Beyla performance calculation

This document is shared publicly by Grafana Labs

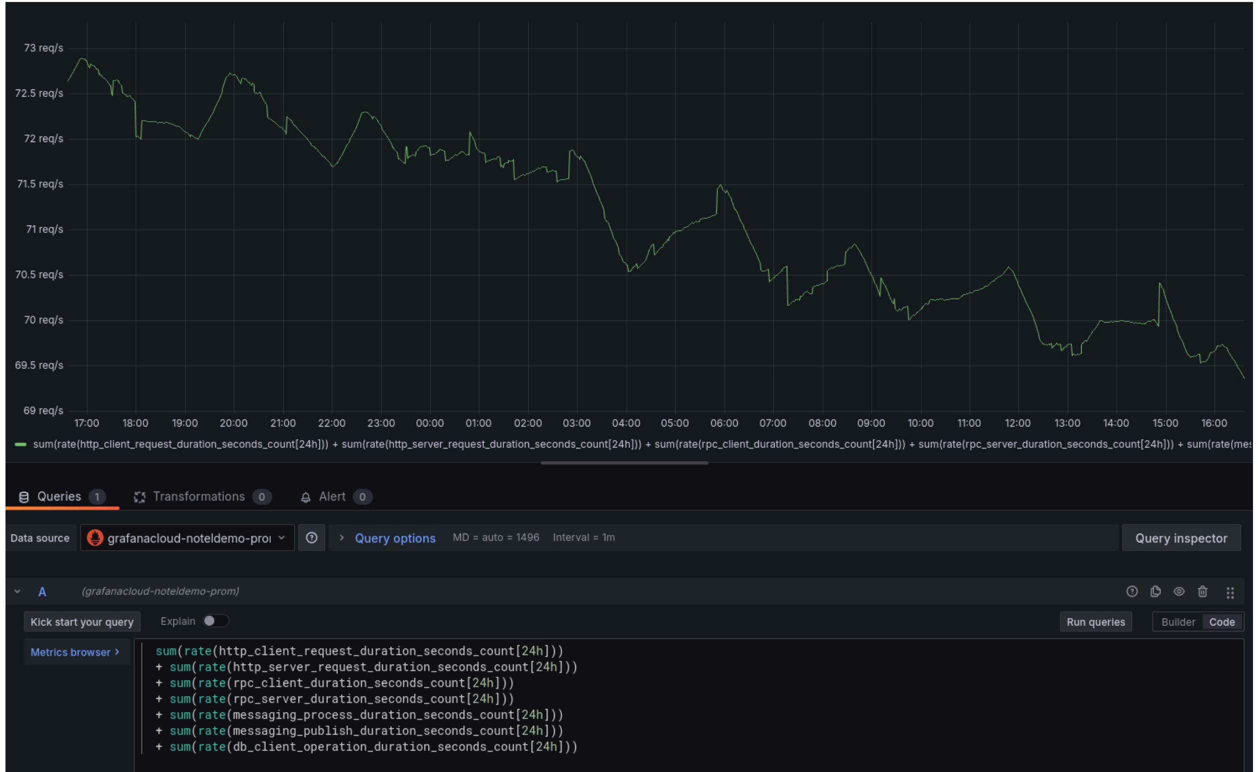
This document gathers all the information regarding performance implications of running Beyla in production where a diverse set of applications are instrumented.

In the first iteration of this we are going to focus on Beyla as single process (either in Kubernetes or not), therefore we are going to get out of the scope the following:

- Overhead caused by running Beyla in a Pod.
- Run Beyla as DaemonSet and have k8s cache functionality activated.
- Be exhaustive with all possible configurations of options and combinations of them.

In the present document we are running different experiments with different options activated. If one other option doesn't impact performance, it won't be documented publicly. We are using the following approach to measure the performance.

- Deploy Beyla in a local kind cluster using Helm (version 1.9)
- Deploy open telemetry demo with load generator activated
    - This script generates traffic, between 20 and 60 requests/s, being the /api/products the endpoint with more traffic. Since there are also requests to redis, kafka and internal RPC, the total amount of requests can be calculated like this.



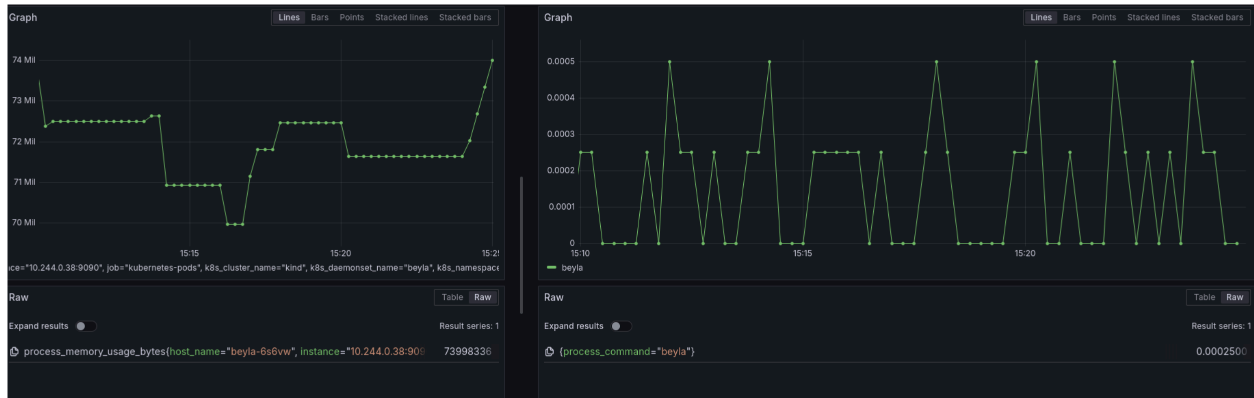
- Measure performance with *application_process* and visualize data in Grafana.
    - Potentially measure the performance in a different way to measure overhead of *application_process* option

These are the different scenarios considered

Beyla self instrumented

```yaml
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
     services:
       - k8s_namespace: default
         k8s_daemonset_name: beyla
   prometheus_export:
     port: 9090
     path: /metrics
```

```
    features:
      - application
      - application_process
```



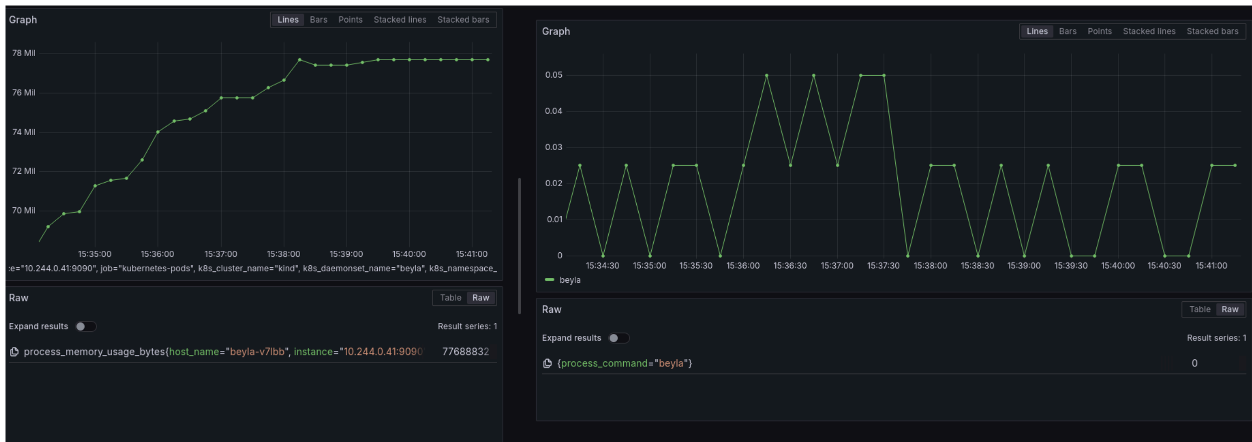
Summary:

Beyla uses around 75mb and 0.02% of CPU when only self-instrumented. Enabling application and application_process has a minimal overhead on the memory.

Instrument Go applications (1, many, with and without load)

1 process

```
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
     services:
       - k8s_namespace: default
         k8s_daemonset_name: beyla
       - k8s_namespace: default
         k8s_deployment_name: notel-demo-checkoutservice
   prometheus_export:
     port: 9090
     path: /metrics
     features:
       - application
       - application_process
```



Summary:

In this case the memory didn't change, so we have memory around 78MB and CPU around 0.05%.

with traffic

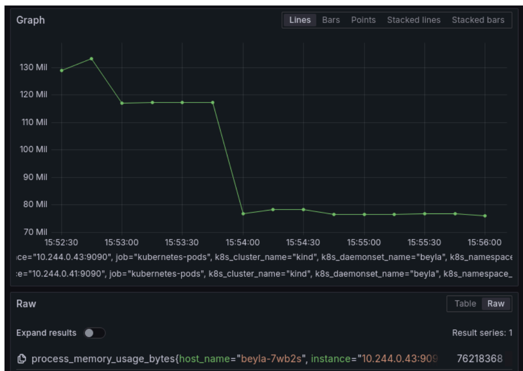
Summary:

The memory stays constant but now Beyla has to process the requests, so the CPU it's around 0.1 %, which is more or less 10x than before.

many processes

```yaml
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
```

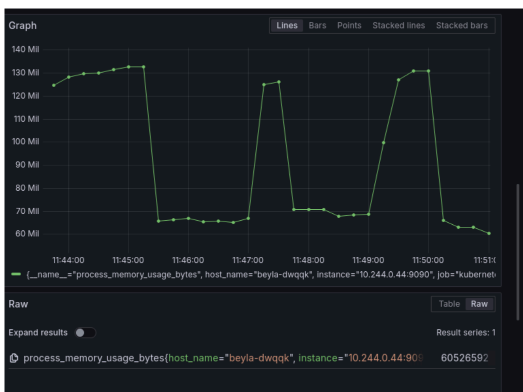
```yaml
      services:
        - k8s_namespace: default
          k8s_daemonset_name: beyla
        - k8s_namespace: default
          k8s_deployment_name:
notel-demo-checkoutservice|notel-demo-productcatalogservice|notel-demo-fla
gd
    prometheus_export:
      port: 9090
      path: /metrics
      features:
        - application
        - application_process
```

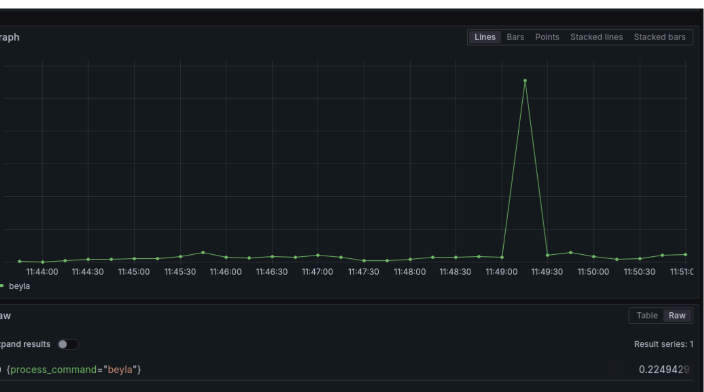


Summary:

When we instrument many processes at the same time, the memory increases dramatically (up to 130MB in this case), but then it returns to its normal state. The CPU also stays constant.

with traffic





Summary:

The memory fluctuates, probably due some caching or metrics expiration. In this case the Go applications are receiving traffic, therefore the CPU is more busy again, reaching levels around 0.2%

Instrument non-Go applications (1, many, with and without load)

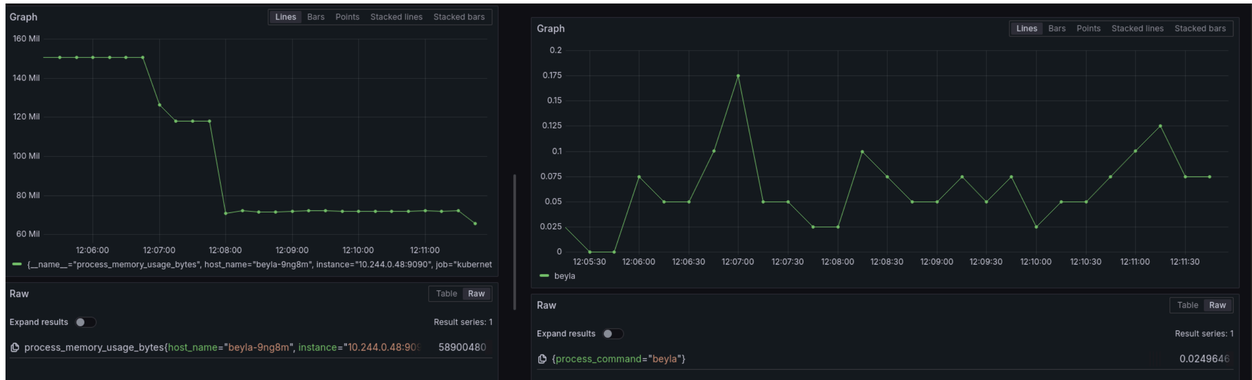
1 process

```yaml
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
     services:
       - k8s_namespace: default
         k8s_daemonset_name: beyla
       - k8s_namespace: default
         k8s_deployment_name: notel-demo-adservice
   prometheus_export:
     port: 9090
     path: /metrics
     features:
       - application
       - application_process
```

Summary:

In this case, the values are pretty much the same as the case of Go.

with traffic



Summary:

In this case, the values are pretty much the same as the case of Go.

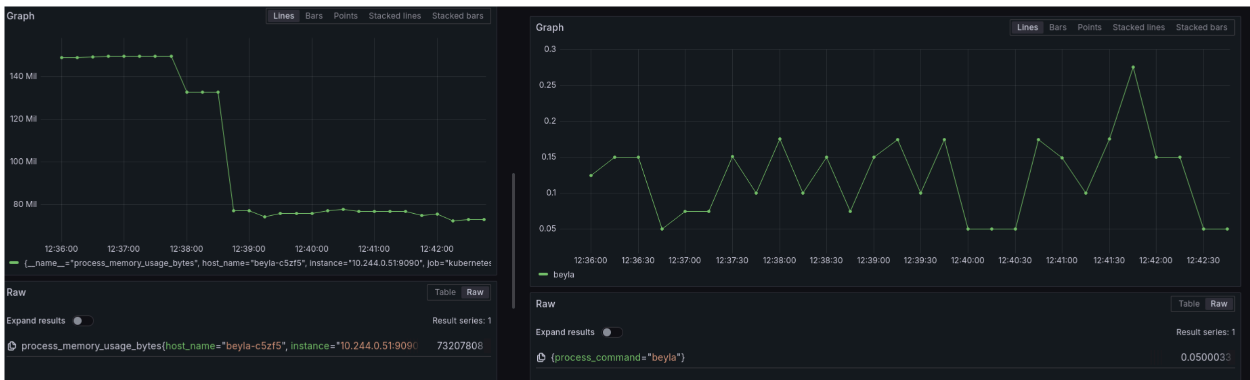
many process

```yaml
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
     services:
       - k8s_namespace: default
         k8s_daemonset_name: beyla
       - k8s_namespace: default
         k8s_deployment_name:
notel-demo-adservice|notel-demo-cartservice|notel-demo-recommendationservi
ce
   prometheus_export:
     port: 9090
     path: /metrics
     features:
       - application
       - application_process
```

Summary:

In this case, the values are pretty much the same as the case of Go.

with traffic



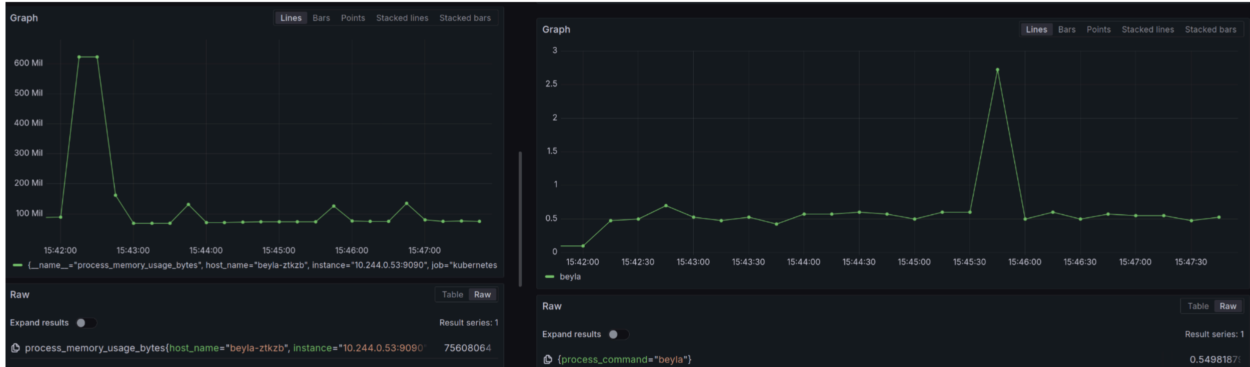
Summary:

In this case, the values are pretty much the same as the case of Go. The memory doesn't fluctuate like in the Go example.

Full OTEL demo with traffic

```
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
     services:
       - k8s_namespace: default
         k8s_daemonset_name: .
       - k8s_namespace: default
```

```yaml
        k8s_deployment_name: .
  prometheus_export:
    port: 9090
    path: /metrics
    features:
      - application
      - application_process
```



Summary:

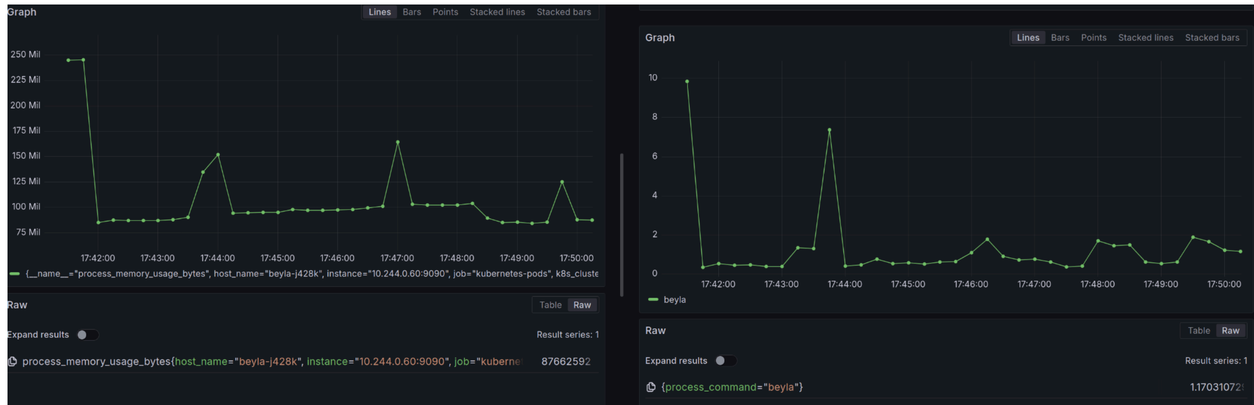
Instrumenting the whole OTEL demo causes a peak of 600mb initially, but then goes down to its normal levels of 75mb. The CPU increased to 0.5% utilization.

Different export modes (Prometheus, OTEL metrics, OTEL traces)

Prometheus

```yaml
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
     services:
       - k8s_namespace: default
         k8s_daemonset_name: .
       - k8s_namespace: default
         k8s_deployment_name: .
   prometheus_export:
     port: 9090
     path: /metrics
```

```
    features:
      - application
      - application_service_graph
      - application_span
      - application_process
```



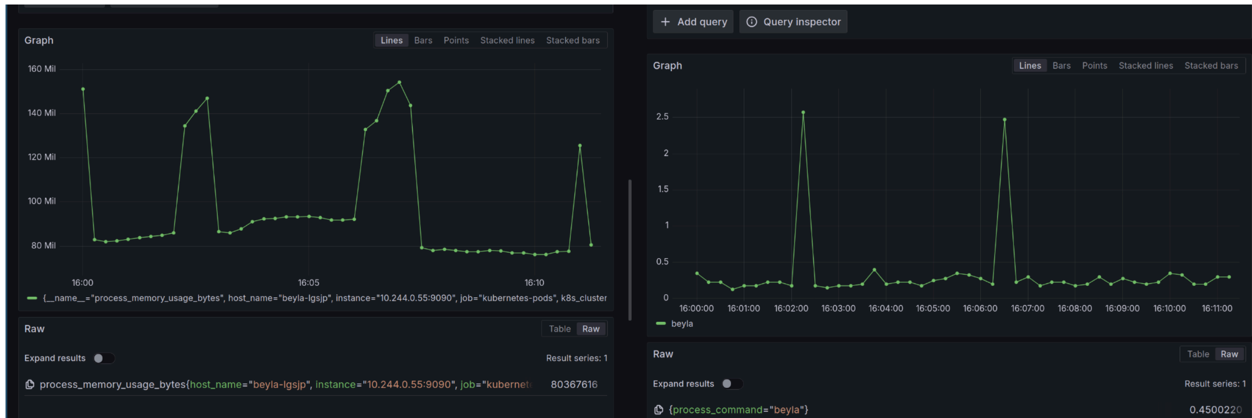
Summary:

Using application_span and application_service_graph causes an increase of memory of 10MB (87MB in total). The utilization increased from 0.5% to 1.2 %

OTEL traces

```
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
     services:
       - k8s_namespace: default
         k8s_daemonset_name: .
       - k8s_namespace: default
         k8s_deployment_name: .
   prometheus_export:
     port: 9090
     path: /metrics
     features:
       - application
       - application_process
```

```yaml
  grafana:
    otlp:
      cloud_zone: prod-us-central-0
      cloud_submit:
      - traces
      cloud_instance_id: user
      Cloud_api_key: key
```



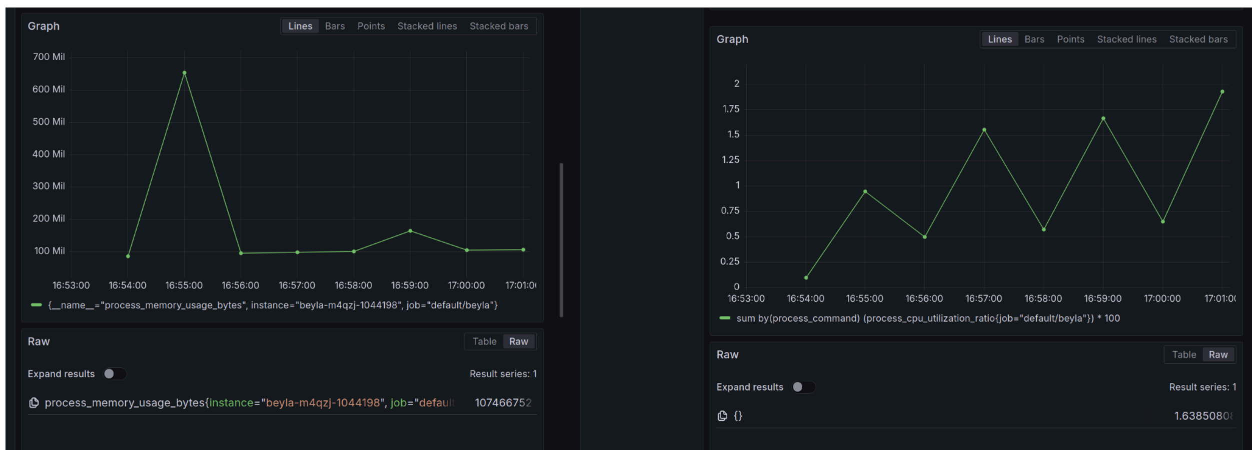
Summary:

Using traces is more costly for the memory, as it needs to create batches (that could explain the peaks of memory). However, it is more efficient in terms of memory as we don't need to create so many metrics.

OTEL metrics

```yaml
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
     services:
       - k8s_namespace: default
         k8s_daemonset_name: .
       - k8s_namespace: default
         k8s_deployment_name: .
   otel_metrics_export:
     features:
```

```
      - application
      - application_process
      - application_service_graph
      - application_span
  grafana:
    otlp:
      cloud_zone: prod-us-central-0
      cloud_submit:
      - metrics
      cloud_instance_id: user
      cloud_api_key: key
```



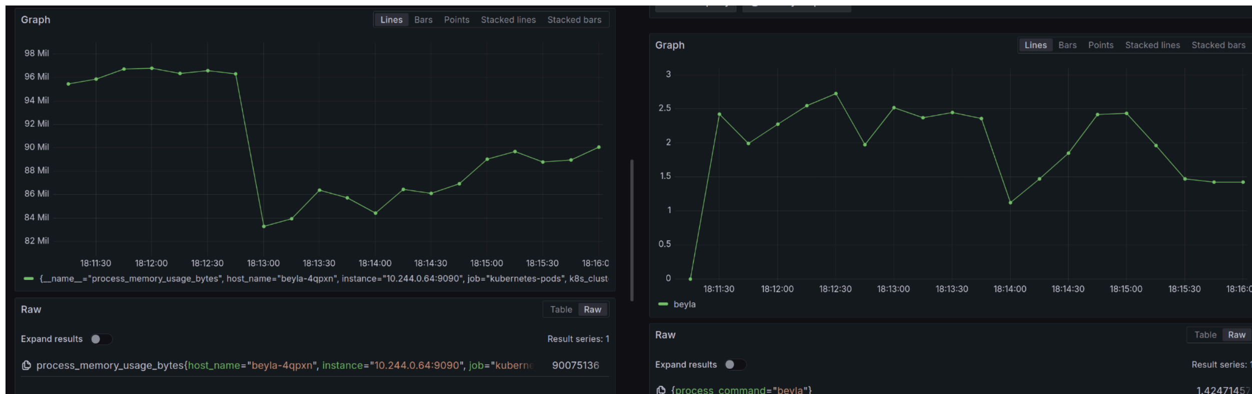
Summary:

Using application_span and application_service_graph, with a combination of OTEL metrics causes an increase of memory of 50MB (107MB in total). The utilization increased from 0.5% to 1.6 %

Debug mode

```
config:
 data:
   log_level: debug
   print_traces: true
   ebpf:
     bpf_debug: true
   attributes:
     kubernetes:
       cluster_name: kind
```

```
  discovery:
    allow_self_instrumentation: true
    services:
      - k8s_namespace: default
        k8s_daemonset_name: .
      - k8s_namespace: default
        k8s_deployment_name: .
  prometheus_export:
    port: 9090
    path: /metrics
    features:
      - application
      - application_process
```



Summary:

Enabling debug mode causes an increase of 20/30 MB and CPU utilization around 2%.

Network observability

```
config:
 data:
   attributes:
     kubernetes:
       cluster_name: kind
   discovery:
     allow_self_instrumentation: true
     services:
       - k8s_namespace: default
         k8s_daemonset_name: .
```

```
      - k8s_namespace: default
        k8s_deployment_name: .
  prometheus_export:
    port: 9090
    path: /metrics
    features:
      - application
      - application_process
      - network
```

Summary:

Instrumenting the whole OTEL demo causes a peak of 600mb initially, but then goes down to to 120mb, which is 70mb more than without the option enabled. The CPU increased to 1.2% (before was 0.5%).

BPF probes latency overhead

The observed latency of all combined probes of Beyla running for 24h in OpenTelemetry demo is around 40ms.

This represents 500ns of latency per request: