Comparing Actual Resource Usage with Defined Limits

In this lesson, we will see how to check if resource usage of our containers reaches the specified 'Limits'.

WE'LL COVER THE FOLLOWING

Discrepancy between requested and actual resource

Knowing when a container uses too much or too few resources compared to requests helps us be more precise with resource definitions and, ultimately, help Kubernetes make better decisions where to schedule our Pods. In most cases, having too big of a discrepancy between requested and actual resource usage will not result in malfunctioning. Instead, it is more likely to result in an unbalanced distribution of Pods or in having more nodes than we need. Limits, on the other hand, are a different story.

Resource usage reaching the specified `limits`

If resource usage of our containers enveloped as Pods reaches the specified `limits`, Kubernetes might kill those containers if there's not enough memory for all. It does that as a way to protect the integrity of the rest of the system. Killed Pods are not a permanent problem since Kubernetes will almost immediately reschedule them if there is enough capacity. If we do use Cluster Autoscaling, even if there isn't enough capacity, new nodes will be added as soon as it detects that some Pods are in the pending state (unschedulable). So, the world is not likely to end if resource usage goes over the limits.

Detecting Limit is reached

Nevertheless, killing and rescheduling Pods can result in downtime. There are, apparently, worse scenarios that might happen. But we won't go into them. Instead, we'll assume that we should be aware that a Pod is about to reach its limits, that we might want to investigate what's going on, and that we might need to take some corrective measures. Maybe, the latest release introduced a memory leak? Or perhaps, the load increased beyond what we expected and tested, and that results in increased memory usage. The cause of using memory close to the limit is not the focus right now. Detecting that we are reaching the limit is.

First, we'll go back to `Prometheus`'s graph screen.

```
open "http://$PROM_ADDR/graph"
```

We already know that we can get actual memory usage through the `container_memory_usage_bytes` metric. Since we already explored how to get requested memory, we can guess that limits are similar. Indeed, they are, and they can be retrieved through the `kube_pod_container_resource_limits_memory_bytes`. Since one of the metrics is the same as before, and the other is very similar, we'll jump straight into executing the full query.

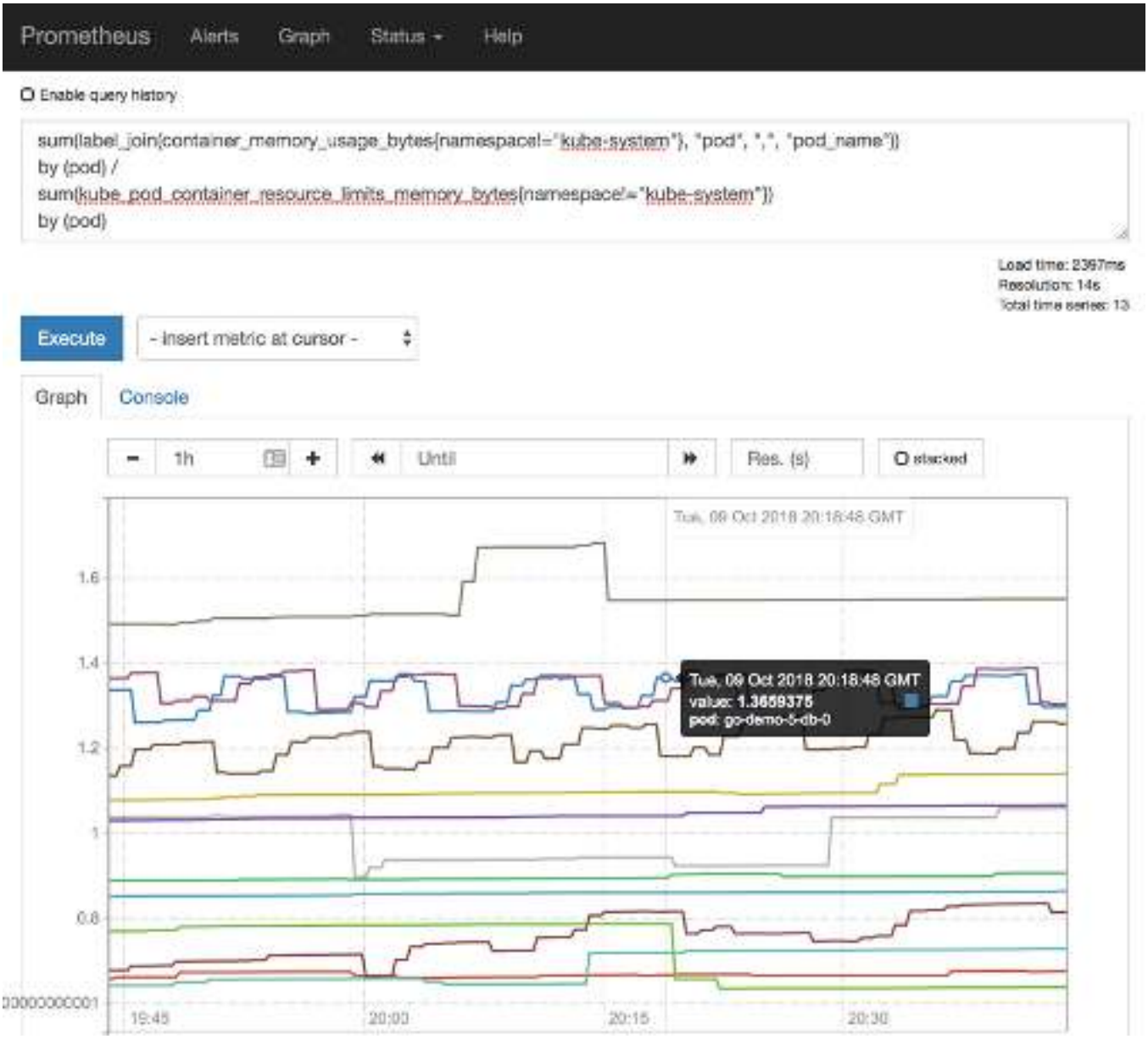
Executing the full query

Please type the expression that follows, press the *Execute* button, and switch to the Graph tab.

```
sum(label_join(
  container_memory_usage_bytes{
    namespace!="kube-system"
  },
  "pod",
  ",",
  "pod_name"
))
by (pod) /
sum(
  kube_pod_container_resource_limits_memory_bytes{
    namespace!="kube-system"
```

```
  }
)
by (pod)
```

In my case (screenshot below), we can see that quite a few Pods use more memory than what is defined as their `limits`. Fortunately, I have spare capacity in my cluster, and there is no imminent need for Kubernetes to kill any of the Pods. Moreover, the issue might not be that Pods are using more than what is set as their limits, but that not all containers in those Pods have the `limits` set. In either case, I should probably update the definition of those Pods/containers and make sure that their `limits` are above their average usage over a few days or even weeks.



Prometheus' graph screen with the percentage of container memory usage based on memory limits and with those from the kube-system Namespace excluded

Q

We can get actual memory usage through the

`container_memory_usage_bytes` metric.

COMPLETED 0%

1 of 1

Next, we'll go through the drill of exploring the diff between the old and the new version of the values.

```
diff mon/prom-values-req-cpu.yml \
    mon/prom-values-limit-mem.yml
```

The **output** is as follows.

```
175c175
<   for: 1m
---
>   for: 1h
184c184
<   for: 6m
---
>   for: 6h
190a191,199
> - alert: MemoryAtTheLimit
>   expr: sum(label_join(container_memory_usage_bytes{namespace!="kube-sys
tem"}, "pod", ",", "pod_name")) by (pod) / sum(kube_pod_container_resource
_limits_memory_bytes{namespace!="kube-system"}) by (pod) > 0.8
>   for: 1h
>   labels:
>     severity: notify
>     frequency: low
>   annotations:
>     summary: Memory usage is almost at the limit
>     description: At least one Pod uses memory that is close it its limit
```

Define new alert `MemoryAtTheLimit`

Apart from restoring sensible thresholds for the alerts we used before, we

defined a new alert called `MemoryAtTheLimit`. It will fire if the actual usage is over eighty percent (`0.8`) of the `limit` for more than one hour (`1h`).

Next is the upgrade of our `Prometheus`'s Chart.

```
helm upgrade prometheus \
  stable/prometheus \
  --namespace metrics \
  --version 9.5.2 \
  --set server.ingress.hosts={$PROM_ADDR} \
  --set alertmanager.ingress.hosts={$AM_ADDR} \
  -f mon/prom-values-limit-mem.yml
```

Finally, we can open the `Prometheus`'s alerts screen and confirm that the new alert was indeed added to the mix.

```
open "http://$PROM_ADDR/alerts"
```

We won't go through the drill of creating a similar alert for CPU. You should know how to do that yourself.

In the next lesson, we will revise and test the concepts of this third chapter through a short quiz.