

PROMETHEUS

MEMORY OPTIMISATION

By Praneeth Pottimuthi, Shashank Sharma



Praneeth Pottimuthi

Engineer

Praneeth is working as a platform developer with Tata Communications, passionate in Kubernetes and advancing programming skills in Golang. With a good understanding of monitoring tools like Prometheus and Grafana, he enjoys learning new technologies and exploring open-source projects like Cilium and K8sGPT.

<https://www.linkedin.com/in/praneeth-pottimuthi-408080236/>



Shashank Sharma

Principal

Shashank Sharma has been following cloud native OSS for more than five years, with a particular focus on security, observability and serverless, and other platform engineering trends. At Tata Communications, he is working as a lead for Kubernetes and PaaS services.

<https://www.linkedin.com/in/shashank-sharma-ft9/>

TABLE OF CONTENTS

1. INTRODUCTION..... 1

2. WHY SHOULD WE OPTIMISE PROMETHEUS MEMORY USAGE?..... 1

3. OUR APPROACH TO PROMETHEUS MEMORY OPTIMISATION..... 2

 3.1 Shortlisting metrics with most timeseries per component..... 2

 3.2 Finding unused metrics..... 5

4. HOW TO DROP METRICS AND LABELS FROM PROMETHEUS METRICS..... 7

5. RESULTS..... 7

 5.1 Cluster 1..... 8

 5.2 Cluster 2..... 9

 5.3 Cluster 3..... 10

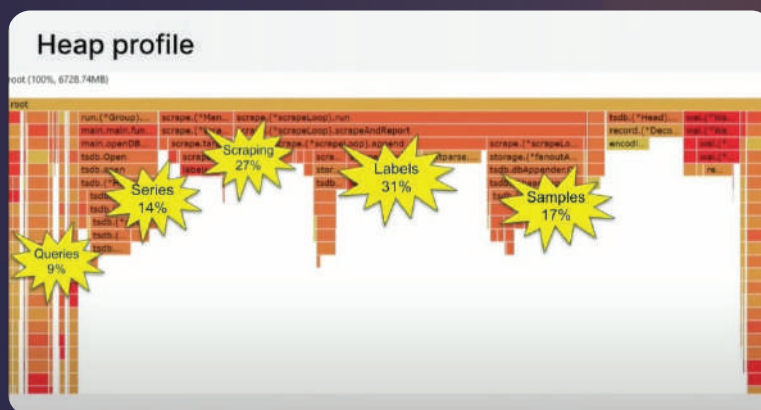
1. INTRODUCTION

Managing memory in Prometheus can be tricky, especially in Kubernetes environments where a vast number of metrics are constantly being generated. Whether you're just getting started with optimising Prometheus memory or have tried it before without success, this blog is here to guide you.

In this blog, we'll cover key strategies for effectively reducing memory usage in Prometheus, troubleshoot common issues like unexpected memory increases.

Before we dive in, here are some general strategies for reducing Prometheus memory usage:

- **Label management:** In a typical Go heap profile, around 31% of memory is occupied by labels. Reducing the number of labels can have a significant impact on memory usage.
- **Scraping optimisation:** Scraping processes take up about 27% of the memory. By optimising scraping intervals and targets, you can further reduce memory consumption.
- **Rule optimisation:** Prometheus rules load all the historical data within the specified time range. If a rule references a large time window, Prometheus must load extensive historical data to evaluate the conditions. Using shorter time ranges in your rules can significantly cut down on memory usage.



For those interested in diving deeper, we were inspired by several YouTube videos on this topic. If you want to explore these resources, feel free to check them out. But if you're eager to jump straight into optimising your setup, skip to the next section and get your hands dirty!

Prometheus memory cut in half: <https://www.youtube.com/watch?v=29yKJ1312AM>

Make Prometheus use less memory: <https://www.youtube.com/watch?v=suMhZfg9Cuk>

How not to scale your Prometheus: <https://www.youtube.com/watch?v=5BqZqGcM4-g>

We used 2.44 version Prometheus in a 3 node cluster (one master and 2 worker node) for running all these tests.

2. WHY SHOULD WE OPTIMISE PROMETHEUS MEMORY USAGE?

In Kubernetes, many components expose metrics that Prometheus scrapes. But with hundreds of metrics coming in, memory consumption can spike unexpectedly. Prometheus stores these metrics as time series in its TSDB (time series database), but not all of them are useful. And let's be honest—nobody wants to sit down and manually sift through every metric to decide what's important and what isn't. Optimising memory is crucial not only for improving performance but also for keeping your infrastructure costs under control. So, let's dive in and see how you can make your Prometheus instance more efficient.

3. OUR APPROACH TO PROMETHEUS MEMORY OPTIMISATION

The smart way to optimise memory is to target the components that have the highest number of time series and focus on removing unnecessary metrics from them. This is more efficient than trying to deal with every single metric, especially in production environments. Components with a lesser number of time series count usually don't impact memory significantly, so it's better to let them be.

Now, if you find a metric useful but think some of its labels are unnecessary, you can remove the labels instead of dropping the metric entirely. This can be done through metric relabeling which we are going to talk about in later sections.

Before diving into relabeling, we have compiled a list of metrics that consume a lot of time series but might not be useful. Note that this is based on our own research and what metrics we think are redundant, so we encourage you to double-check before dropping anything.

3.1 Shortlisting metrics with most timeseries per component

To identify these metrics, you can use a command like this:

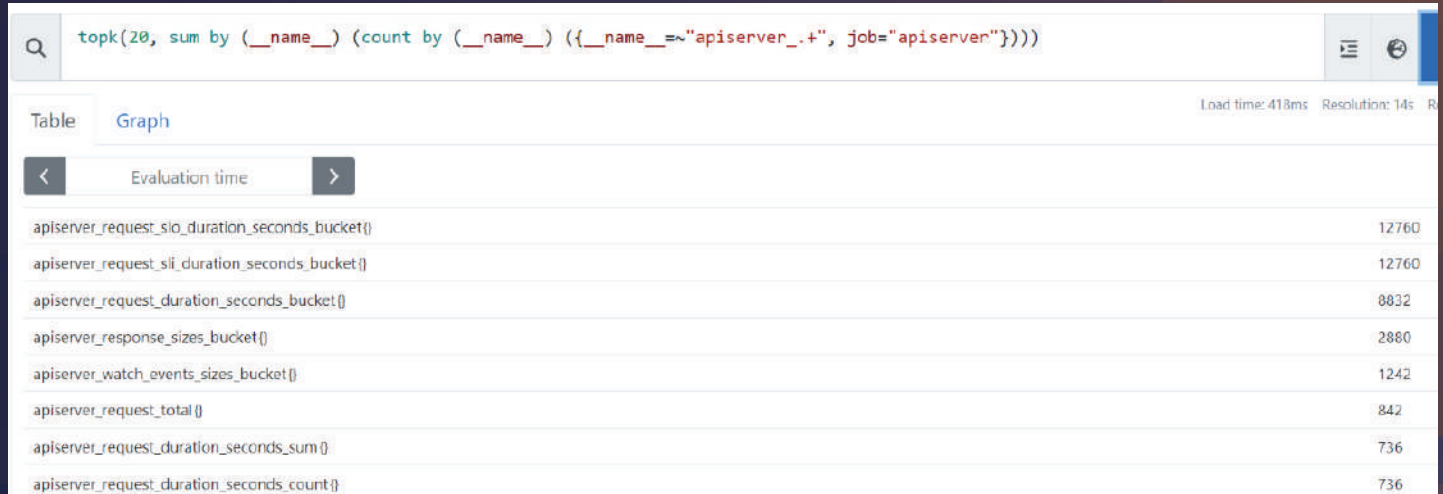
```
topk(20, sum by (__name__) (count by (__name__) ({__name__=~"apiserver_.+", job="apiserver"})))
```

However, there's a catch - the above command works, but the apiserver metrics might be used by other components, not just the apiserver itself. So, it's better to refine the query to something like this:

```
topk(20, sum by (__name__) (count by (__name__) ({__name__=~".+", job="apiserver"})))
```

For other components, I've updated this approach.

apiserver:



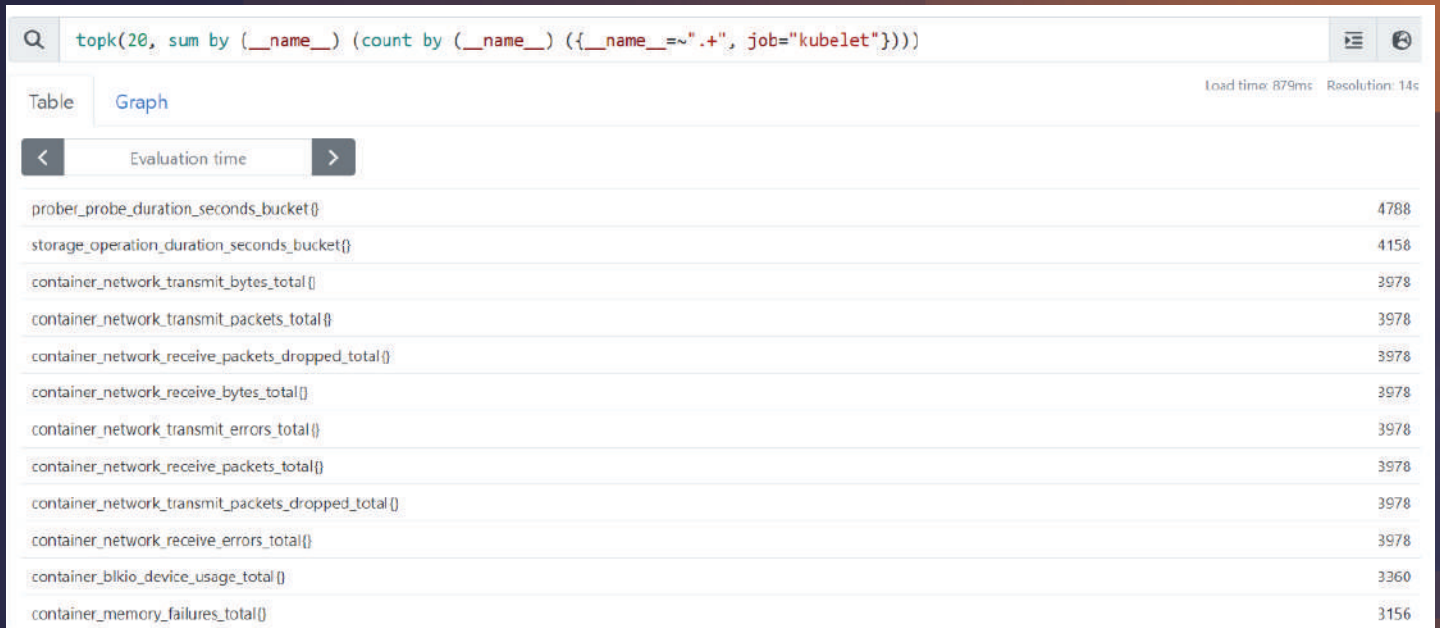
Dropped metrics/labels:

apiserver_request_slo_duration_seconds_bucket

Metric	Labels dropped
apiserver_request_duration_seconds_bucket	Component, endpoint, dry_run
apiserver_request_slo_duration_seconds_bucket	NA
apiserver_request_sli_duration_seconds_bucket	NA
apiserver_response_sizes_bucket	NA
apiserver_watch_events_sizes_bucket	NA

(NA indicate whole metric is dropped)

Kubelet:



Dropped metrics/labels:

Metric	Labels dropped
container_blkio_device_usage_total	id(because each container has got unique id)


kube-state-metrics:



Dropped metrics/labels:

Metric	Labels dropped
kube_pod_status_reason	NA
kube_pod_tolerations	NA
kube_pod_status_qos_class	NA
kube_pod_container_status_waiting_reason	NA
kube_pod_init_container_status_waiting_reason	NA

Istio:



Table

Graph

<

Evaluation time

>

istio_request_duration_milliseconds_bucket{}

istio_request_bytes_bucket{}

istio_response_bytes_bucket{}

envoy_listener_manager_lds_update_duration_bucket{}

envoy_cluster_upstream_cx_length_ms_bucket{}

envoy_cluster_manager_cds_update_duration_bucket{}

envoy_cluster_upstream_cx_connect_ms_bucket{}

envoy_server_initialization_time_ms_bucket{}

istio_request_duration_milliseconds_sum{}

istio_request_duration_milliseconds_count{}

Dropped metrics/labels:

Metric	Labels dropped
istio_request_duration_milliseconds_bucket	le,request-protocol,source-app, source-canonical-serivce, source-worload, response flags, reporter
istio_request_bytes_sum	le, request-protocol, source-app,source-canonical-serivce, source-worload , response flags ,reporter

Metric	Labels dropped
istio_request_bytes_bucket	NA
istio_response_bytes_bucket	NA
envoy_listener_manager_lds_update_duration_bucket	NA
envoy_cluster_manager_cds_update_duration_bucket	NA
envoy_server_initialisation_time_ms_bucket	NA

If you're unsure which metrics to drop, start here. But remember, this is just a starting point—don't rely solely on it when deciding which metrics to drop.

<https://grafana.com/blog/2021/07/02/how-to-quickly-find-unused-metrics-and-get-more-value-from-grafana-cloud/>

3.2 Finding unused metrics

Generally speaking, the metrics exposed by each component are primarily consumed in one of two ways: through Grafana dashboards or via Alertmanager rules. To optimise memory effectively, we need to evaluate both.

Let's start with the Grafana dashboards. Instead of manually checking each dashboard's JSON, which can be time-consuming, you can identify unused metrics by finding which metrics are actively used across all your dashboards. To simplify this process, you can use a tool called Cortex.

Installing cortex

First, download and install **Cortex**:

```
$ curl -fSL -o "cortextool"
```

```
"https://github.com/grafana/cortex-tools/releases/download/v0.11.0/cortextool_v0.11.0_Linux_x86_64"
```

```
$ chmod a+x "cortextool"
```

```
$ ./cortextool -help
```

Using cortex with Grafana

As of Grafana version 9.1, the API key method has been replaced by service account tokens. To generate a token:

Navigate to the Service Accounts section under the **Administration dropdown** in Grafana

1. Create a service account token to use as your Grafana key.

With your new token, you can run Cortex to analyse your Grafana dashboards and output a list of used metrics:

```
./cortextool analyse grafana --address=http://grafanaserviceip:port --key=<your_key> --output=example.json
```

This command generates a JSON file (example.json) listing all metrics currently used in your Grafana dashboards.

By comparing this list with the metrics collected by Prometheus, you can identify and remove those that are no longer in use, streamlining your memory usage and improving overall performance.

We should now check with alert manager rules

For this, we'll use the [Mimir](#) tool, which analyses rules under `/etc/prometheus/rules` and helps you find unused metrics in a clear and readable format.

<https://Oxdc.me/blog/how-to-find-unused-prometheus-metrics-using-mimirtool/>

Installing Mimirtool

First, install Mimirtool by running the following commands:

```
bash
```

```
curl -fLo mimirtool
```

```
https://github.com/grafana/mimir/releases/latest/download/mimirtool-linux-amd64
```

```
chmod +x mimirtool
```

Next Steps: Analysing Prometheus rules

To analyse your Prometheus rules and identify unused metrics, follow these steps:

- 1. Extract your Prometheus rules using kubectl:**

```
kubectl exec -it <prometheus-pod-name> -n <namespace> -- sh -c 'for i in `find /etc/prometheus/rules/ -type f`; do cat $i; done' > my-prom-rules.yaml
```
- 2. Clean up the extracted rules file:**

```
sed -i -e 's/groups://g' -e '1s/^/groups:/' my-prom-rules.yaml
```
- 3. Use Mimirtool to analyse the rules:**

```
mimirtool analyse rule-file my-prom-rules.yaml
```

This will generate a `metrics-in-ruler.json` file in your current directory. This file lists all the metrics used in your Prometheus rules, making it easier to spot those that aren't in use.

By cross-referencing this with the metrics gathered by Prometheus, you can confidently decide which metrics to drop, further optimising your memory usage.

Now, let's get to the main topic - **Dropping Metrics and Labels**

Once you've narrowed down the list of metrics to drop, the next step is to actually configure Prometheus to not scrape them and/or remove them from the existing TSDB. Learn how you can do it in the next chapter.

4. HOW TO DROP METRICS AND LABELS FROM PROMETHEUS METRICS

To drop metrics or labels, you'll need to use the `metricRelabelConfigs` section in your ServiceMonitor, podmonitor or Prometheus scrape config's. It's important to note that this isn't the same as `relabel_config`, which applies during the scrape phase, before metrics are ingested. Since we're dealing with metrics that have already been scraped, we need to use `metricRelabelConfigs`, which is applied post-ingestion. Use `drop` action to remove entire metrics and `labeldrop` action to remove specific labels from metrics. Here's an example configuration

```
spec:
  endpoints:
    - bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
      metricRelabelings:
        - action: drop
          regex:
            (apiserver_request_slo_duration_seconds_bucket|apiserver_request_sli_duration_seconds_bucket|apiserver_response_sizes_bucket|apiserver_watch_events_sizes_bucket)
          sourceLabels:
            - __name__
        - action: labeldrop
          regex: (component|endpoint|dry_run)
```

Important catch: avoid redundancies!

Here's a crucial point that could save you from a big headache: In some clusters, instead of reducing memory usage, you might see a 2x increase. The reason? Redundant services sending metrics to Prometheus or duplicate ServiceMonitors capturing the same metrics with slight label variations.

This can occur when applying ServiceMonitors with updated configurations based on previous subsections. So, extra care is needed when modifying ServiceMonitors, PodMonitors, or Prometheus scrape configs to avoid unintentional duplication.

Why this matters:

Such duplicates can inadvertently inflate memory usage due to data duplication. To avoid this:

1. **Check for redundancies:** Use Prometheus' UI to verify that each metric is exposed by a single service.
2. **Consolidate monitors:** Ensure no duplicate ServiceMonitors/PodMonitors or Prometheus scrape config are adding the same metric.

This step is critical to truly optimising your memory usage without unintended consequences.

5. RESULTS

After applying the drop and labeldrop actions, we have observed a consistent pattern: initially, the time series count may increase temporarily and typically takes around 2 to 2.5 hours to drop below the original levels before the label drop.

Why it takes around 2 hours to reflect changes?

The slight memory increase after removing metrics and labels in metric relabeling is a common observation. This happens because Prometheus initially holds onto the dropped time series in memory for a short period, typically until they naturally expire from the database (around 2 hours). During this time, Prometheus may still track these series, even though they are no longer being updated.

5.1 Cluster 1

At 12:00, ServiceMonitor changes were applied in Cluster 1. As shown, there was an immediate spike in all metrics after 12:00. However, around 14:30, the figures started decreasing—head series dropped from 212k to 120k, and memory usage decreased by 100MB.



Figure 1: Headseries, headchunks in cluster 1

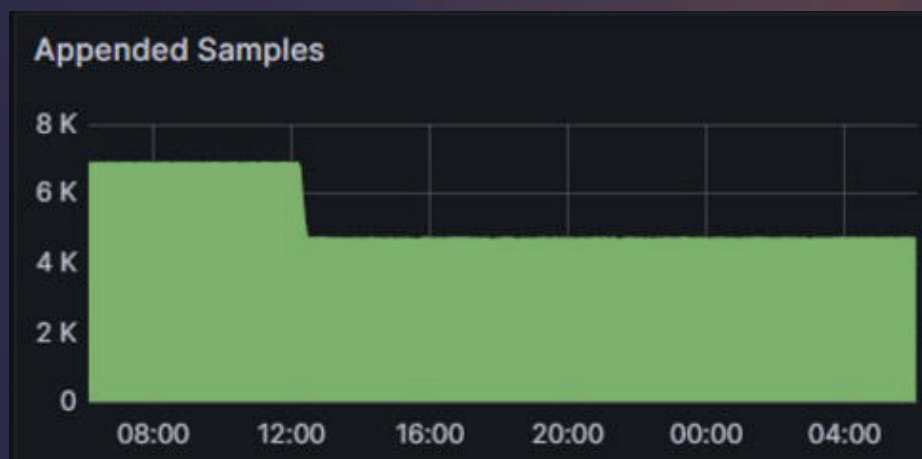


Figure 2: Appended samples in cluster 1

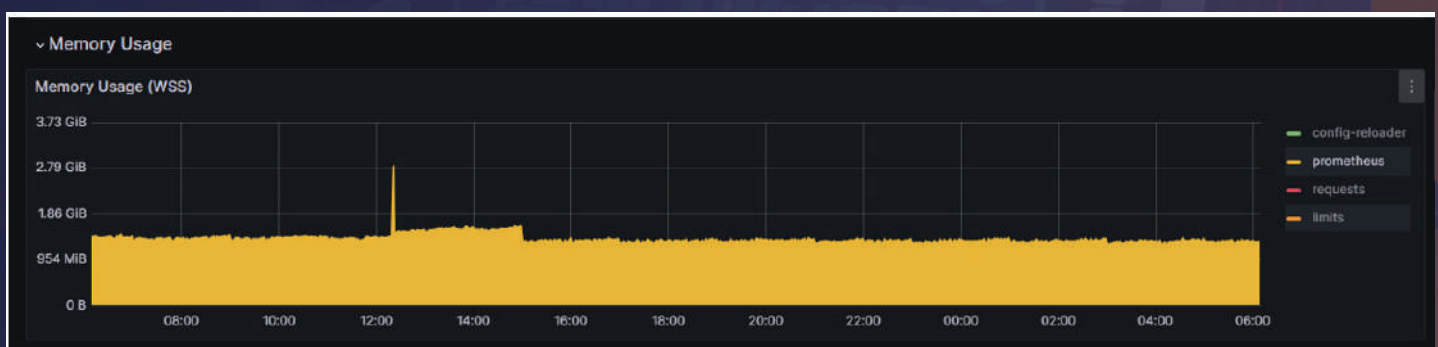


Figure 3: Memory usage in cluster 1

5.2 Cluster 2

At 10:30 08/16, ServiceMonitor changes were applied in Cluster 2. As shown, there was an immediate spike in all metrics after 10:30. However, around 12:00, the figures started decreasing—head series dropped from 260k to 160k, and memory usage decreased by 100MB.

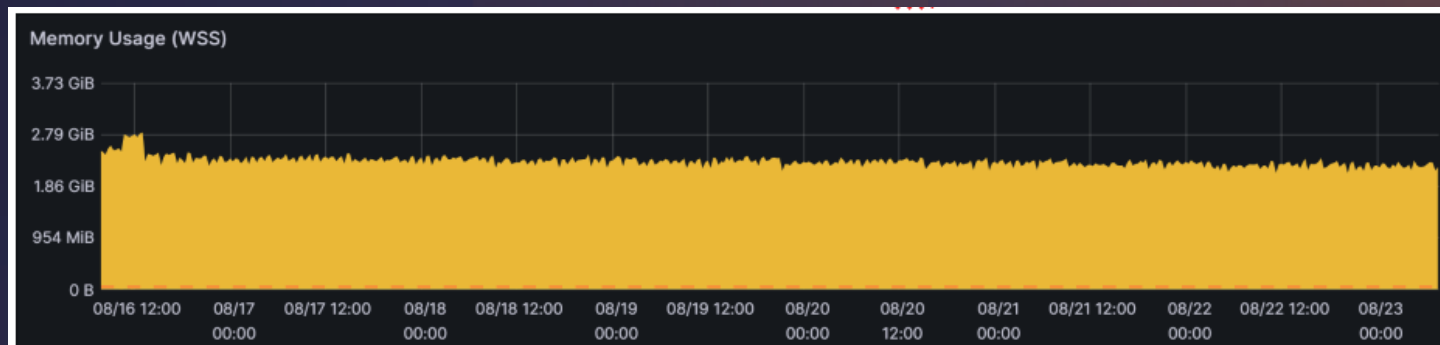


Figure 4: Memory usage in cluster 2

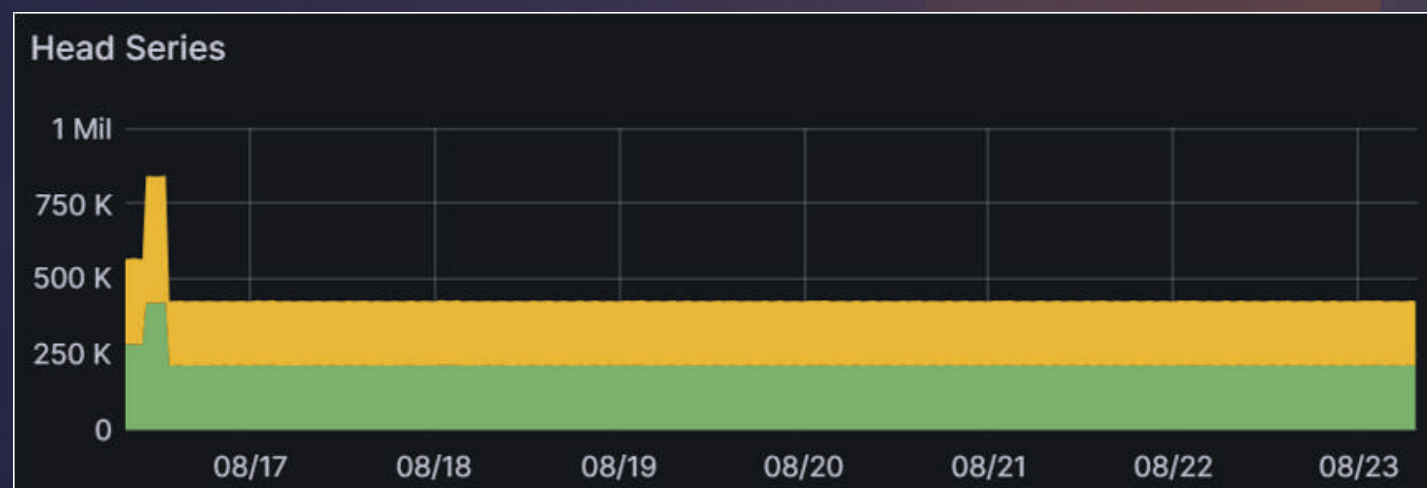


Figure 5: Headseries in cluster 2

Let's look at bigger clusters

5.3 Cluster 3

Before applying modified service monitors:

Below are the observations snapped at 11:00 to 12:00 before service monitors were applied.

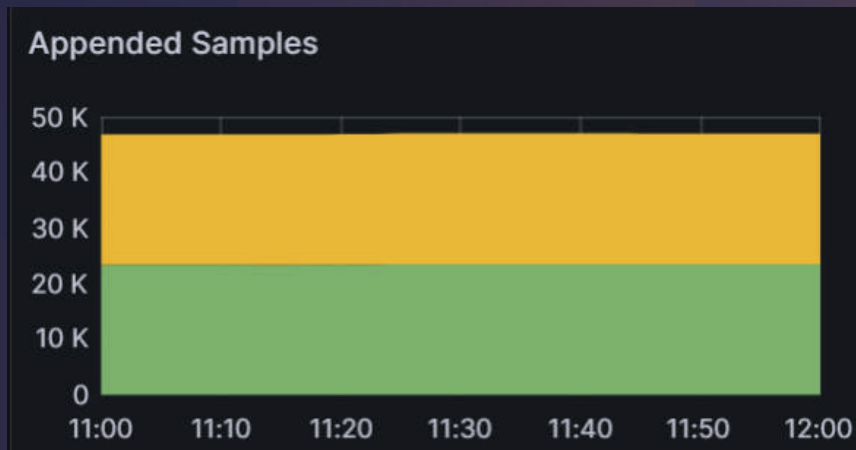


Figure 6: Appended samples in cluster 3 before applying modified service monitors

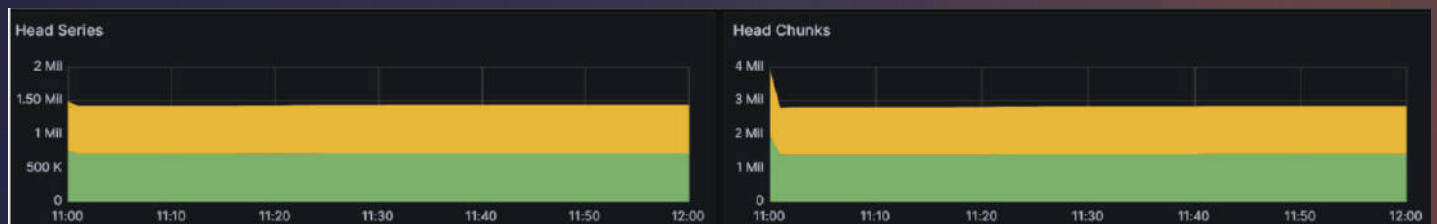


Figure 7: Headseries, headchunks in cluster 3 before applying modified service monitors

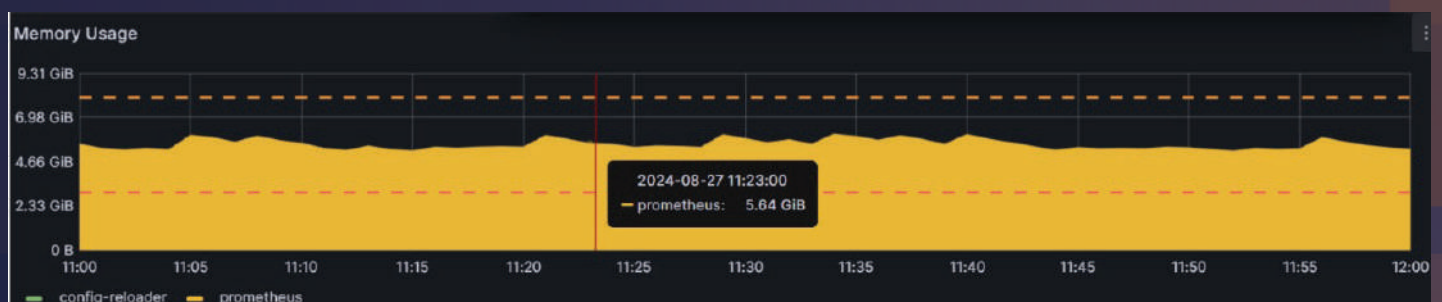


Figure 8: Memory usage in cluster 3 before applying modified service monitors

After applying modified service monitors:

Below are the observations snapped at 11:00 to 12:00 ,few days after applying service monitors as shown metrics started to reduce, appended samples reduced from 22k to 11k, Prometheus memory usage went down from 5.64gb to 4.54gb etc.

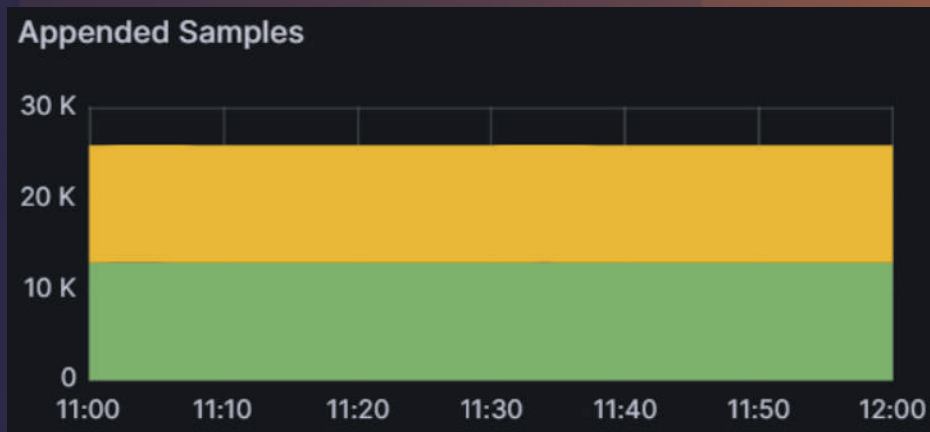


Figure 9: Appended samples in cluster 3 after applying modified service monitors



Figure 10: Headseries, headchunks in cluster 3 after applying modified service monitors

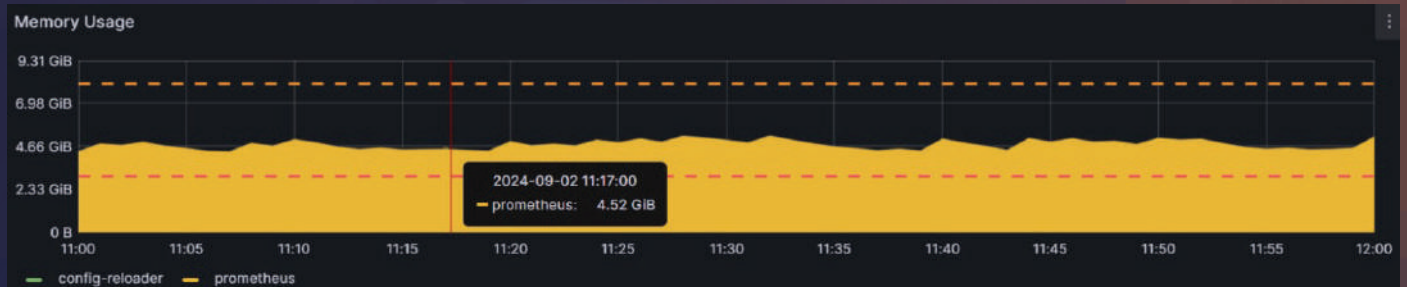


Figure 11: Memory usage in cluster 3 after applying modified service monitors

If you still feel the memory reduction isn't enough, you can revisit the methods mentioned earlier to analyse and drop more metrics and labels for further optimisation.