ebay

Continuous Profiling Your Production Kubernetes Deployments At Scale

Nick Pordash - Lead Engineer, Observability
Vijay Samuel - Architect, Observability

Agenda

Pillars of Observability

What is Continuous Profiling

Architecture

Examples

Potential

Questions

Pillars of Observability

Logs

Unstructured/ structured strings (logs). Ex: sl4j, zap logger generated logs

Metrics

Continuous time series data. Ex: cpu usage, request count, latency etc.

Traces

Record of request execution through highly distributed microservices

Profiles

Code-level consumption numbers (e.g., memory used, CPU time spent) for various resources across different program functions over time.

Profiles

Performance metrics at the most granular level possible.
Historically viewed as non-production friendly (expensive).

Types

- CPU
- Heap
- Mutex
- IO
- GPU
- Language specific (Goroutines/JVM)

Information Available

- Why part of server resources are being consumed
- Statistics about execution down to the line number

Continuous Profiling

Advent of sampling profilers made profiling a lot cheaper. A time axis to information collected as part of profiling can be viewed on a chart.

Image Credit

Available Open Source Products

- Pyroscope
- CNCF Pixie
- Parca
- OTEL Profiles - In progress

Methodology

- Scrape - Collect profiles at a periodic interval by scraping something like a PPROF endpoint
- Push - Instrument with a client that can collect profiles and periodically push them
- eBPF - Use features on the Linux Kernel to extract profiling information.

Our Setup

Pyroscope as a candidate
Of all the choices, Pyroscope had a nice UX for us to be able to visualize profiling data.

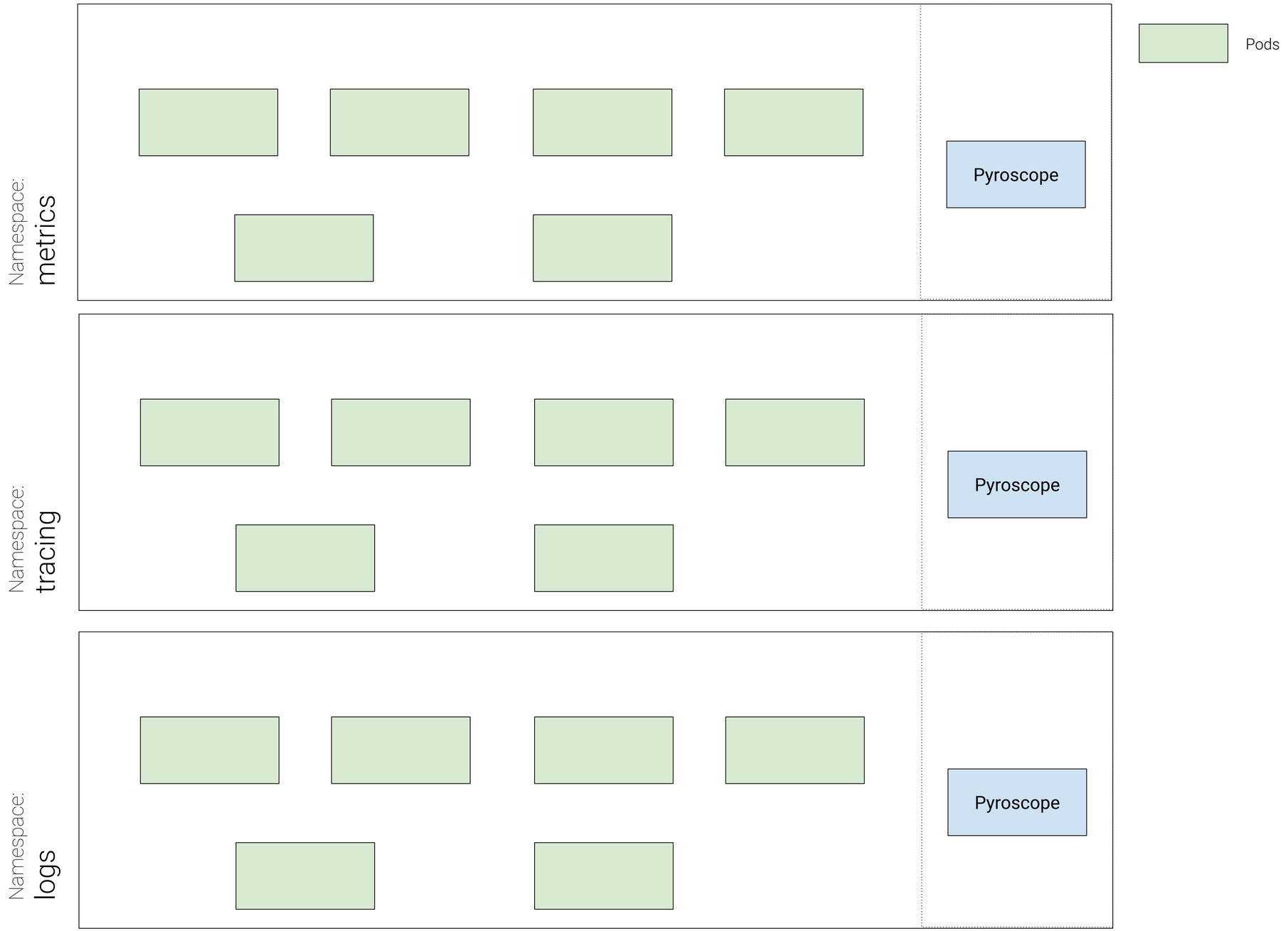
Easy to set up

Deployed against most of our platform
Environments that we actively use for L&P analysis were configured with Pyroscope installations.

Subsequently production namespaces were configured too.

Pyroscope kubernetes discovery uses Pod annotations to scrape profiles every 10 seconds.

Deployment Architecture



Obvious Questions

Are there performance degradations?
No! Golang pprof is very cheap. 10s polling cause less than 2% increase in CPU usage.

Are Profiles expensive to store?
Sort of. Raw samples do indeed take up a lot of space.

Pyroscope recommends storing few hours of raw samples. Aggregated data retained longer.

Did we find good use?
YES! Few examples next.

Example - Missed Compiler Optimization

Context

Golang compiler optimizes the following idiom to perform a map clear instead of iteratively deleting each key:

```
for k := range m {
    delete(m, k)
}
```

Problem

Highest cpu consumer is caused by an unoptimized map clear. Expected to see this code path call `runtime.mapclear` instead of `runtime.mapiternext`.

Example - Missed Compiler Optimization

Let's check the assembly instructions

```
$ go build -gcflags='-S' tsdb/tsdbpb/symbols.go 2>&1 | grep runtime.mapclear
    0x0023 00035 (/tsdbpb/symbols.go:38)        CALL    runtime.mapclear(SB)
```

Huh? It says that `runtime.mapclear` is used!

Let's check the assembly instructions from a caller

```
$ go build -gcflags='-S' ingest/*.go 2>&1 | grep symbols.go | grep runtime.mapiternext
    0x00cb 00203 (/tsdb/tsdbpb/symbols.go:38)        CALL    runtime.mapiternext(SB)
```

Theory: *is function inlining preventing the optimization?*

Example - Missed Compiler Optimization

Verify `runtime.mapclear` is still called for the updated method

```
$ go build -gcflags='-S' tsdb/tsdbpb/symbols.go 2>&1 | grep runtime.mapclear
    0x0023 00035 (/tsdb/tsdbpb/symbols.go:43)        CALL    runtime.mapclear(SB)
```

Verify callers do not inline the `reset` method

```
$ go build -gcflags='-S' ingest/*.go 2>&1 | grep symbols.go | grep reset
    0x0056 00086 (/tsdbpb/symbols.go:38)        CALL    /tsdb/tsdbpb.(*SymbolIndexer).reset(SB)
```

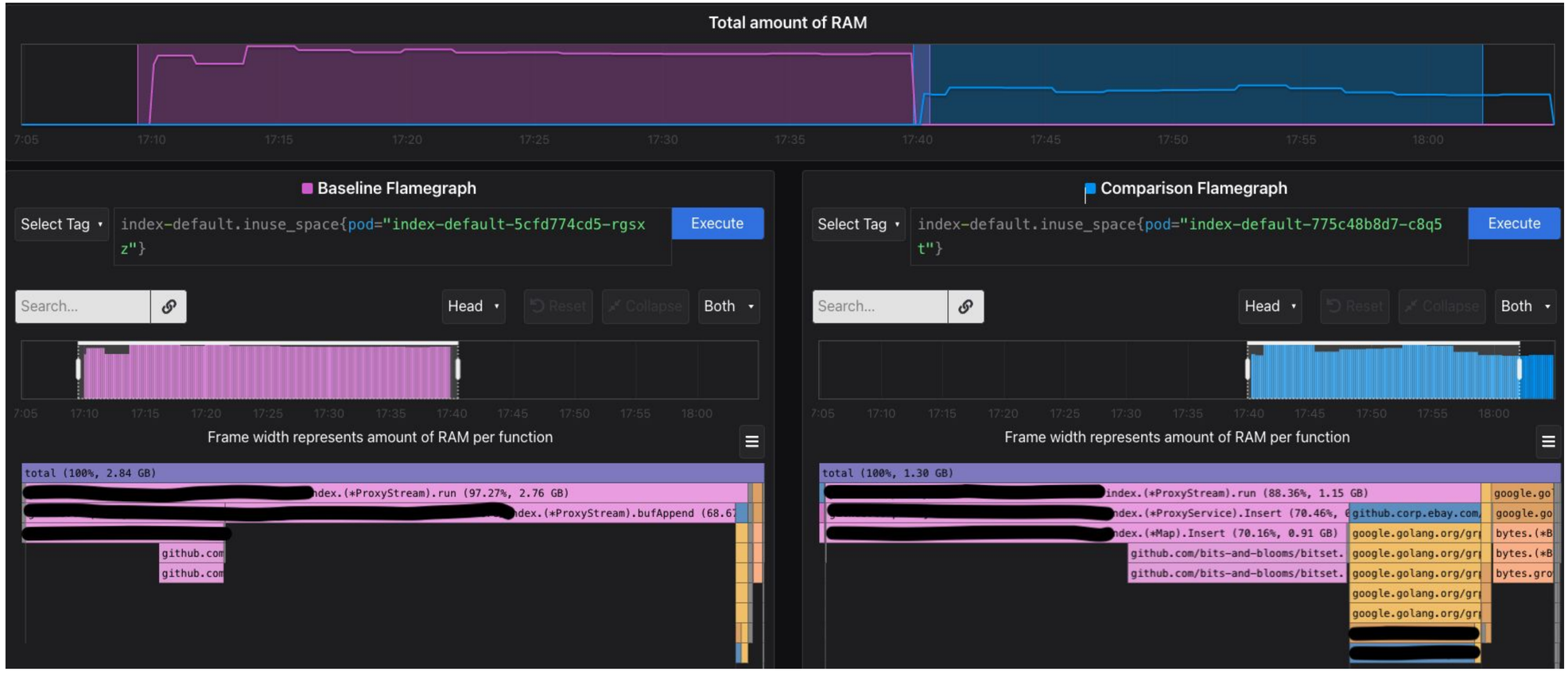
Success!

Example - Removing a Harmful Buffer

Problem

Memory buffer intended to ease memory-pressure is consuming an absurd amount of overall memory. Classic case of a premature optimization backfiring in a significant way.

Let’s compare profiles with and without the buffer



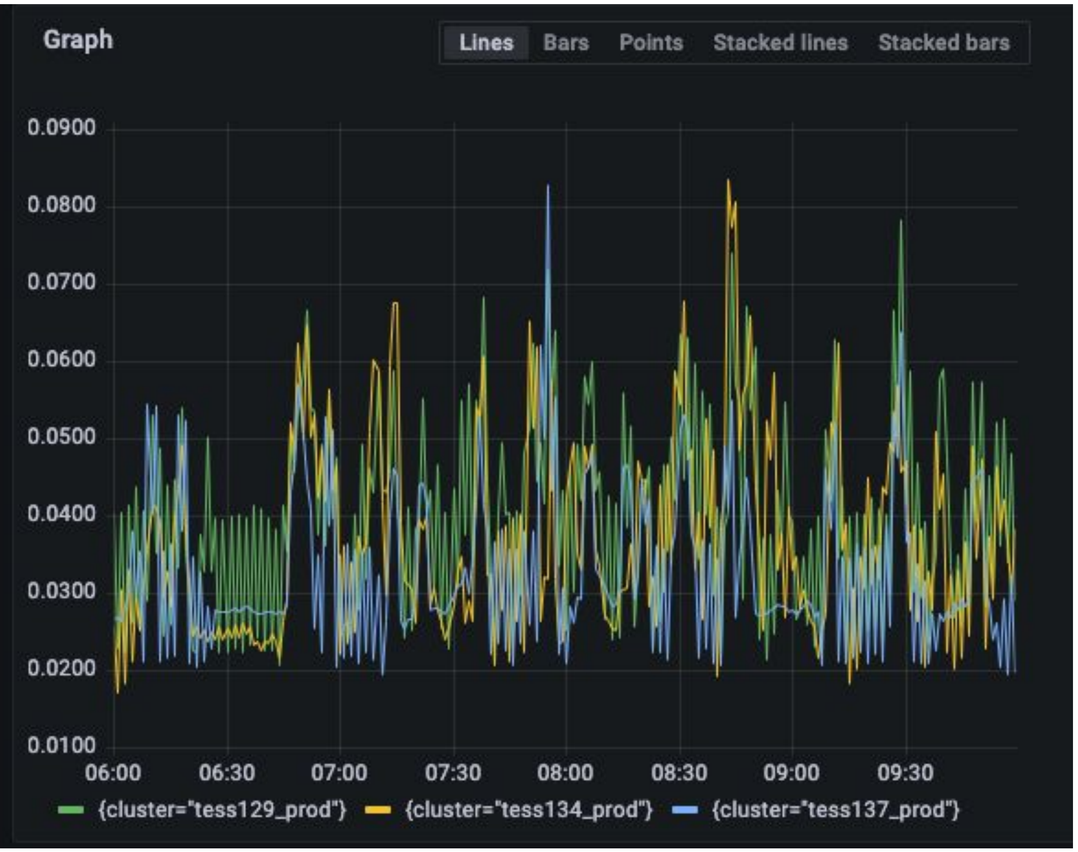
Example - Removing a Harmful Buffer

Let’s compare cpu and heap usage metrics for additional validation

Cluster 129 is running without the buffer

Memory

CPU



Demo

Potential

Memory Leak/CPU Spike Detection

Allow automation to look at profiles and detect leaks/spikes.

Compare previous and new build to spot anomalous resource usage.

Never have to run a bad build to analyze leaks. Roll it back immediately!!

3-Click RCA

Starting from an alert, view metrics, traces and finally profiles to identify method/line level root cause analysis.

Helps drive reduction in time-to-triage.

Continuous Cost Savings

Analyse performance measures and proactively optimize CPU and memory utilization.

Never waste resources ever again!

Easy to have information readily available than manually profile!

Questions?

Thank you