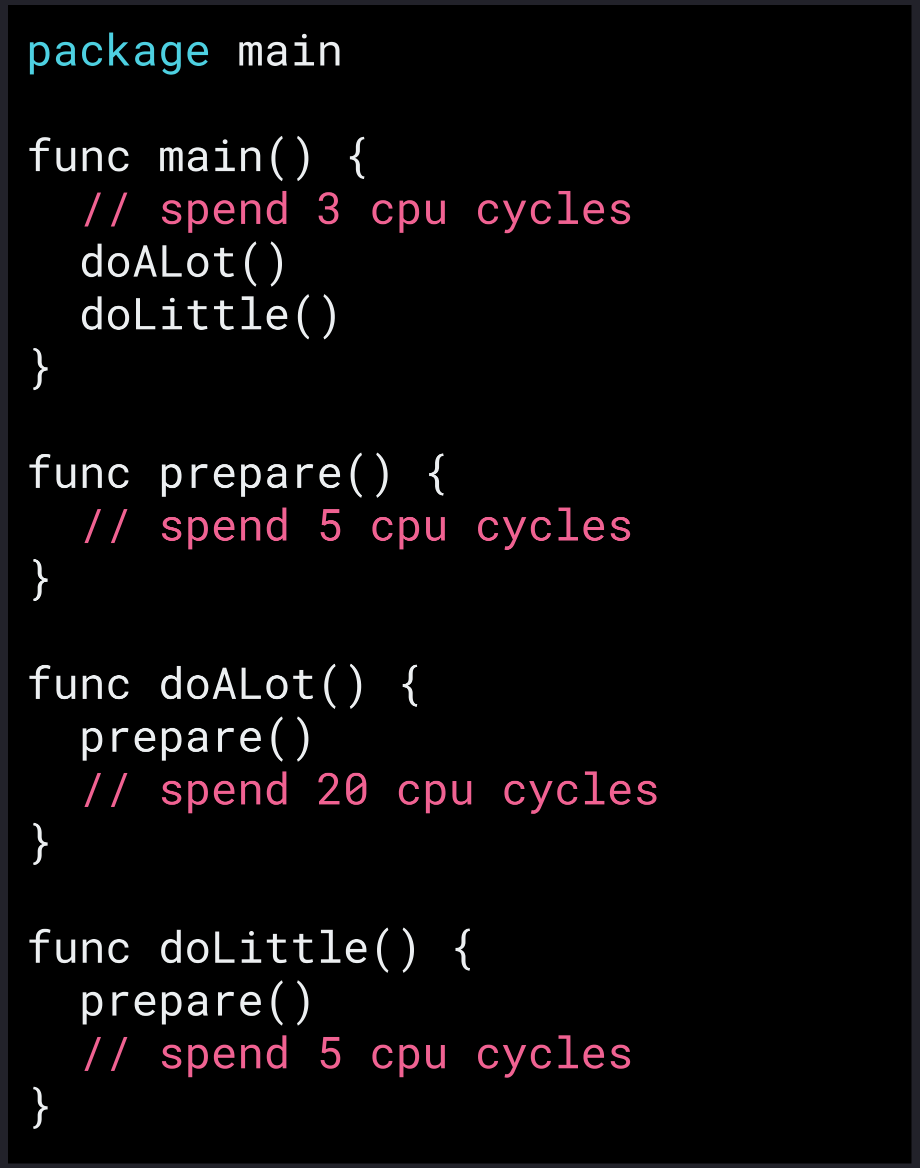
Profiling shows the resource usage of an application

- Profiling information serves **to aid program optimization**, and more specifically, **performance engineering**. ⚡
- Profiling can help reduce and understand workload **cost** 💰(TCO), improve service **latency** and fixes applications problems (OOM)🔥
- Multiple types of profiling data
  - Space (memory): How much memory my application uses or allocates ? And where ?
  - Time (complexity): The frequency and duration of function calls. Where is my application spending most of CPU time ?
  - And more.... threads, synchronization....

What is measured in a Profile?

```go
package main

func main() {
  // work
  doALot()
  doLittle()
}

func prepare() {
  // work
}

func doALot() {
  prepare()
  // work
}

func doLittle() {
  prepare()
  // work
}
```

What is measured in a profile? Time on CPU



```
package main

func main() {
  // spend 3 cpu cycles
  doALot()
  doLittle()
}

func prepare() {
  // spend 5 cpu cycles
}

func doALot() {
  prepare()
  // spend 20 cpu cycles
}

func doLittle() {
  prepare()
  // spend 5 cpu cycles
}
```

Each measurement gets recorded on a stack-trace level

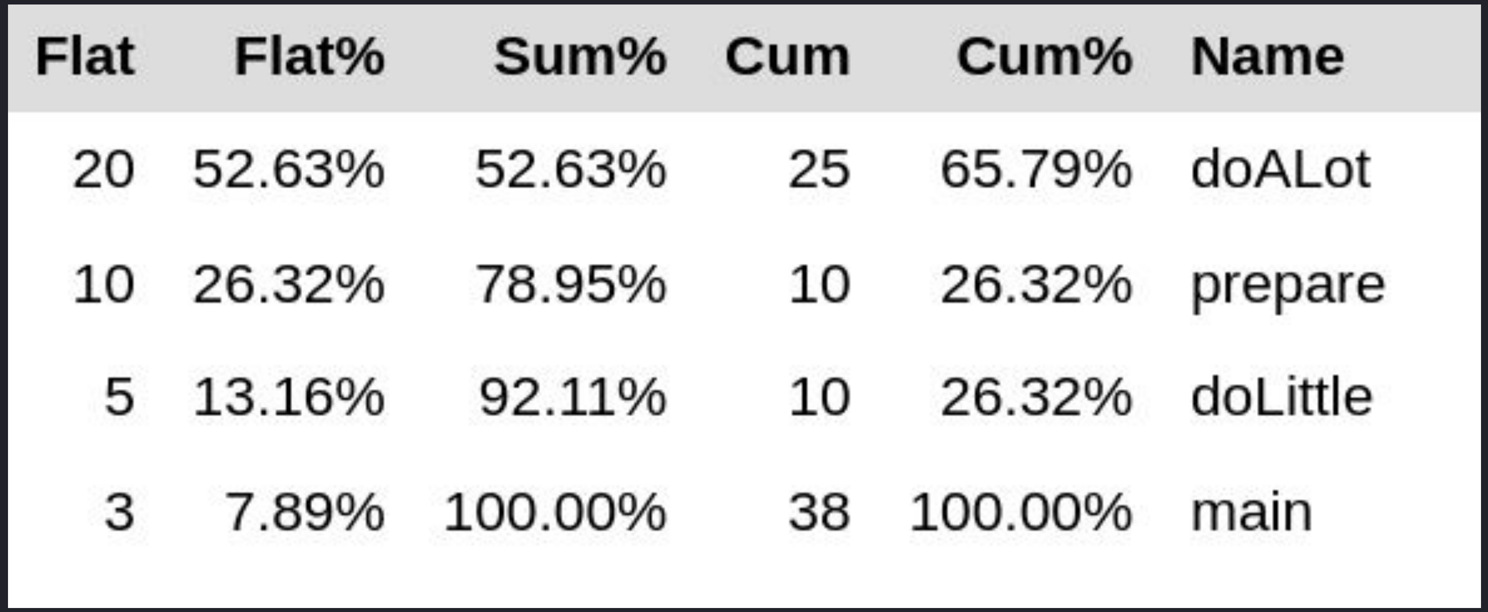
`main()`	3
`main() > doALot() > prepare()`	5
`main() > doALot()`	20
`main() > doLittle() > prepare()`	5
`main() > doLittle()`	5

Visualization of Profiles (try it yourself: https://pprof.me/b9d077f)

```
package main

func main() {
  // spend 3 cpu cycles
  doALot()
  doLittle()
}

func prepare() {
  // spend 5 cpu cycles
}

func doALot() {
  prepare()
  // spend 20 cpu cycles
}

func doLittle() {
  prepare()
  // spend 5 cpu cycles
}
```

TOP table

- Flat: Consumption by the function only
- Cumulative: Consumption by the function and its descendants
- Sum%: Based on the order of the table how much of the total measured consumption is covered by the row



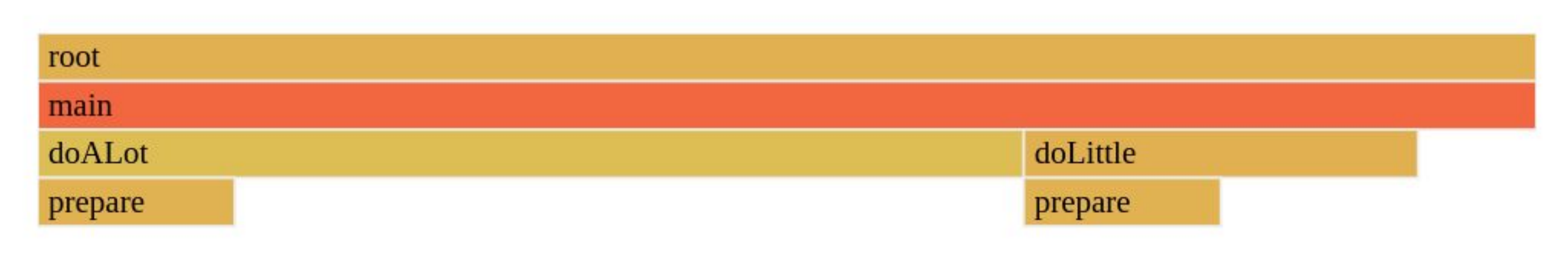
Flat	Flat%	Sum%	Cum	Cum%	Name
20	52.63%	52.63%	25	65.79%	doALot
10	26.32%	78.95%	10	26.32%	prepare
5	13.16%	92.11%	10	26.32%	doLittle
3	7.89%	100.00%	38	100.00%	main

Visualization of Profiles (try it yourself: https://pprof.me/b9d077f)

```go
package main

func main() {
  // spend 3 cpu cycles
  doALot()
  doLittle()
}

func prepare() {
  // spend 5 cpu cycles
}

func doALot() {
  prepare()
  // spend 20 cpu cycles
}

func doLittle() {
  prepare()
  // spend 5 cpu cycles
}
```

Flamegraph

- Whole width represent the total resources used (over the whole measurement duration)
- Ability to spot higher usage nodes
- Colours are random



How to gather a profile?

(further read: eBPF pros/cons)

- Instrumenting the code base
    - Tooling and formats depending on each language ecosystem
    - Access to more detailed runtime information
- eBPF based collection
    - No insights into runtime information
      (so better suited for compiled languages)
    - Doesn’t require instrumentation of application

How to gather a profile? Let’s take a look at Go

- Standard library includes CPU, Memory, Goroutine, Mutex and Block resources
- Provides profiles using a HTTP interface
    - Profiling data is returned using protobuf definition
- Data meant to be consumed by the pprof CLI

    -
      ```
      # Get a CPU profile over the last 2 seconds
      $ pprof "http://localhost:6060/debug/pprof/profile?seconds=2"
      # Get the heap memory allocations
      $ pprof "http://localhost:6060/debug/pprof/allocs"
      ```
    - Common to use the `-http` parameter to view profiles using the web interface
- Find more on Profiling in Go on https://pkg.go.dev/runtime/pprof#Profile

Instrumentation of Go code

```go
package main

import (
    "log"
    "net/http"
    _ "net/http/pprof"
    "time"
)

func main() {
    go func() {
        log.Println(http.ListenAndServe("localhost:6060", nil))
    }()
    // spend 3 cpu cycles
    doALot()
    doLittle()
}

[...]
```

The challenges with deterministic profiling

- Significant runtime overhead
- Hard to recreate problematic scenarios
- Even harder in distributed systems / microservices
- Large volume of profiling data

Continuous profiling

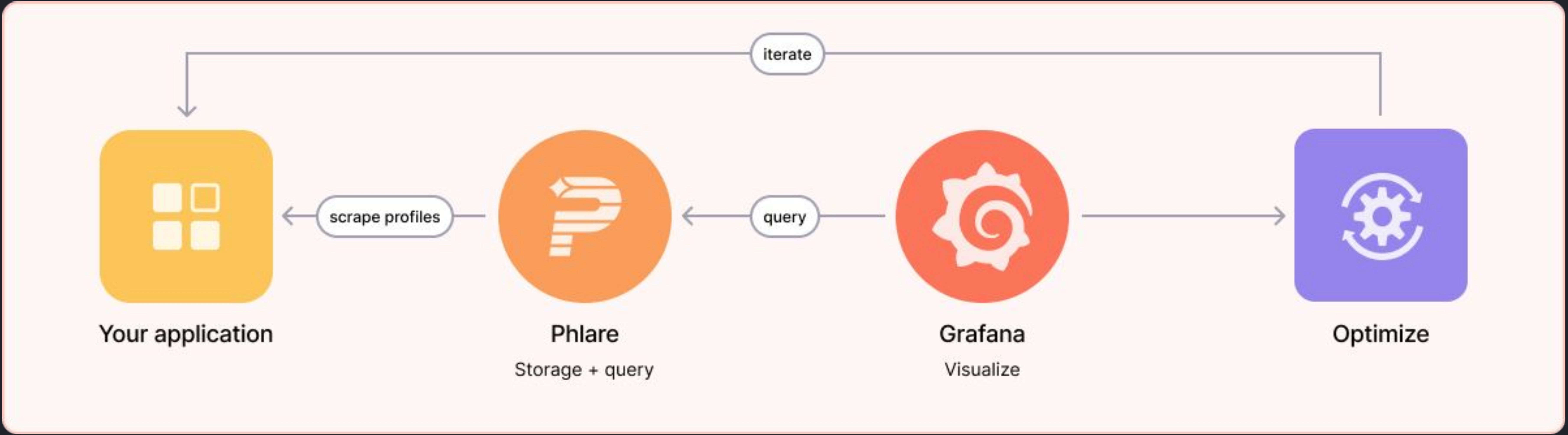
- Championed by Google in production
- Sampling the call stack
- Sampling ⇒ very low overhead
- “Always on” in production

A typical continuous profiling workflow

Store / Query / Visualize

- pprof CLI and profile collection at scale can become tedious
- Multiple solutions exist
- Profiling databases can simplify the workflow
  - CNCF Pixie
  - Pyroscope
  - Polarsignal Parca
  - Grafana Phlare

Demo time 🤞


iterate
scrape profiles
query
Your application
Phlare
Storage + query
Grafana
Visualize
Optimize

Profile guided optimizations

- “Optimize” step of the workflow typically is involves a human reasoning about profiling data and the code
- Compilers can also do Profile guided optimization (PGO)
- Having production/real world profiling information allows to improve decision making at compile time
- Go 1.20 includes PGO in public review, which improves the inlining decision making

https://github.com/simonswine/demo-pprof

#phlare on https://grafana.slack.com/

📖 https://grafana.com/docs/phlare/latest/

🛝 https://play-phlare.grafana.org/