coroot :~#

Zero-instrumentation observability based on eBPF

March 18, 2025

Peter Zaitsev,
Co-Founder at Coroot

DevOpsDays
Chicago


DevOpsDays
Chicago 2025
https://devopsdays.org/

My Perspective

- Involved in Open Source since late 1990s
- Early Staff member of MySQL AB
- Founder, until recently CEO at Percona
- Co-Founder Altinity, FerretDB, Coroot
- Investor, Advisor, Mentor in Open Source Businesses
- Open Source Advocate

PERCONA

What is Observability ?

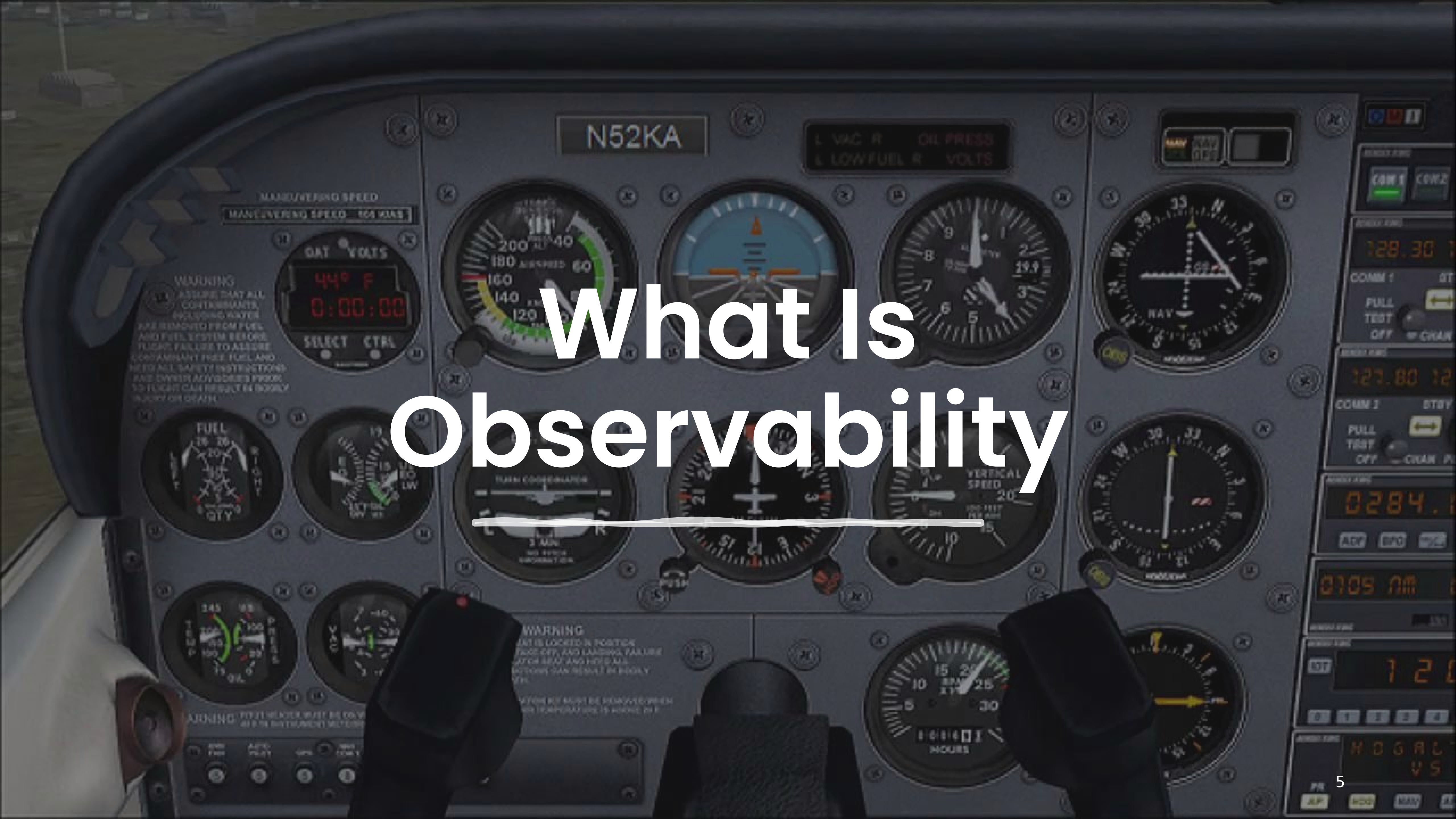
You

what is observability

ChatGPT

Observability is a concept in systems theory and engineering that refers to the ability to understand the internal state of a system based on its external outputs. In simpler terms, it's about how well you can observe and understand what's happening inside a system just by looking at its behavior from the outside.

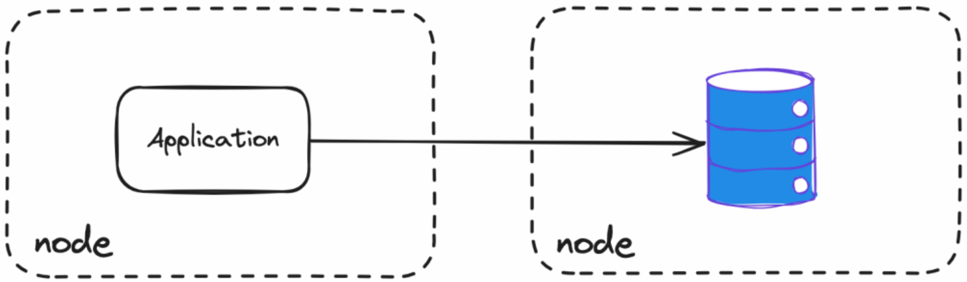
In software engineering and computer science, observability is particularly important in the context of complex distributed systems, where traditional debugging techniques might not be sufficient. It involves collecting and analyzing data from various sources such as logs, metrics, traces, and events to gain insights into the performance, health, and behavior of the system.



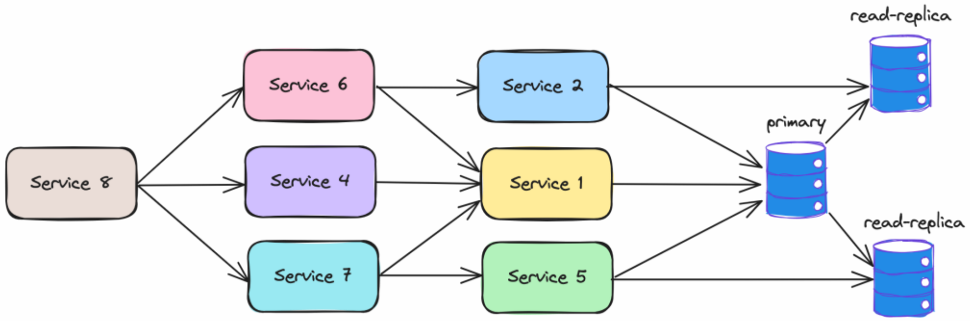
What Is Observability

Evolution of the Applications

2000s



2020s



Why Observability ?

- Availability
- Performance
- Cost Management
- Security

First Three Usually Come Together

Availability

Performance

Cost

4 Pillars of Observability

- Metrics
- Logs
- Tracing
- Profiling

What is the Most Useful?

Peter Zaitsev • You
Entrepreneur | Driving Success with MySQL, MariaDB, MongoDB & Post...
1d •

What is most important pillar of Observability ? #observability

What is most important pillar of #observability

You can see how people vote. Learn more

Metrics	51%
Traces	13%
Logs	26%
Profiling	10%

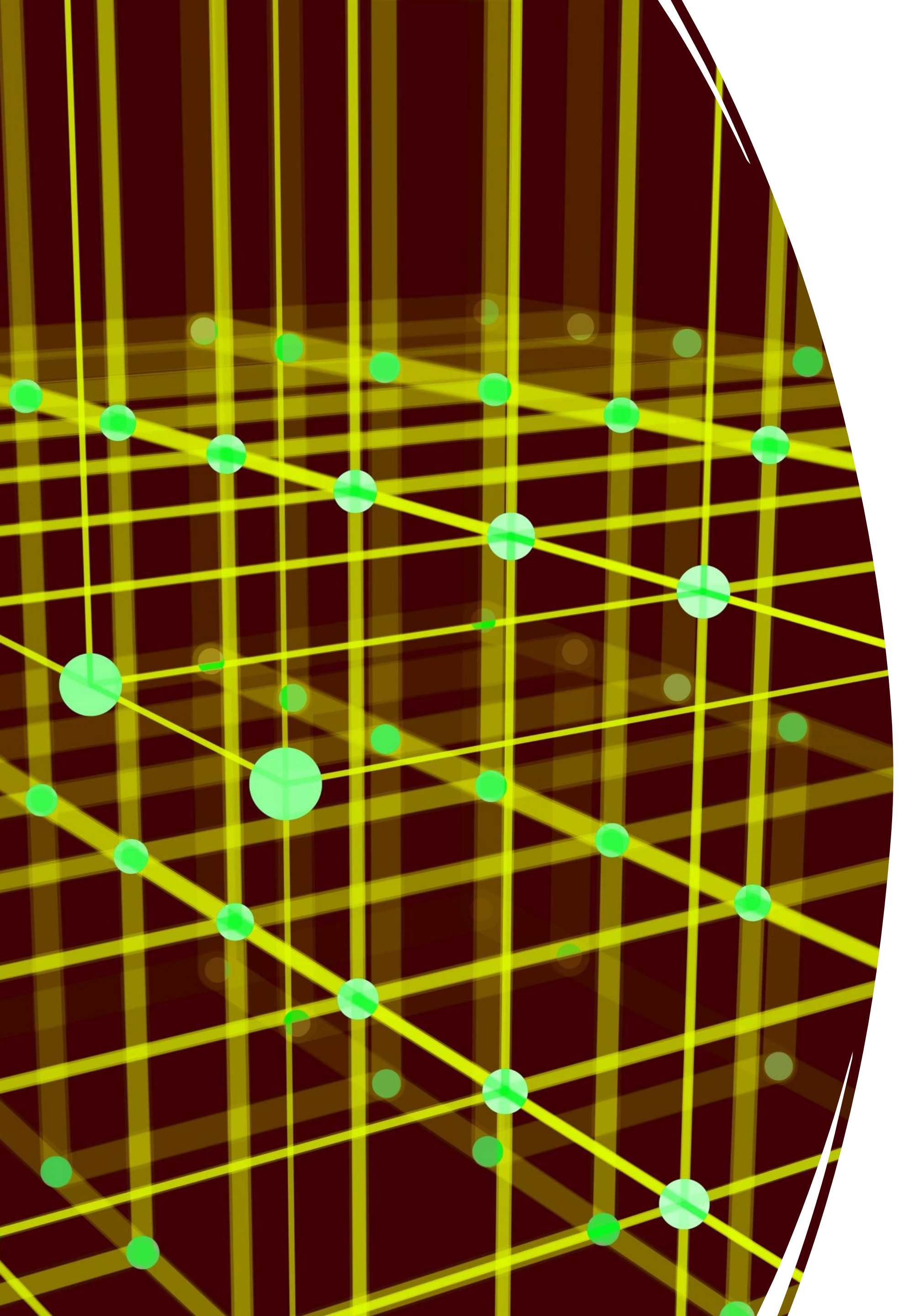
220 votes • Poll closed

Metrics

- High Level overview
- How many requests/sec there are happening?
- How many errors?
- Is the Host Down?
- 1000s of metrics may be collected every second
- Displayed on hundreds of graphs

Logs

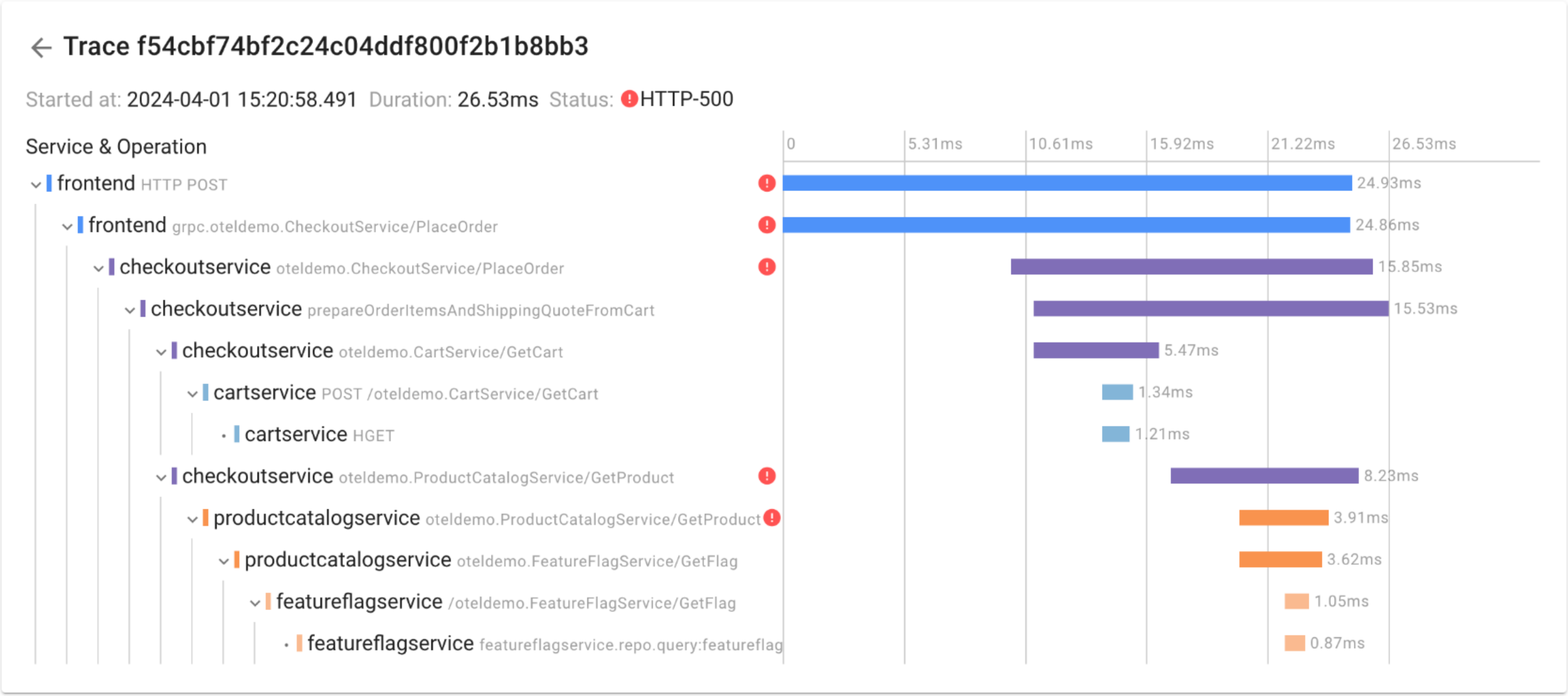
- Structured and Unstructured Format
- Have detailed information on what is happening
- Error logs – contain detailed information about cause of errors
- Expensive to Produce
- Expensive to Store and Analyze
- Sampling and Filtering is often used



Distributed Tracing

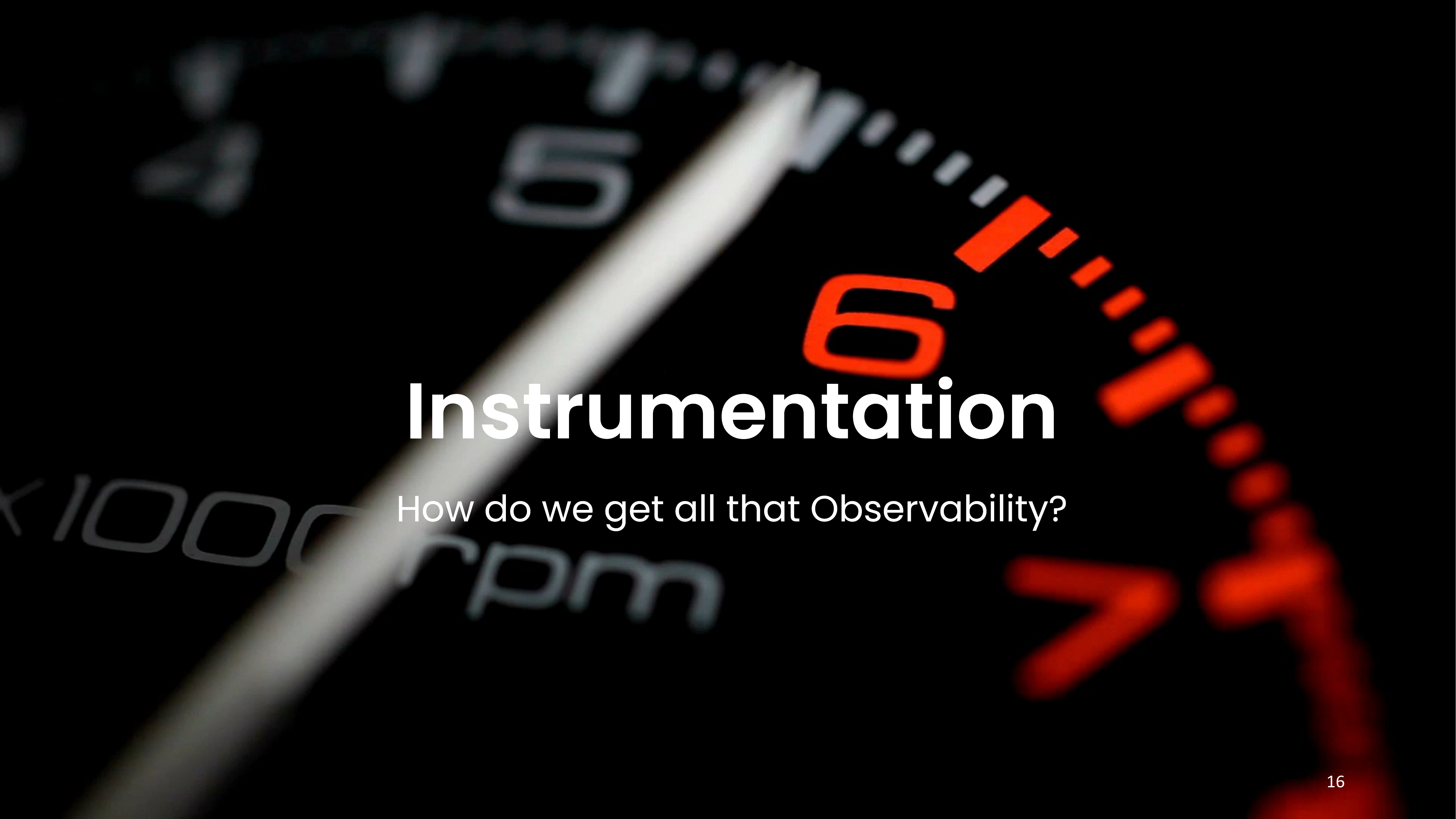
- Tracks Application Requests as they Pass through the system
- Tricky as we need to pass some Trace_ID between different services
- Span – Named, Timed Operation which represents part of Workflow
- Great for Root Cause Analyses
- Often Sampled
- Expensive to produce and store

Distributed Tracing Example



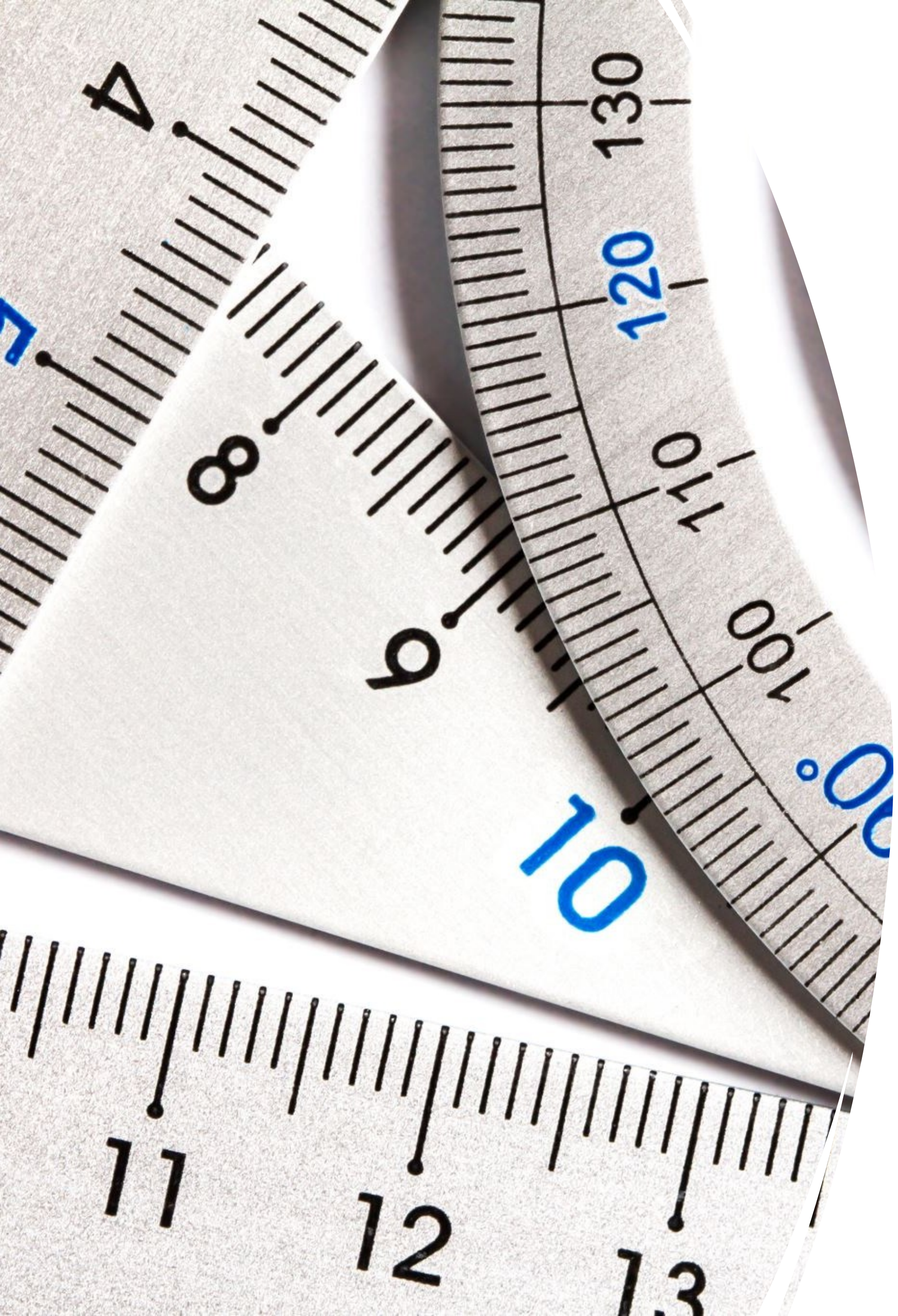
Profiling

- Where CPU Time or Wall Clock time is Spent
- Single Service or Distributed
- Language Developer can Understand
- Comparisons are very helpful
- Programming language specific support needed



Instrumentation

How do we get all that Observability?



Types of Instrumentation

- Static Instrumentation
  - Specific Places in the Code can Produce Metrics, Emit Logs, Traces
  - Linux ProcFS
- Dynamic Instrumentation
  - Allow Instrumentation “anything” dynamically
  - dTrace, eBPF

When and Where should we Instrument?

- Always-On Instrumentation
  - Data is always captured and retained
- Temporary Instrumentation
  - Instrumentation enabled when needed to diagnose the problems
- Canary System Instrumentation
  - Small Portion of Systems run additional (expensive) instrumentation

Instrumentation Challenge

Better observability comes from more Instrumentation

If Instrumentation is Hard it does not get Done

Swiss Cheese of Observability

Making a system observable

- Collecting telemetry data: metrics, logs, traced, profiles
  - Time- and resource-consuming process since it requires **adding instrumentation into every application**
  - Hard to achieve 100% coverage without blind spots (3rd party and legacy services)
- Storing telemetry data in some databases
- Learning how to troubleshoot your system using all that data
  - The most challenging part

Collecting telemetry data

Before answer HOW to gather, let's discuss WHAT to gather or what we want to know about our apps.

- SLI (Service Level Indicators): **requests, errors, latency**
- Communication with other services or databases: requests, errors, latency
- Resource-related metrics: CPU, Memory, Disk
- Network-related metrics: latency, connectivity, packet loss
- Node-level metrics and logs
- Runtime-related metrics: GC, Thread Pools, Connection pools, Locks
- Orchestrator-related metrics
- Logs to identify application-specific issues
- Profiles to explain spikes in CPU or Memory usage

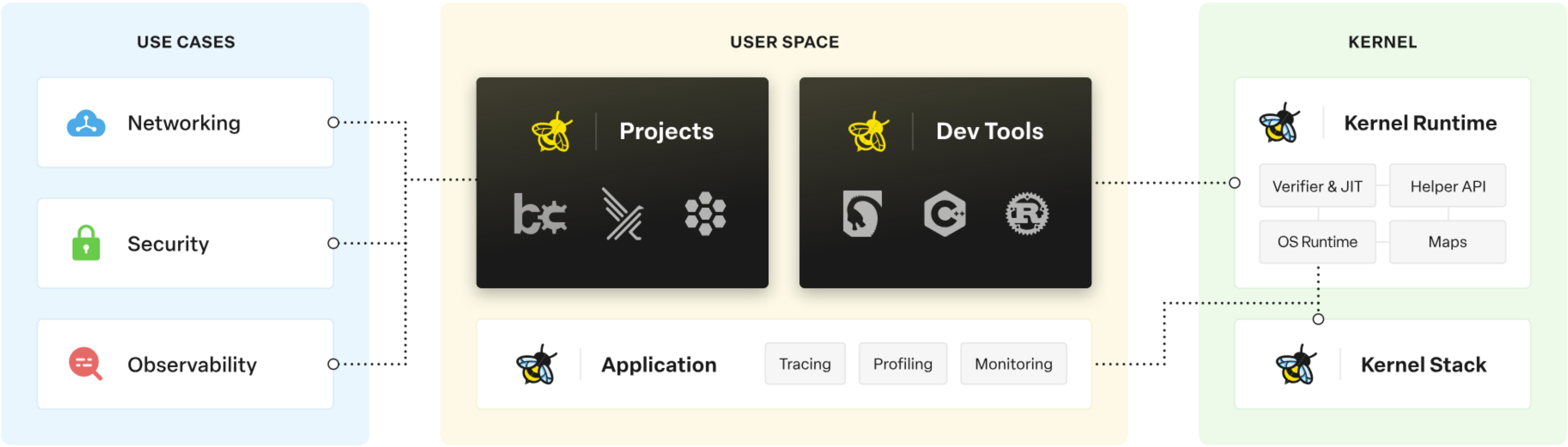
Collecting telemetry data

- It's possible to collect all these data without using eBPF, but eBPF allows to achieve that in **MINUTES**
- There are always legacy and 3rd party services that you can't instrument. eBPF doesn't require code changes and redeployments.
- Usually, developers instrument only most critical services, so you can't be sure that you have no blind spots.
- Instrumentation is a continuous process, so you need to ensure that every new service integrates OpenTelemetry SDKs.

A quick intro into eBPF

- A feature of the Linux kernel
- Allows to run small programs in the kernel-space and call them on any kernel or app function call
- Such programs have access to function arguments and returning values
- Then, they can send some data to a program in the user-space

eBPF is just a way how we can obtain data, we just need to implement kernel-space and user-space programs



eBPF Illustrated

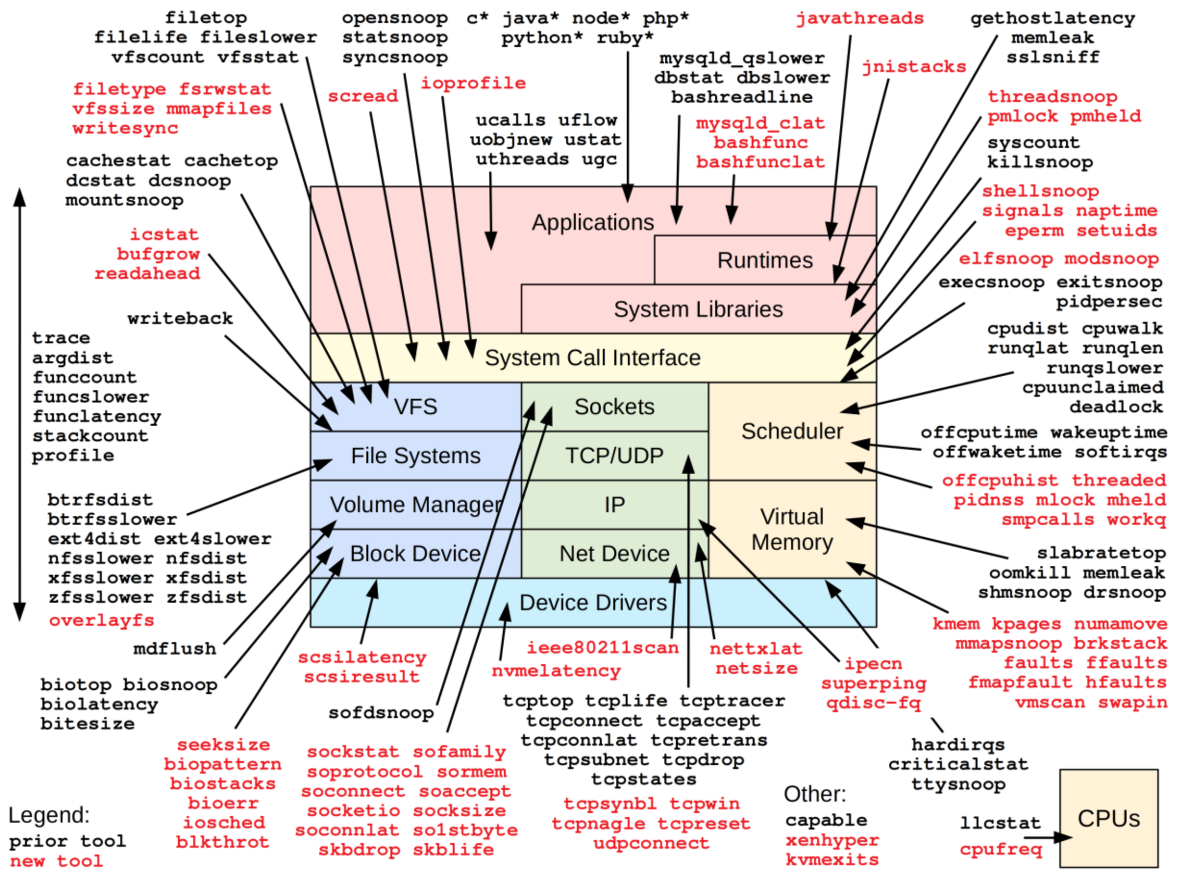
https://ebpf.io

How to use eBPF

- **Kprobe** allows to capture any kernel function call
  - Kernel functions can be renamed, deleted, their arguments can change
  - Some functions, in fact, almost never change
- **Tracepoints** – statically instrumented places in the kernel which are relatively stable comparing to **Kprobe**
- **Uprobe** allows to capture user-space programs calls
- **MAPS** allow to store some state in the kernel space
- **PERF_MAPS** allows to share data between the kernel-space and user-space

It’s good to know, but you don’t have to write your own eBPF programs. There are a lot of ready-made tools, such as **Coroot**

eBPF Awesome Command Line Tools



https://www.brendangregg.com/blog/2019-07-15/bpf-performance-tools-book.html

Example: CPU Runqueue Latency

```
root@localhost:/usr/share/bcc/tools# ./runqlat 10 1
Tracing run queue latency... Hit Ctrl-C to end.

     usecs               : count     distribution
         0 -> 1          : 13       |                                        |
         2 -> 3          : 285      |**                                      |
         4 -> 7          : 2564     |*********************                   |
         8 -> 15         : 4827     |****************************************|
        16 -> 31         : 4817     |*************************************** |
        32 -> 63         : 2141     |*****************                       |
        64 -> 127        : 1086     |********                                |
       128 -> 255        : 709      |*****                                   |
       256 -> 511        : 588      |****                                    |
       512 -> 1023       : 426      |***                                     |
      1024 -> 2047       : 192      |*                                       |
      2048 -> 4095       : 95       |                                        |
      4096 -> 8191       : 41       |                                        |
      8192 -> 16383      : 3        |                                        |
```

https://coroot.com/blog/engineering/runqlat-and-runqslower-ebpf-command-line-tools/

eBPF Applications

bcc

Toolkit and library for efficient BPF-based kernel tracing

BCC is a toolkit for creating efficient kernel tracing and manipulation programs built upon eBPF, and includes several useful command-line tools and examples. BCC eases writing of eBPF programs for kernel instrumentation in C, includes a wrapper around LLVM, and front-ends in Python and Lua. It also provides a high-level library for direct integration into applications.

GITHUB

Cilium

eBPF-based Networking, Security, and Observability

Cilium is an open source project that provides eBPF-powered networking, security and observability. It has been specifically designed from the ground up to bring the advantages of eBPF to the world of Kubernetes and to address the new scalability, security and visibility requirements of container workloads.

GITHUB · WEBSITE

---

bpftrace

High-level tracing language for Linux eBPF

bpftrace is a high-level tracing language for Linux eBPF. Its language is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap. bpftrace uses LLVM as a backend to compile scripts to eBPF bytecode and makes use of BCC as a library for

https://ebpf.io/applications/

Coroot-node-agent (Apache 2.0 license)

- An open-source Prometheus/OpenTelemetry compatible agent that gathers metrics, logs, traces and profiles
- Discovers containers/processes running on the node
- Discovers their logs (k8s, docker, containerd, journald) and sends them over OTLP
- Extracts repeated patterns from logs and generates log-based metrics
- Monitors TCP connections of every container
- Measures network latency between each container and its peers
- Tracks communications between services (requests, errors, latency), supports HTTP, GRPC, Postgres, MySQL, MongoDB, Redis, Memcached, Cassandra, Kafka, Rabbitmq, NATS

How the agent leverages eBPF

tracepoint/task/task_newtask: tracking new process creation. It reports only a PID, then the agent discovers container metadata using /proc

tracepoint/oom/mark_victim: marking a process as a victim of the OOM killer

tracepoint/sched/sched_process_exit: tracking process terminatio. If a process was terminated by the OOM killer, the event is enriched with the reason of the termination

tracepoint/syscalls/sys_enter_open(at): tracking file openings to identify the logs and partitions used by a specific container

How the agent leverages eBPF

tracepoint/syscalls/sys_enter_connect: tracking FD of a TCP connection

tracepoint/sock/inet_sock_set_state: tracking peers and states of TCP connections

tracepoint/tcp/tcp_retransmit_skb: tracking TCP retransmissions

coroot :~#

How the agent leverages eBPF

tracepoint/syscalls/sys_enter_write/writev/sendmsg/sendto: track writes to an FD (socket)

tracepoint/syscalls/sys_enter_read/readv/recvmsg/recvfrom: track reads from an FD (socket)

2-phase L7-protocol parsing:

- Kernel space: high-performance protocol detection
- User—space: protocol parsing for generating metrics and traces

SSL

Capturing the data before encryption and after decryption.

- For apps using OpenSSL:
  - uprobe/SSL_read
  - uprobe/SSL_write

- For GO apps:
  - uprobe/go_crypto_tls_write
  - uprobe/go_crypto_tls_read

eBPF: performance impact

The Linux kernel ensures minimal interruption to kernel code execution by validating each eBPF program before execution:

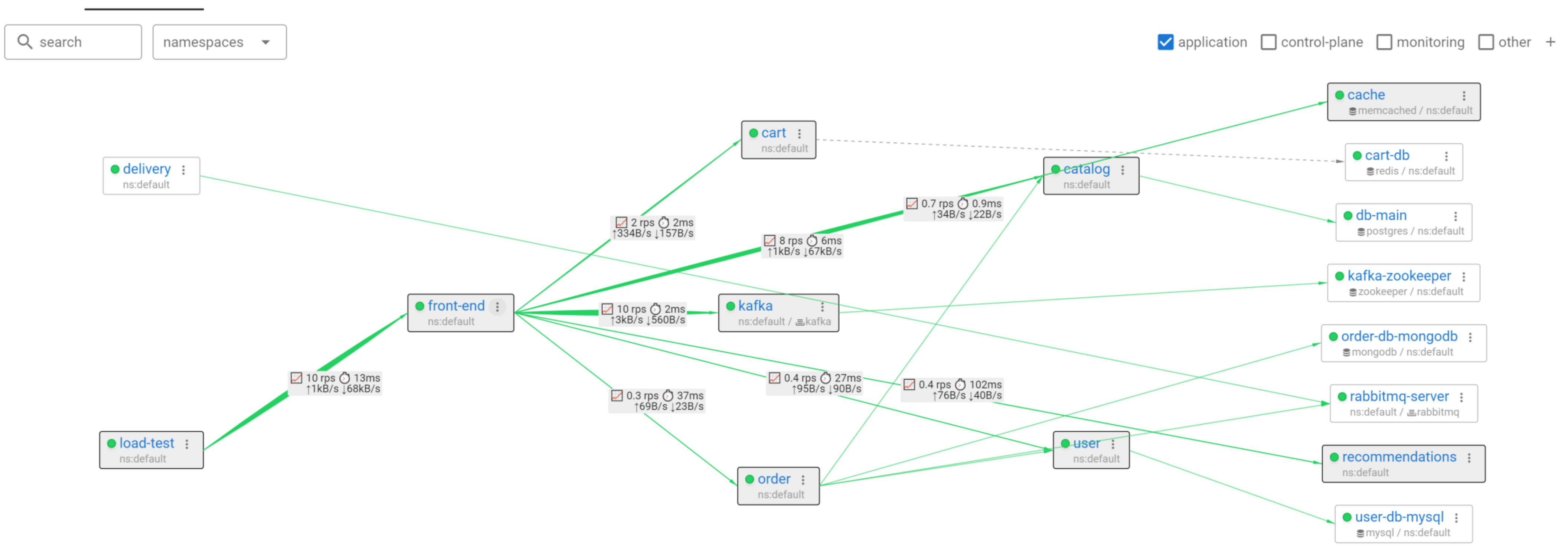
- Program **must** have a finite complexity.
- The verifier evaluates all possible execution paths within configured upper complexity limits

Communication between kernel-space and user-space programs occurs through a ring buffer:

- If the user-space program delays data reading, it may miss data due to overwriting

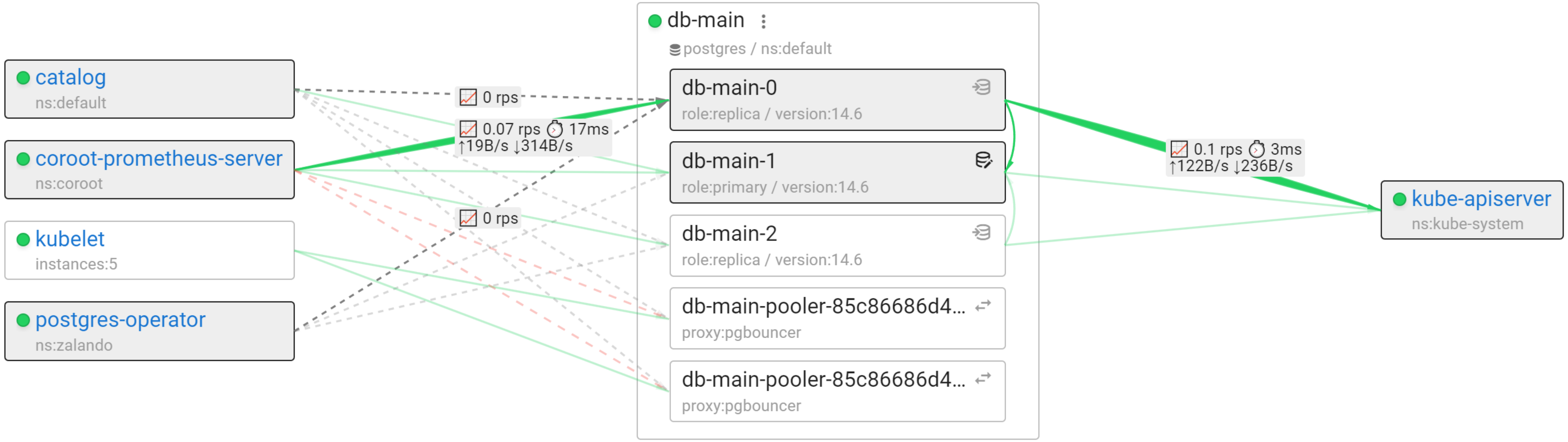
For observability, it's a great deal: although we might lose some telemetry data, we can be sure that there is no impact on performance

eBPF-based metrics



coroot :~#

eBPF-based metrics



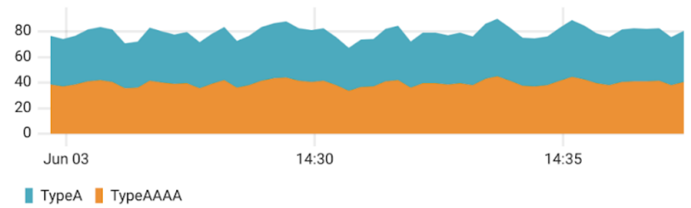
- L7: requests per second, Errors, Latency
- Network Round-trip-time (RTT)
- TCP: connections, failed connection attempts, retransmissions (can signify packet loss)

eBPF Based DNS Profiling

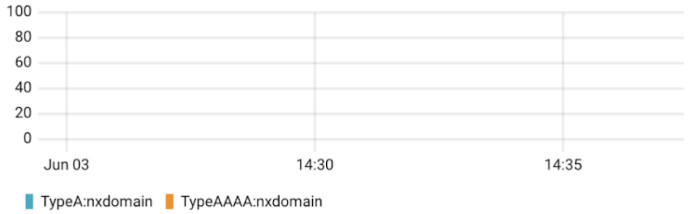
- DNS latency: ok
  Condition: the 95th percentile of DNS response times > 100ms
- DNS server errors: ok
  Condition: the number of server DNS errors (excluding NXDOMAIN) > 0
- DNS NXDOMAIN errors: ok
  Condition: the number of the NXDOMAIN DNS errors (for previously valid requests) > 0

Domain	Requests	No results (IPv4: A)	No results (IPv6: AAAA)	No results (other)	ServFail
demo-currencyservice.default.svc.cluster.local	20.2 k	—	—	—	—
demo-shippingservice.default.svc.cluster.local	14.5 k	—	—	—	—
demo-cartservice.default.svc.cluster.local	14.2 k	—	—	—	—
demo-paymentservice.default.svc.cluster.local	7.3 k	—	—	—	—
demo-productcatalogservice.default.svc.cluster.local	7.1 k	—	—	—	—
demo-emailservice.default.svc.cluster.local	78	—	—	—	—

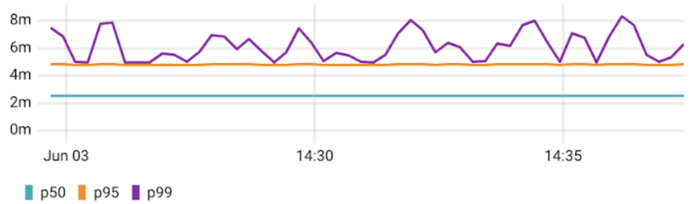
DNS requests by type, per second



DNS errors, per second



DNS latency, seconds



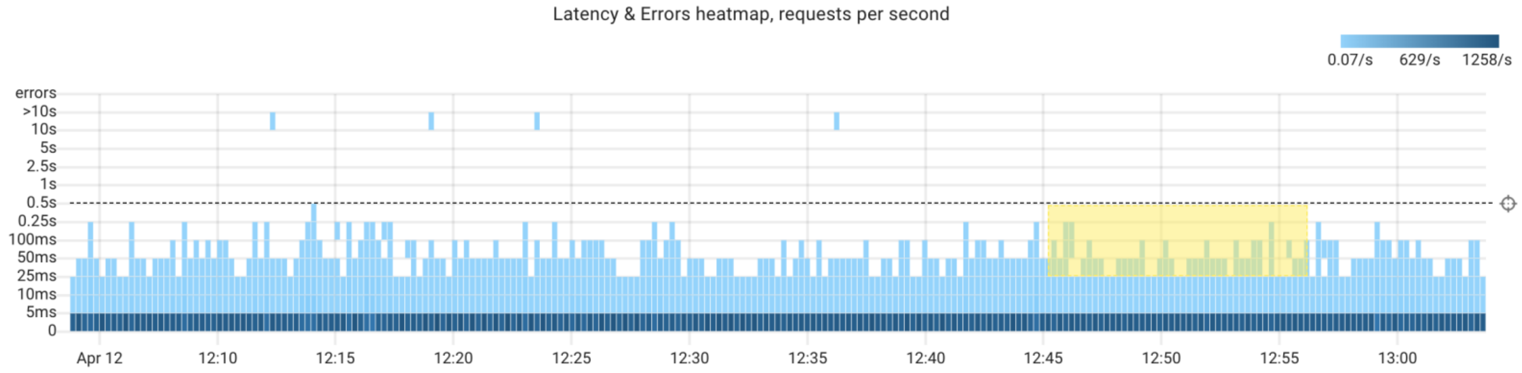
eBPF Based Network Cost Monitoring

← Applications ⓘ category: monitoring ⊗ Search

Application	↓ Usage costs	Allocation costs	Overprovisioning costs	Cross-AZ traffic	Internet egress traffic
coroot-prometheus-server	$11.84/mo	$29.57/mo	$15.52/mo	$2.87/mo	–
coroot-node-agent	$9.36/mo	$5.45/mo	–	$0.38/mo	–
coroot	$4.78/mo	$8.68/mo	$2.76/mo	$1.30/mo	–
coroot-clickhouse-shard0	$2.73/mo	–	–	–	–
coroot-cluster-agent	$1.52/mo	$8.68/mo	$6.76/mo	$0.08/mo	–
coroot-connect	$0.49/mo	$8.68/mo	$8.01/mo	$8.94/mo	$93.52/mo
coroot-kube-state-metrics	$0.22/mo	–	–	$0.02/mo	–
coroot-connect-dev2	$0.05/mo	$8.68/mo	$8.53/mo	–	–
coroot-connect-dev1	$0.03/mo	$8.68/mo	$8.53/mo	–	$0.00/mo

Rows per page: 20 ▾ 1-9 of 9 ‹ ›

eBPF-based traces (spans)



	Client	Status	Duration	Name	Details
	app	OK	27.04ms	query	`PREPARE AS select title, body from articles where id = any ($1)`
	app	OK	53.30ms	query	`insert into articles (created, title, body) values ($1, $2, $3)`
	app	OK	38.06ms	query	`select id from articles where created < $1 order by created desc limit $2`
	app	OK	26.22ms	query	`PREPARE AS select id from articles where created < $1 order by created desc limit $2`

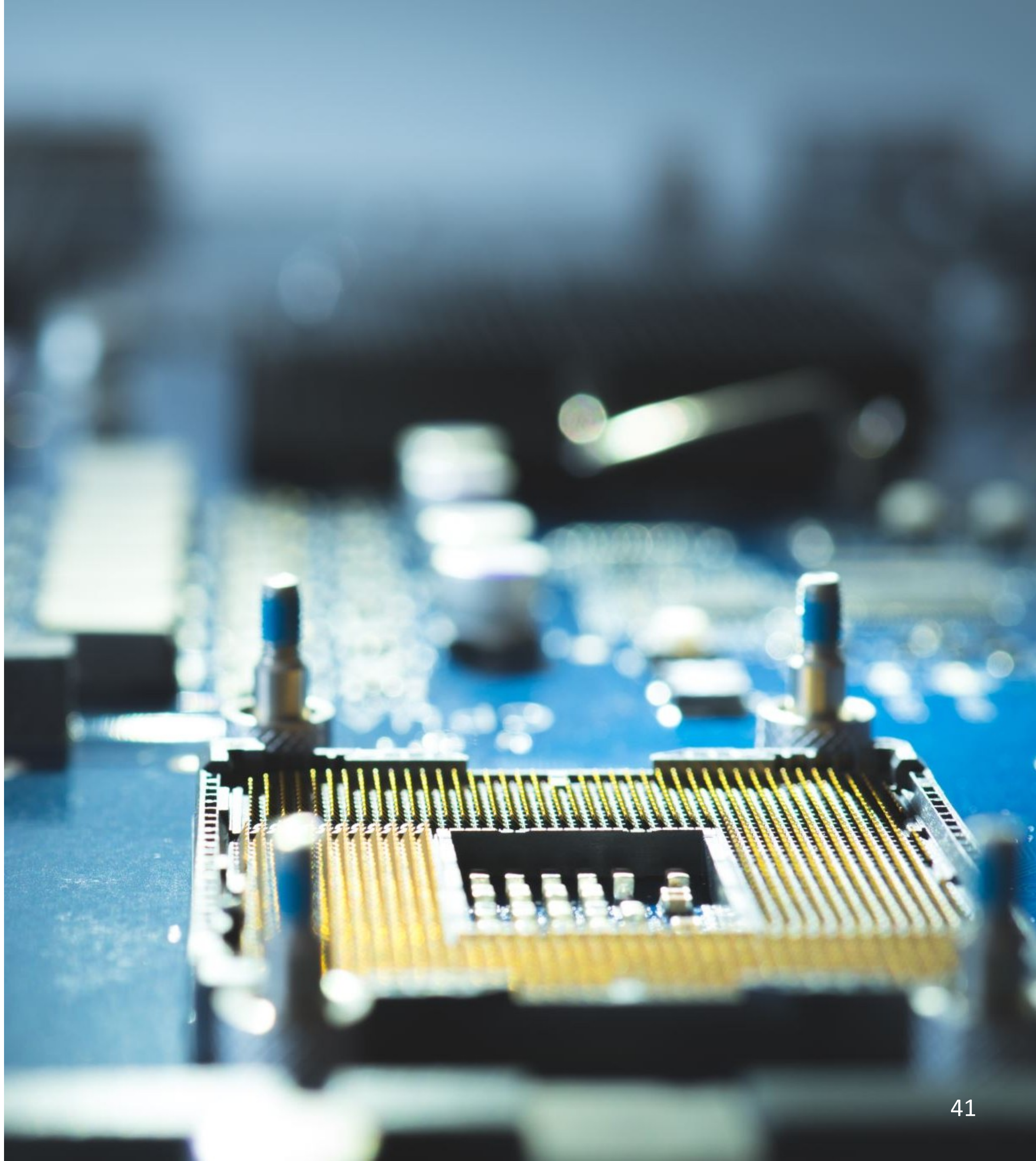
- Traces are extremely useful for identifying the particular requests within an anomaly
- They also provide a more granular distribution of requests by latency and status

eBPF-based tracing limitations

- The OpenTelemetry SDKs generate a TRACE_ID for each request and propagate it to other services
- When using eBPF, TRACE_IDs are not available, limiting us to capturing individual spans (requests) rather than complete traces
- There's a tool that claims to generate TRACE_IDs by intercepting and **modifying** requests, but I think it's not a good idea
- Coroot supports both traditional OpenTelemetry integration and eBPF-based tracing methods

eBPF-based continuous CPU profiling

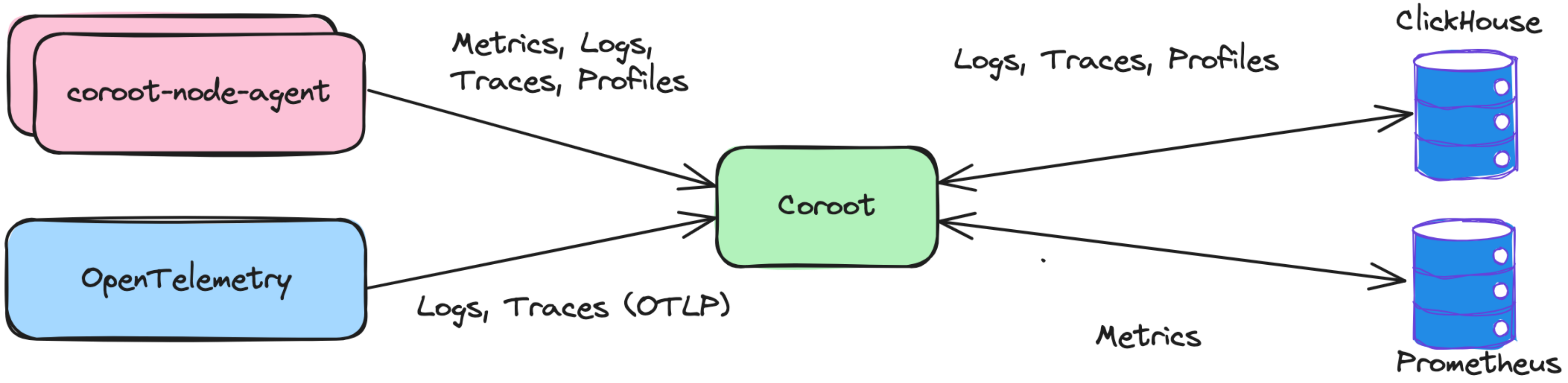
- Allows to explain any anomaly in CPU usage precise to the particular line of code
- Doesn’t require any code changes
- Gathers per-process call stacks and aggregates them by containers
- Resolution by default is 60 seconds, so you can compare profiles within and anomaly with previous periods



eBPF-based CPU profiling

eBPF-based CPU profiling (comparison mode)

How Coroot works



- coroot-node-agent: gathers metrics, logs, traces, and profiles. Installed on every node in the cluster (k8s, docker, VM, bare-metal)
- Prometheus for storing metrics
- ClickHouse for storing logs, traces, and profiles
- Coroot: UI, alerts, predefined inspections
- You can use Coroot as an OpenTelemetry backend for logs and traces

Conclusion

- eBPF is awesome!
- It enables gathering almost any telemetry data you need without requiring code changes.
- The performance impact on your apps is negligible.
- Want to gain system visibility in minutes? Install Coroot (Open Source, Apache 2.0).
- https://github.com/coroot/coroot

Thank you, Let's connect!

DevOpsDays Chicago

https://devopsdays.org/

https://www.linkedin.com/in/peterzaitsev

https://twitter.com/PeterZaitsev

http://www.peterzaitsev.com

coroot :~#