Kubernetes in Production: The Ultimate Guide to Monitoring Resource Metrics with Prometheus

In this instalment of the Kubernetes in Production blog series, we take a deep dive into monitoring Kubernetes resource metrics. We will see why monitoring resource metrics is important for Kubernetes in production, choose resource metrics to monitor, set up the tools required including Metrics-Server and Prometheus and query those metrics.

So, you've launched your Kubernetes clusters in production. Everything is nominal, and you can sit back and relax, right? Wrong! To borrow from SpaceX, you never know when there might be a "rapid unscheduled disassembly", or as regular people say; things might blow up.

While you can definitely relax it is a great idea to monitor your clusters and avoid any unpleasant unscheduled disassemblies.

Why Monitor Kubernetes Resource Metrics in Production?

Every Kubernetes administrator has stumbled across the "Failed Scheduling No nodes are available that match all of the following predicates:: Insufficient memory" or "Failed Scheduling No nodes are available that match all of the following predicates:: Insufficient CPU" error message.

It is obvious that these errors are caused by insufficient CPU or memory resources. Such issues can be avoided by having a real-time view of resource usage and availability for individual nodes or pods.

Monitoring Kubernetes is also important in the context of resource usage, utilization and cost control. Kubernetes clusters have to be actively managed to ensure pods utilize underlying node resources efficiently. The same is true of resources allocated to individual containers or namespaces.

Below we will review some best practices for monitoring Kubernetes resource metrics in production. We will go through the entire process of setting up the tools required, choosing which metrics to monitor as well as deploying dashboards.

Monitoring Kubernetes metrics is just the start though. Ensuring that Kubernetes clusters utilize underlying resources efficiently also requires active management and continuous optimization based on historical data. Replex's Kubernetes solution does this by collecting and analyzing Kubernetes resource metrics to flag idle resources, dynamically optimize infrastructure footprint and cut costs.

Which Resource Metrics Should I be Monitoring?

The Utilization Saturation and Errors (USE) Method gives us an idea of the performance metrics we should be tracking for Kubernetes in production. The USE method was developed by Brendann Egg to analyze the performance of any system in production. It recommends tracking utilization, saturation and errors for all resources.

We can start with OS and hardware metrics exposed by the underlying Linux kernels of the containers running inside Kubernetes clusters. However, tracking resource metrics for OS and hardware is not enough in the Kubernetes context. For these metrics to be meaningful we need to co-relate them to Kubernetes primitives.

Kubernetes introduces a number of abstractions that we need to consider when choosing which metrics to track in production. These are abstractions on both the hardware as well as software layer including nodes, pods, containers, namespaces and clusters.

The resource management model of Kubernetes also has to be considered. Kubernetes has two types of resources that can be consumed by containers: CPU and memory. Container resource consumption can be managed using

resource requests and resource limits. Resource requests are the minimum amount of resources requested for containers, limits, on the other hand, are the maximum amount that can be consumed by containers.

Bringing all of these concepts together results in the following exhaustive list of resource metrics that we can monitor in production. The list takes into consideration both OS and hardware metrics as well as the specific hardware and software abstractions introduced by Kubernetes and the resource management model.

Ready for Production? Download the Complete Production Readiness Checklist with Checks, Recipes and Best Practices for Availability, Resource Management, Security, Scalability and Monitoring

Below is the list with brief descriptions of each resource metric:

Kubernetes Container CPU and Memory Usage

Containers are the building blocks of containerized applications. Container CPU usage refers to the amount of CPU resources consumed by containers in production. Memory usage is the measure of memory resources consumed. CPU resources are measured in CPU cores while memory is measured in bytes

Kubernetes Pod CPU and Memory Usage

Pods are collections of containers and as such pod CPU usage is the sum of the CPU usage of all containers that belong to a pod. Similarly, pod memory usage is the total memory usage of all containers belonging to the pod

Kubernetes Node CPU and Memory Usage

Node CPU usage is the number of CPU cores being used on the node by all pods running on that node. Similarly node memory usage is the total memory usage of all pods

Kubernetes Namespace CPU and Memory Usage

You can think of Kubernetes namespaces as boxes. DevOps can create separate boxes to isolate the resources belonging to individual applications or teams. Namespace resource usage is the sum of CPU or memory usage of all pods that belong to that namespace

Kubernetes Cluster CPU and Memory Usage

The sum of CPU or memory usage of all pods running on nodes belonging to the cluster gives us the CPU or memory usage for the entire cluster

Kubernetes Container CPU and Memory Requests

You can think of container resource requests as a soft limit on the amount of CPU or memory resources a container can consume in production

Kubernetes Pod CPU and Memory Requests

The sum of CPU or memory requests of all containers belonging to a pod

Kubernetes Node CPU and Memory Requests

Node CPU requests are a sum of the CPU requests for all pods running on that node. Similarly, node memory requests are a sum of memory requests of all pods

Kubernetes Namespace CPU and Memory Requests

The sum of CPU or memory requests of all pods belonging to a namespace

Kubernetes Cluster CPU and Memory Requests

Sum of CPU requests or memory requests for all pods running on nodes belonging to a cluster

Kubernetes Container CPU and Memory Limits

Container CPU limits are a hard limit on the amount of CPU a container can consume in production. Memory limits are the maximum amount of bytes that a container can consume in production

Kubernetes Pod CPU and Memory Limits

The sum of CPU or memory limits for all containers belonging to a pod

Kubernetes Node CPU and Memory Limits

Node CPU limits are the sum of CPU limits for all pods running on that specific node. Node memory limits on the other hand, are a sum of the memory limits of all pods

Kubernetes Namespace CPU and Memory Limits

Sum of CPU or memory limits of all pods belonging to a namespace

Kubernetes Cluster CPU and Memory Limits

Sum of CPU or memory limits for all pods running on nodes belonging to a cluster

Kubernetes Resource Capacity

In cloud environments, Kubernetes nodes usually refer to cloud provider instances. Node CPU capacity is the total number of CPU cores on the node. Node memory capacity is the total number of bytes available on the node. For example, an N1-standard-1 instance has 2 vCPUs and 3.75 GB of memory

Cluster CPU capacity is the sum of CPU capacities of all Kubernetes nodes that are part of the cluster. Cluster memory capacity is the sum of memory capacities of all nodes belonging to the cluster. For example a cluster with 4 N1-standard-1 instances has 8vCPUs and 15 GB of memory

Kubernetes CPU and Memory Request Commitment (Resource Requests vs Capacity)

CPU request commitment is the ratio of CPU requests for all pods running on a node to the total CPU available on that node. In the same way, memory commitment is the ratio of pod memory requests to the total memory capacity of that node. We can also calculate cluster level request commitment by comparing CPU/memory requests on a cluster level to total cluster capacity

Request commitments give us an idea of how much of the node or cluster is committed in terms of soft resource usage limits

Kubernetes CPU and Memory Limit Commitment (Resource Limits vs Capacity)

CPU limit commitment is the ratio of CPU limits for all pods running on a node to the total CPU available on that node. Similarly, memory commitment is the ratio of pod memory limits to the total memory capacity of that node. Cluster level limit commitments can be calculated by comparing total CPU/memory limits to cluster capacity

Limit commitments give us an idea of how much of the node or cluster is committed in terms of hard CPU and memory limits

Kubernetes Resource Utilization (Resource Usage vs Capacity)

With the emphasis on pay as you go billing models for cloud deployments, resource utilization is an important metric to monitor and has major implications for cost control

CPU utilization is the ratio of CPU resources being currently consumed by all pods running on a node to the total CPU available on that node. Memory utilization is the ratio of memory usage by all pods to the total memory capacity of that node

Cluster resource utilization will compare resource usage (both CPU and memory) for all pods with the total resource capacity of all nodes

Kubernetes Resource Saturation

Node CPU saturation is the measure of requests for CPU which cannot be fulfilled because of unavailability. Similarly, memory saturation on the node is a measure of the memory requests which cannot be met due to unavailability

Saturation on the cluster level can be calculated based on saturation numbers across all nodes.

Now that we have identified Kubernetes specific resource metrics we can take a look at some of the tools that will help us monitor these resources in production.

Which Tools Can I Use to Monitor Kubernetes Metrics?

There are a number of tools that can be used to monitor Kubernetes in production. These include cAdvisor, Heapster, Prometheus, metrics-server and kube-state-metrics.

Let's start by taking a look at metrics-server.

Monitoring Resource Metrics with Metrics-Server

Metrics-server allows us to query resource metrics using kubectl right from the command line. However, It gives us access to only some of the metrics outlined above.

Follow the steps below to install metrics-server in your kubernetes cluster. We assume that you already have a Kubernetes cluster up and running and have kubectl installed.

```
git clone https://github.com/kubernetes-incubator/metrics-server
cd metrics-server/deploy
kubectl create -f 1.8+/
```

Once you do this, you will see an instance of the metrics-server running in the kube-system namespace

```
kubectl get pods -n kube-system
```

```
hasham@DESKTOP-PEGP5DE:~$ kubectl get pods -n kube-system
NAME                                                READY   STATUS    RESTA
dns-controller-6d6b7f78b-zqqm7                      1/1     Running   0
etcd-server-events-master-us-central1-a-l5td        1/1     Running   0
etcd-server-master-us-central1-a-l5td               1/1     Running   0
kube-apiserver-master-us-central1-a-l5td            1/1     Running   0
kube-controller-manager-master-us-central1-a-l5td   1/1     Running   0
kube-dns-5fbcb4d67b-c7lfp                           3/3     Running   0
kube-dns-5fbcb4d67b-fqj7z                           3/3     Running   0
kube-dns-autoscaler-6874c546dd-tlxfp                1/1     Running   0
kube-proxy-master-us-central1-a-l5td                1/1     Running   0
kube-proxy-nodes-rzg7                               1/1     Running   0
kube-proxy-nodes-sf4b                               1/1     Running   0
kube-proxy-nodes-v0dr                               1/1     Running   0
kube-scheduler-master-us-central1-a-l5td            1/1     Running   0
metrics-server-749b7b675d-xpkcj                     1/1     Running   0
tiller-deploy-778f674bf5-p8f77                      1/1     Running   0
```

However, If you try to access the logs for the metrics-server using `kubectl logs -n kube-system metrics-server-749b7b675d-xpkcj` you might see the `unable to fetch pod metrics for pod <namespace_name>/<pod_name>` message.

You will also get the error message `error: Metrics not available for pod <namespace_name>/<pod_name>` if you query `kubectl top pod` from the command line.

To get around this edit the metrics-server deployment config

```
kubectl edit deploy -n kube-system metrics-server
```

and add the following under `.spec.template.spec.containers`

```
command:
- /metrics-server
- --kubelet-insecure-tls
```

Passing these flags should only be considered as a work-around for test clusters. In production Kubernetes environments you should consider setting up the node's serving certificates to list the internal IP as an alternate name.

Now we are ready to query resource metrics using kubectl from the command line. Let's start with resource usage per pod.

Resource Metric: Kubernetes Pod Resource Usage

The following command will give us both the CPU usage as well as the memory usage for all pods in the default namespace.

```
kubectl top pod
```

```
hasham@DESKTOP-PEGP5DE:~$ kubectl top pod
NAME                        CPU(cores)   MEMORY(bytes)
frontend-b59484f56-2d5r7    1m           5Mi
frontend-b59484f56-2dpps    1m           5Mi
frontend-b59484f56-4bqnv    1m           5Mi
frontend-b59484f56-5cq2w    1m           5Mi
frontend-b59484f56-5jqmq    1m           5Mi
frontend-b59484f56-66xq9    1m           5Mi
frontend-b59484f56-68gqm    1m           5Mi
frontend-b59484f56-6dk8p    1m           5Mi
```

To get the resource usage for pods in another namespace use

```
kubectl top pod --namespace=<namespace_name>
```

For example we can see the resource usage for all pods in kube-system namespace using:

```
kubectl top pod --namespace=kube-system
```

```
hasham@DESKTOP-PEGP5DE:~$ kubectl top pod --namespace=kube-system
NAME                                                CPU(cores)   MEMORY(byt
dns-controller-6d6b7f78b-v8tbk                      1m           7Mi
etcd-server-events-master-us-central1-a-ctqs        3m           11Mi
etcd-server-master-us-central1-a-ctqs               7m           59Mi
kube-apiserver-master-us-central1-a-ctqs            29m          454Mi
kube-controller-manager-master-us-central1-a-ctqs   37m          54Mi
kube-dns-5fbcb4d67b-hkfvv                           2m           21Mi
kube-dns-5fbcb4d67b-zwbpr                           2m           21Mi
kube-dns-autoscaler-6874c546dd-hqtxr                1m           6Mi
kube-proxy-master-us-central1-a-ctqs                3m           10Mi
kube-proxy-nodes-c472                               2m           10Mi
kube-proxy-nodes-wns0                               4m           10Mi
kube-proxy-nodes-wzgd                               2m           10Mi
kube-scheduler-master-us-central1-a-ctqs            13m          16Mi
metrics-server-6bbc4bb954-lcgkm                     1m           15Mi
```

To see the CPU and memory usage for the individual containers of a pod use:

```
kubectl top pod <pod_name> --containers

kubectl top pod my-pod --containers
```

```
hasham@DESKTOP-PEGP5DE:~$ kubectl top pod my-pod --containers
POD       NAME   CPU(cores)   MEMORY(bytes)
my-pod    2nd    929m         0Mi
my-pod    1st    0m           1Mi
```

Resource Metric: Kubernetes Node Resource Usage

To get the CPU and memory usage for individual nodes use the following command

```
kubectl top node
```

```
hasham@DESKTOP-PEGP5DE:~$ kubectl top node
NAME                        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
master-us-central1-a-ctqs   125m         6%     2822Mi          38%
nodes-c472                  44m          4%     1623Mi          45%
nodes-wns0                  35m          3%     1689Mi          46%
nodes-wzgd                  1000m        100%   1607Mi          44%
```

We can also see the CPU and memory usage for individual nodes by specifying a node name:

```
kubectl top node <node_name></code>
kubectl top node nodes-c472
```

```
hasham@DESKTOP-PEGP5DE:~$ kubectl top node nodes-c472
NAME         CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
nodes-c472   44m          4%     1622Mi          44%
```

Monitoring Resource Metrics with Prometheus

Next we will look at Prometheus which has become something of a favourite among DevOps. We will install Prometheus using Helm and the Prometheus operator. The Prometheus operator is a Kubernetes specific project that makes it easy to set up and configure Prometheus for Kubernetes clusters.

Let's first install Helm. To do that, head over here and download the latest version. I downloaded the helm-v2.11.0-linux-amd64.tar.gz version.

Next, unpack it:

```
tar -zxvf helm-v2.11.0-linux-amd64.tar.gz
```

Move it to your bin directory:

```
mv linux-amd64/helm /usr/local/bin/helm
```

Initialize helm and install tiller:

```
helm init
```

Create a service account:

```
kubectl create serviceaccount --namespace kube-system tiller
```

Bind the new service account to the cluster-admin role. This will give tiller admin access to the entire cluster:

```
kubectl create clusterrolebinding tiller-cluster-rule --clusterrole=cluster-admin --serviceaccount=kube-system:tiller
```

Deploy tiller and add the line serviceAccount: tiller to spec.template.spec:

```
kubectl patch deploy --namespace kube-system tiller-deploy -p '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'
```

Now we are ready to install the Prometheus operator:

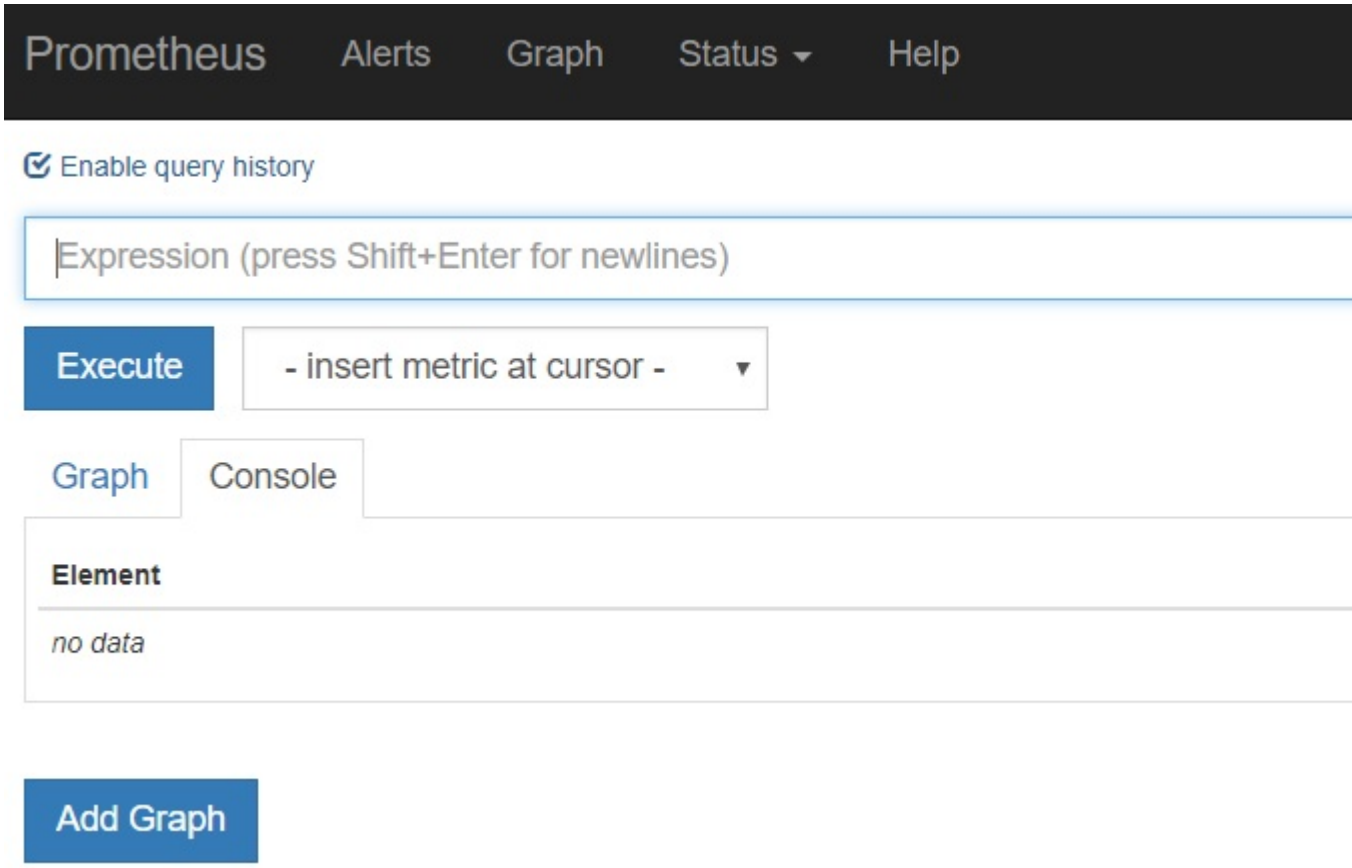
```
helm install --name prom-operator stable/prometheus-operator --namespace monitoring
```

This will create a separate namespace monitoring and will install the Prometheus operator in that namespace. It will also install Grafana, Kube-state-metrics and the Prometheus alert manager in the same namespace.

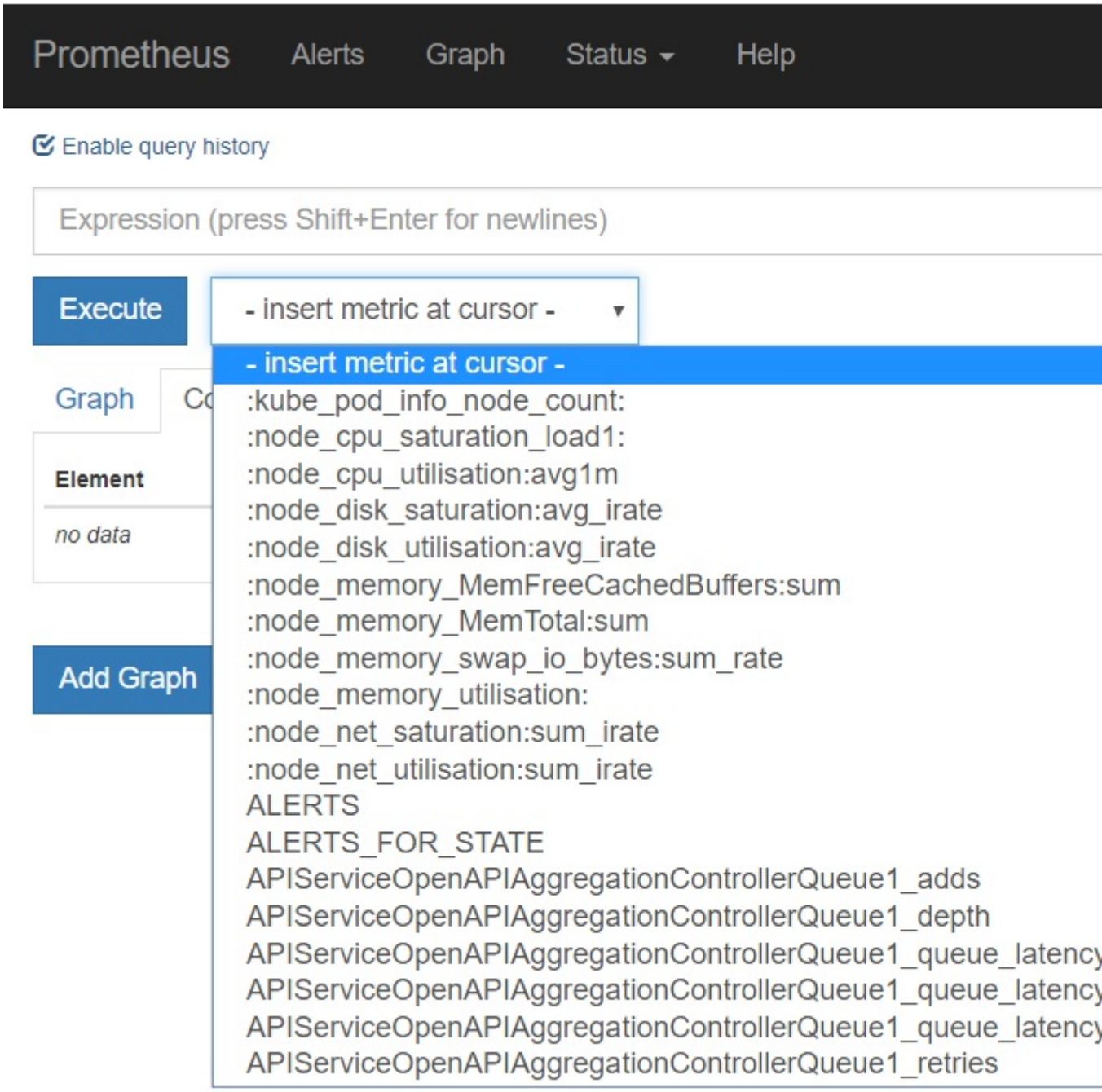
Once the Prometheus operator is installed we can forward the Prometheus server pod to a port on our local machine

```
kubectl port-forward -n monitoring prometheus-prom-operator-prometheus-o-prometheus-0 9090:9090
```

Access the Prometheus dashboard by navigating to http://localhost:9090



On the Prometheus dashboard, we can query metrics by typing them in the query field and pressing execute. Metrics can also be queried by choosing one from the drop down list shown below:



Now we are ready to start querying metrics. Below we will go through some of the resource metrics we identified earlier for containers, pods, namespaces, nodes and clusters. We will outline the metric name and the expression you can use in Prometheus to query it followed by a screenshot.

Resource Metric: Kubernetes CPU Usage

Expression:

```
sum(rate(container_cpu_usage_seconds_total{container_name!="POD",pod_name!=""}[5m]))
```


Prometheus
Alerts
Graph
Status
Help
Enable query history
sum(rate(container_cpu_usage_seconds_total{container_name!="POD",pod_name!=""}[5m])
Execute
- insert metric at cursor -
Graph
Console
1h
Until
Res. (s
0.21
0.205
0.2
0.195
14:30
{}
Add Graph

The screenshot above shows us the CPU usage for all pods belonging to the cluster.

Resource Metric: Kubernetes Container CPU usage

Expression:

```
rate(container_cpu_usage_seconds_total{container_name!="POD",pod_name!=""}[5m])
```

OR

```
sum(rate(container_cpu_usage_seconds_total{container_name!="POD",pod_name!=""}[5m])) by (container_name)
```

Prometheus Alerts Graph Status Help

Enable query history

sum(rate(container_cpu_usage_seconds_total{container_name!="POD",pod_name

Execute - insert metric at cursor -

Graph Console

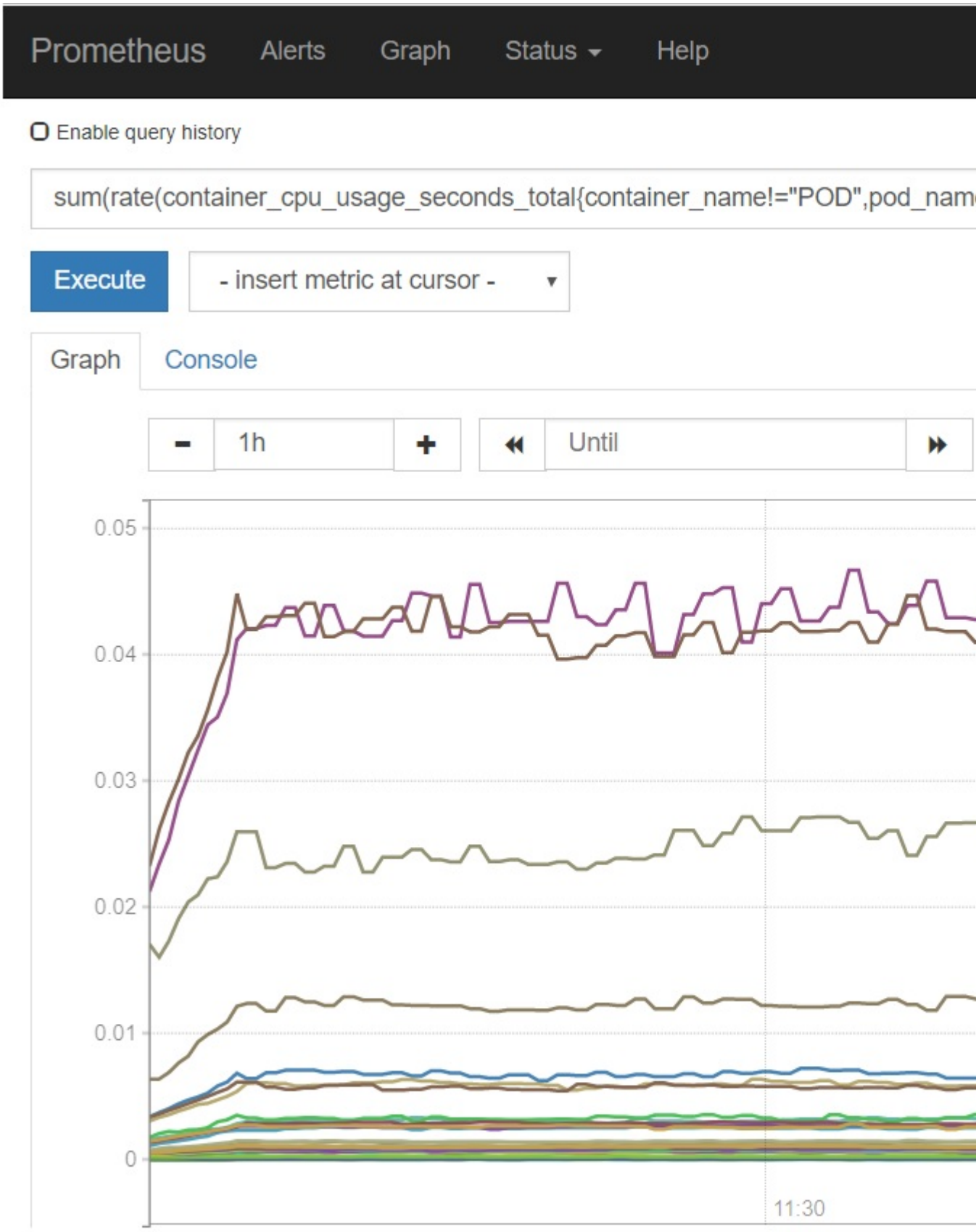
Element
{container_name="grafana-sc-datasources"}
{container_name="kube-controller-manager"}
{container_name="kube-proxy"}
{container_name="alertmanager"}
{container_name="dns-controller"}
{container_name="dnsmasq"}
{container_name="etcd-container"}
{container_name="grafana"}
{container_name="node-exporter"}
{container_name="php-redis"}
{container_name="prometheus"}
{container_name="prometheus-config-reloader"}
{container_name="sidecar"}
{container_name="autoscaler"}
{container_name="config-reloader"}
{container_name="grafana-sc-dashboard"}
{container_name="kube-state-metrics"}
{container_name="master"}

The first expression will show CPU usage for individual instances of each container. The second expression aggregates CPU usage for all containers with the same name.

Resource Metric: Kubernetes Pod CPU usage

Expression:

```
sum(rate(container_cpu_usage_seconds_total{container_name!="POD",pod_name!=""}[5m])) by (pod_name)
```


Prometheus
Alerts
Graph
Status
Help
Enable query history
sum(rate(container_cpu_usage_seconds_total{container_name!="POD",pod_nam
Execute
- insert metric at cursor -
Graph
Console
1h
Until
0.05
0.04
0.03
0.02
0.01
0
11:30

The screenshot above shows the CPU usage for each individual pod. It is a sum of the CPU usage of each container belonging to the pod.

Resource Metric: Kubernetes Namespace CPU usage

Expression:

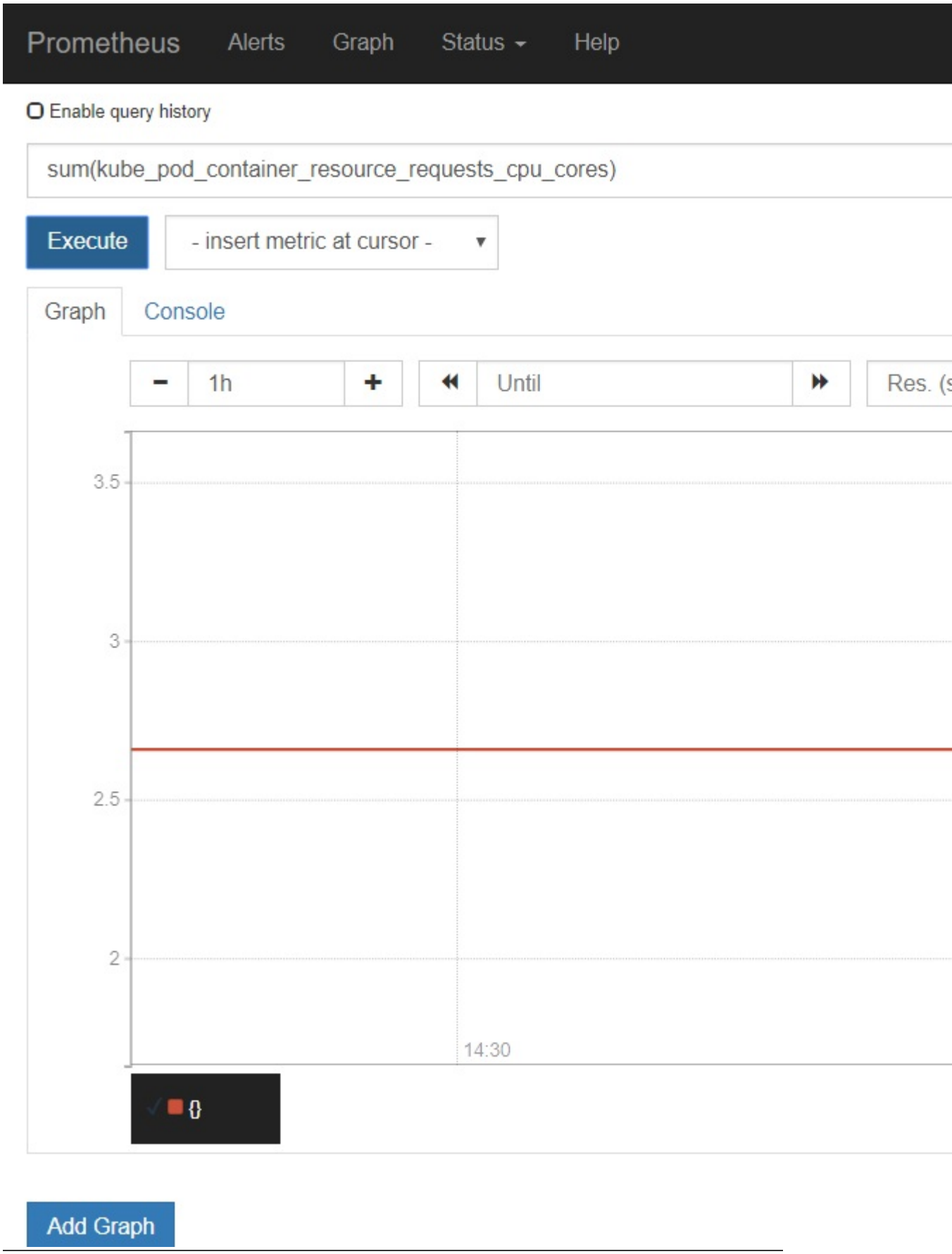
```
sum(rate(container_cpu_usage_seconds_total{container_name!="POD",namespace!=""}[5m])) by (namespace)
```


Prometheus
Alerts
Graph
Status
Help
Enable query history
sum(rate(container_cpu_usage_seconds_total{container_name!="POD",namespace!=""}[5m
Execute
- insert metric at cursor -
Graph
Console
1h
Until
Res. (s
0.1
0.05
0
11:30
11:45
{namespace="production"}
{namespace="monitoring"}
{namespace="kube-system"}
{namespace="development"}

Resource Metric: Kubernetes CPU requests

Expression:

```
sum(kube_pod_container_resource_requests_cpu_cores)
```


Prometheus
Alerts
Graph
Status
Help
Enable query history
sum(kube_pod_container_resource_requests_cpu_cores)
Execute
- insert metric at cursor -
Graph
Console
1h
Until
Res. (s
3.5
3
2.5
2
14:30
{}
Add Graph

Resource Metric: Kubernetes Pod CPU requests

Expression:

```
sum(kube_pod_container_resource_requests_cpu_cores) by (pod)
```

Prometheus Alerts Graph Status Help

Enable query history

```
sum(kube_pod_container_resource_requests_cpu_cores) by (pod)
```

Execute - insert metric at cursor -

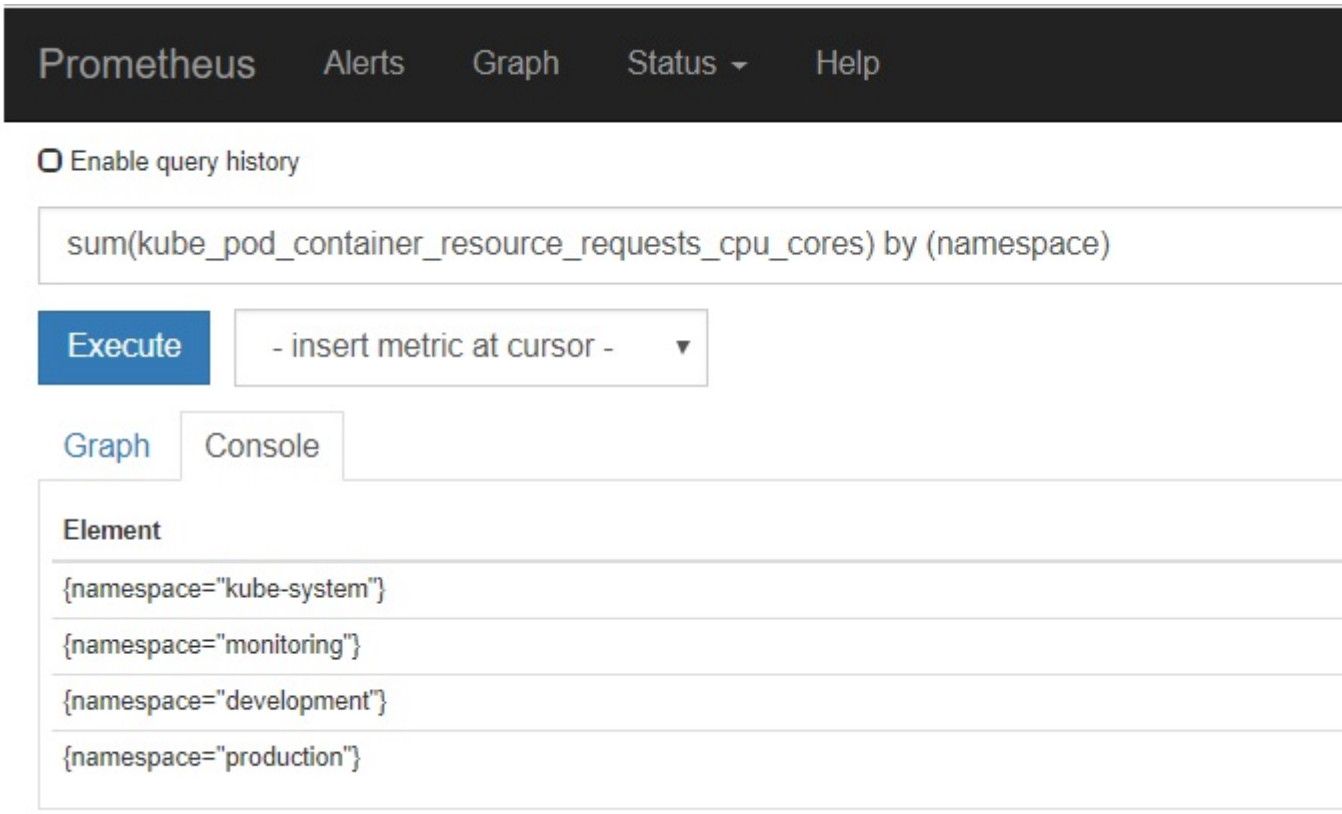
Graph Console

Element
{pod="etcd-server-events-master-us-central1-a-b665"}
{pod="etcd-server-master-us-central1-a-b665"}
{pod="redis-master-5897b5c7fd-xf47g"}
{pod="frontend-b59484f56-fmhfb"}
{pod="frontend-b59484f56-j5vdh"}
{pod="frontend-b59484f56-vbs5j"}
{pod="frontend-b59484f56-ccbck"}
{pod="frontend-b59484f56-d8l9g"}
{pod="frontend-b59484f56-m9zq6"}
{pod="alertmanager-prom-operator-prometheus-o-alertmanager-0"}
{pod="kube-apiserver-master-us-central1-a-b665"}
{pod="redis-master-5897b5c7fd-bq5vs"}
{pod="redis-master-5897b5c7fd-lkfcg"}
{pod="frontend-b59484f56-4dqmp"}
{pod="frontend-b59484f56-t78r2"}
{pod="frontend-b59484f56-f24sk"}
{pod="frontend-b59484f56-fhzsr"}
{pod="frontend-b59484f56-8p6vd"}
{pod="frontend-b59484f56-zwmgt"}
{pod="redis-slave-7fd5945648-2kbwf"}
{pod="dns-controller-6d6b7f78b-fh4d4"}
{pod="redis-master-5897b5c7fd-ktjqv"}

Resource Metric: Kubernetes Namespace CPU Requests

Expression:

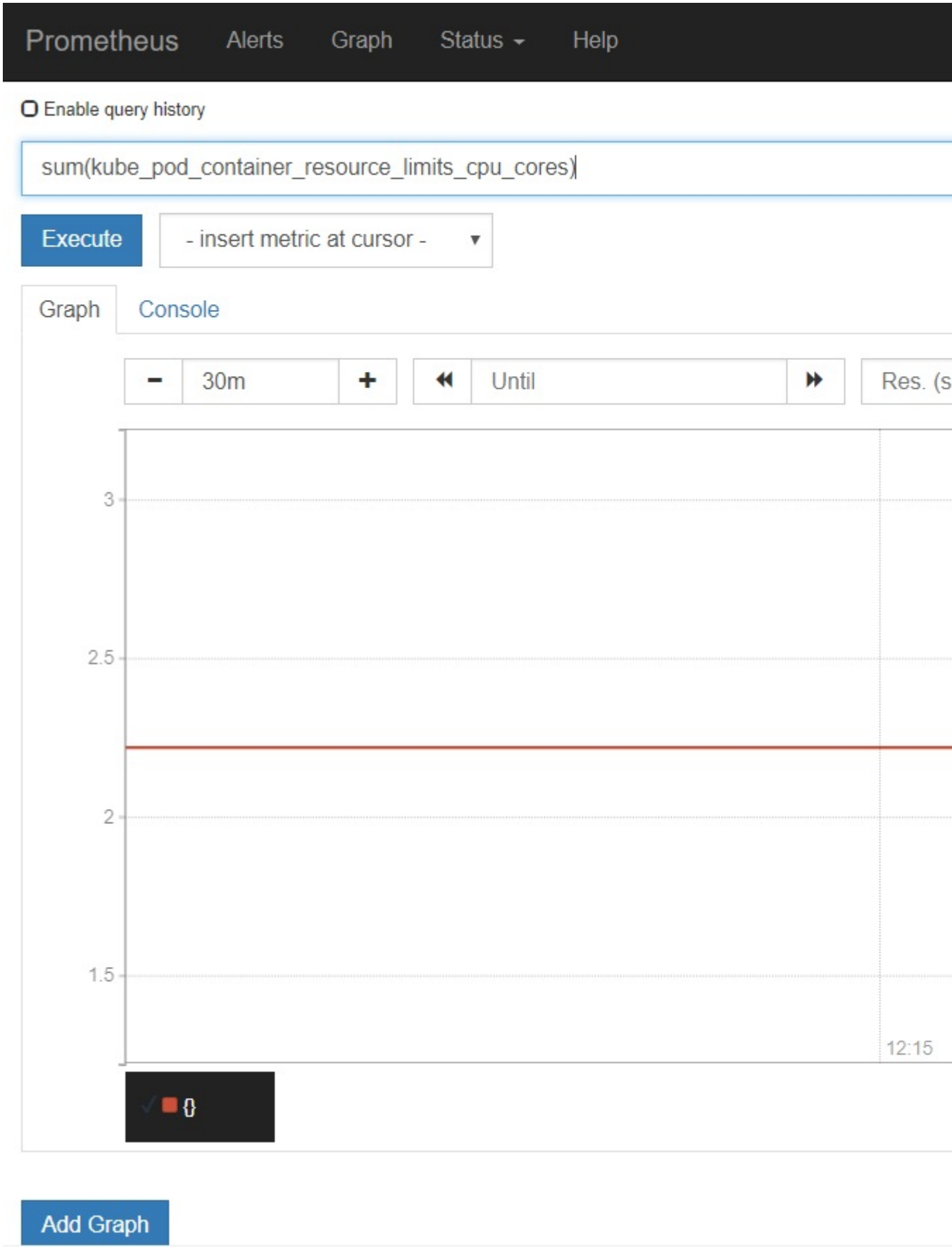
```
sum(kube_pod_container_resource_requests_cpu_cores) by (namespace)
```



Resource Metric: Kubernetes CPU Limits

Expression:

```
sum(kube_pod_container_resource_limits_cpu_cores)
```

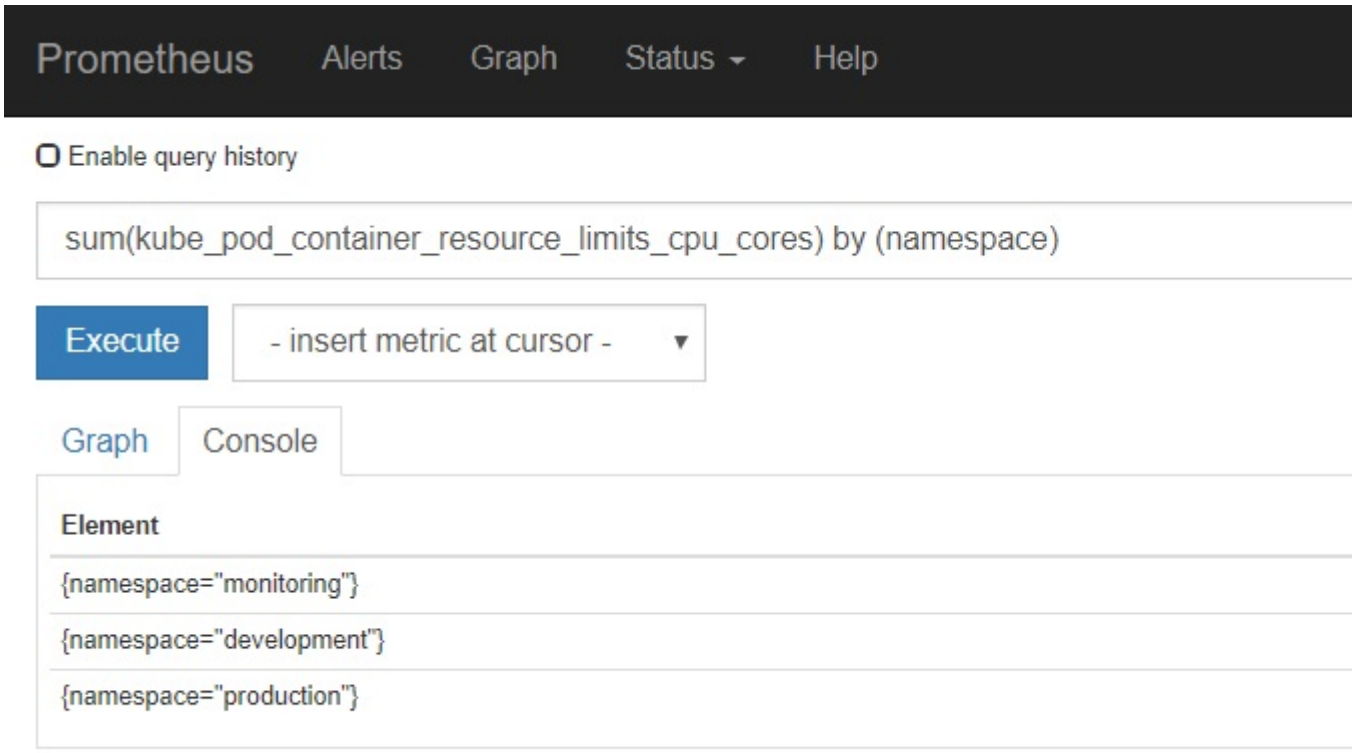

Prometheus
Alerts
Graph
Status
Help
Enable query history
sum(kube_pod_container_resource_limits_cpu_cores)
Execute
- insert metric at cursor -
Graph
Console
30m
Until
Res. (s
3
2.5
2
1.5
12:15
{}
Add Graph

The screenshot above shows us the sum of CPU limits for all containers in the cluster.

Resource metric: Kubernetes Namespace CPU Limits

Expression:

```
sum(kube_pod_container_resource_limits_cpu_cores) by (namespace)
```

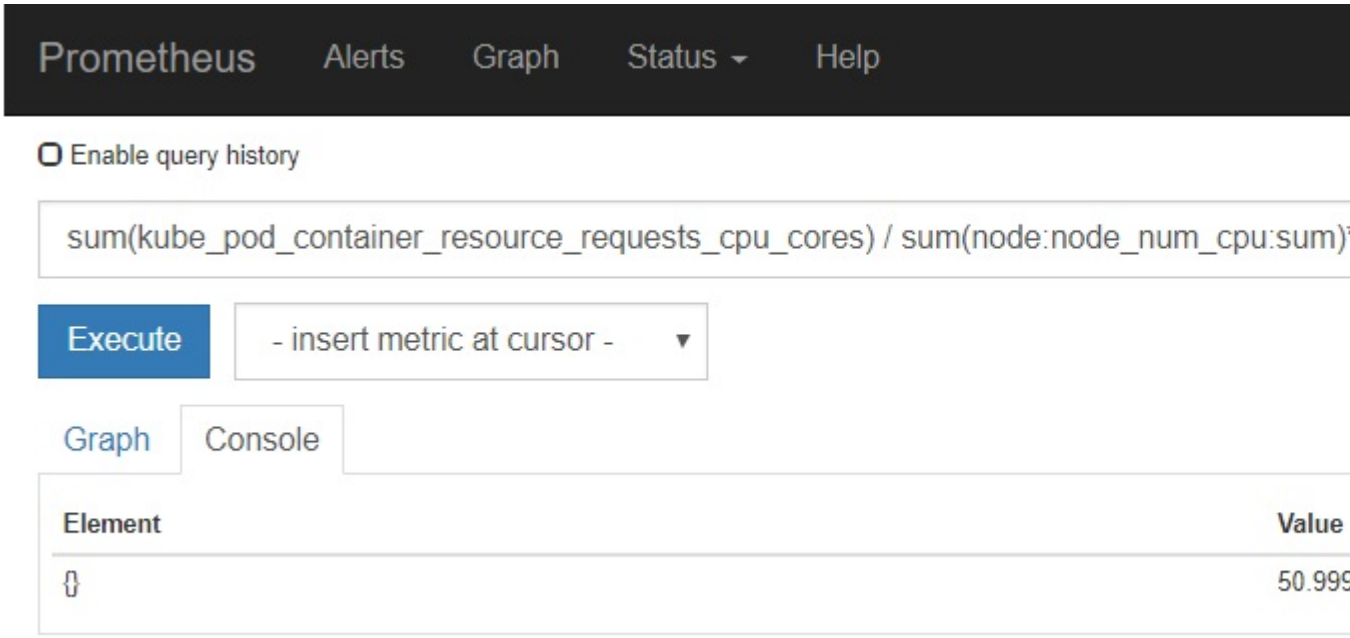


Resource Metric: Kubernetes CPU Request Commitment

Expression:

```
sum(kube_pod_container_resource_requests_cpu_cores) / sum(node:node_num_cpu:sum)*100
```

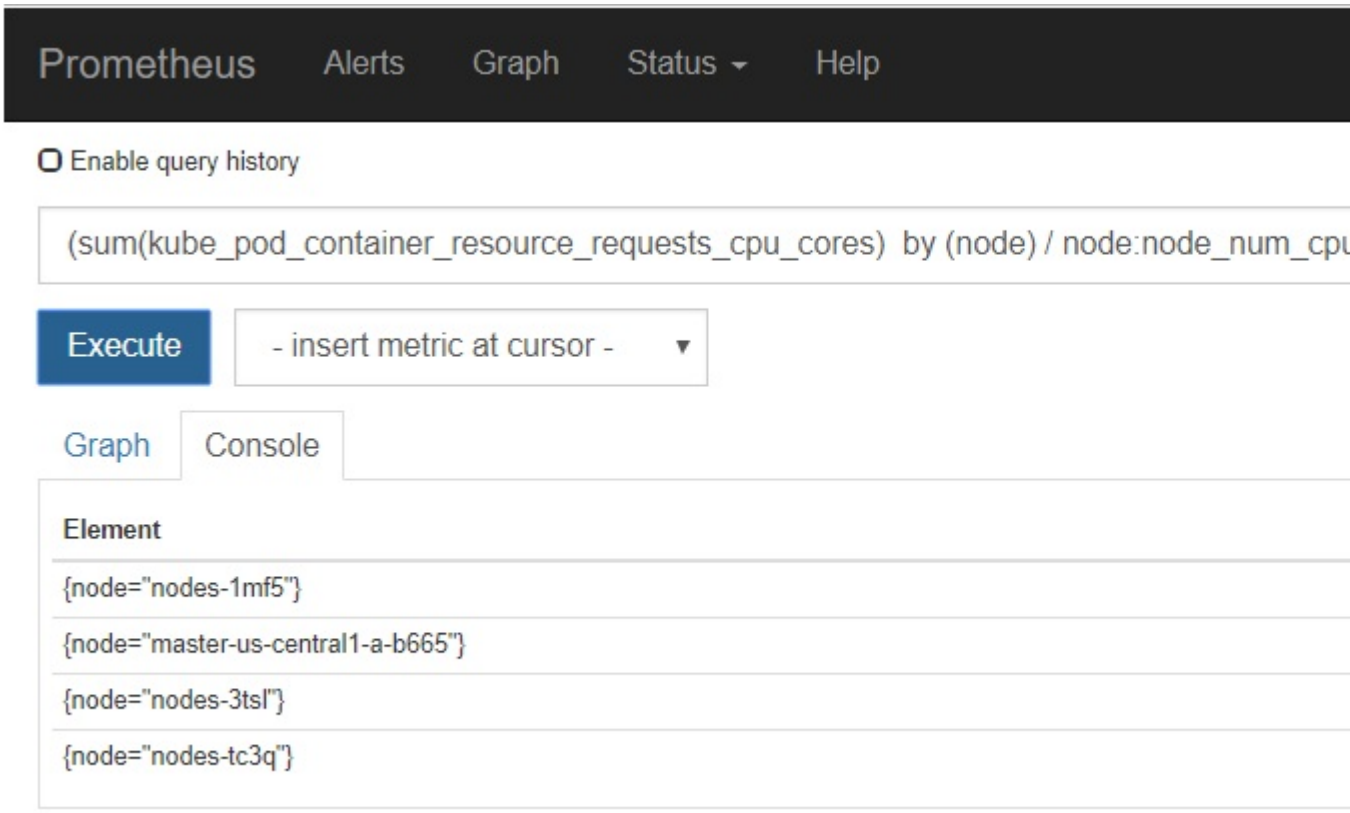
This will give us the CPU request commitment for the entire cluster.



Resource Metric: Kubernetes Node CPU Request Commitment

Expression:

```
(sum(kube_pod_container_resource_requests_cpu_cores) by (node) / node:node_num_cpu:sum)*100
```

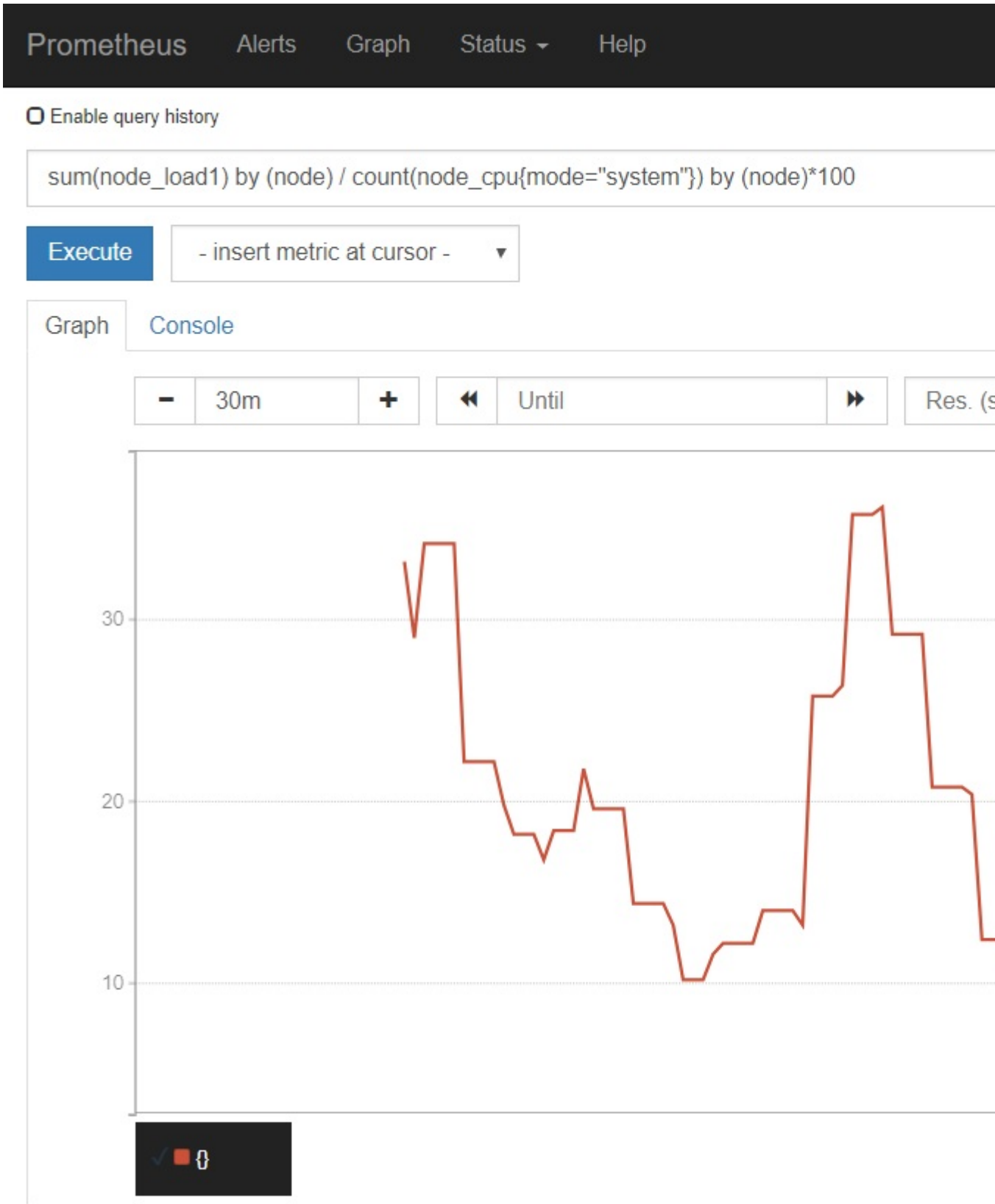


Resource Metric: Kubernetes CPU Saturation

Expression:

```
sum(node_load1) by (node) / count(node_cpu{mode="system"}) by (node)*100
```

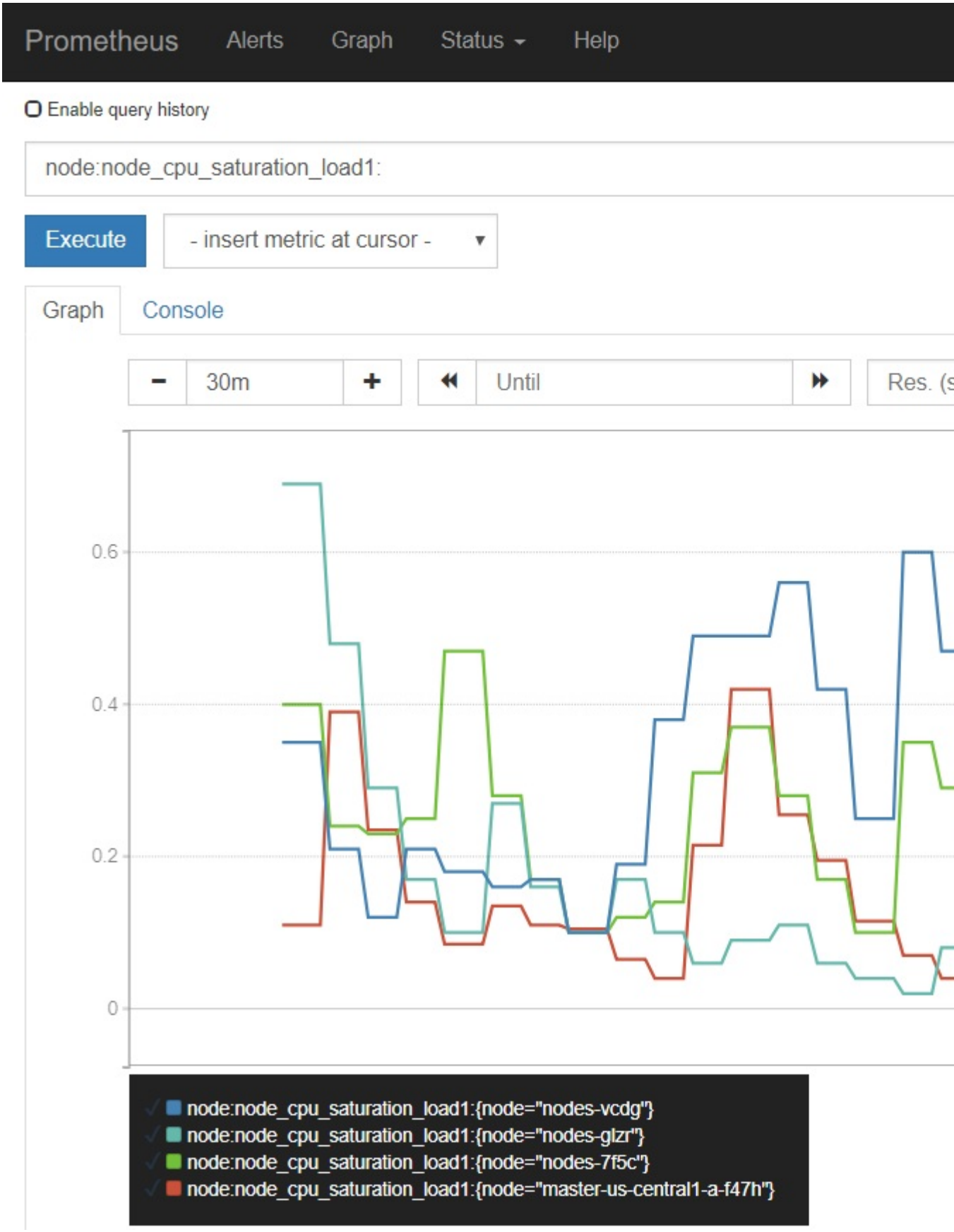
This gives us the CPU saturation for the cluster.



Resource Metric: Kubernetes Node CPU Saturation

Expression:

```
node:node_cpu_saturation_load1:
```

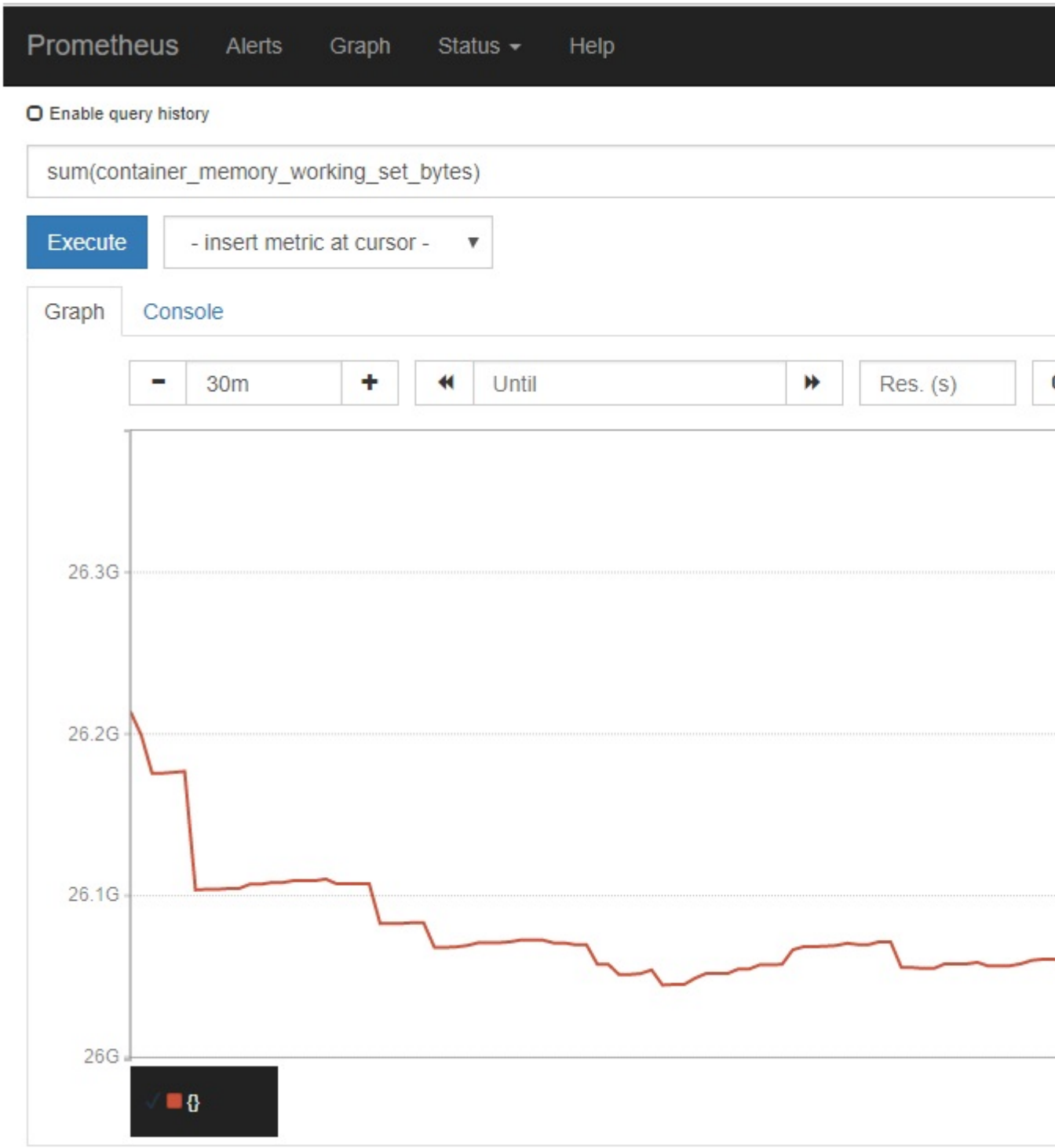

Prometheus
Alerts
Graph
Status
Help
Enable query history
node:node_cpu_saturation_load1:
Execute
- insert metric at cursor -
Graph
Console
30m
Until
Res. (s
0.6
0.4
0.2
0
node:node_cpu_saturation_load1:{node="nodes-vcdg"}
node:node_cpu_saturation_load1:{node="nodes-glzr"}
node:node_cpu_saturation_load1:{node="nodes-7f5c"}
node:node_cpu_saturation_load1:{node="master-us-central1-a-f47h"}

Resource Metric: Kubernetes Memory Usage

Expression:

```
sum(container_memory_usage_bytes{container_name!="POD",container_name!=""})
```

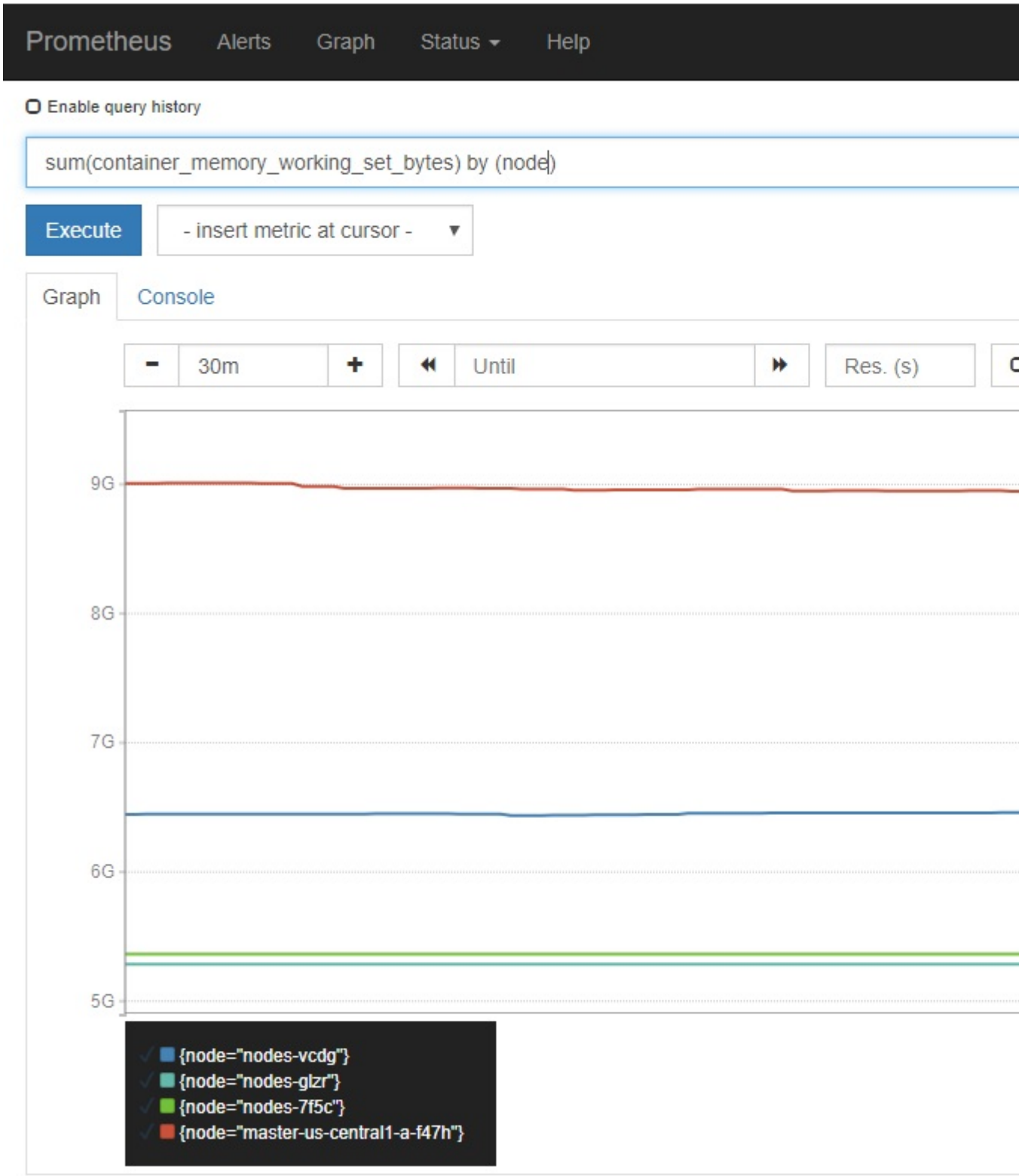
This will give us the total memory usage for all pods in the cluster



Resource Metric: Kubernetes Node Memory Usage

Expression:

```
sum(container_memory_usage_bytes{container_name!="POD",container_name!=""}) by (node)
```

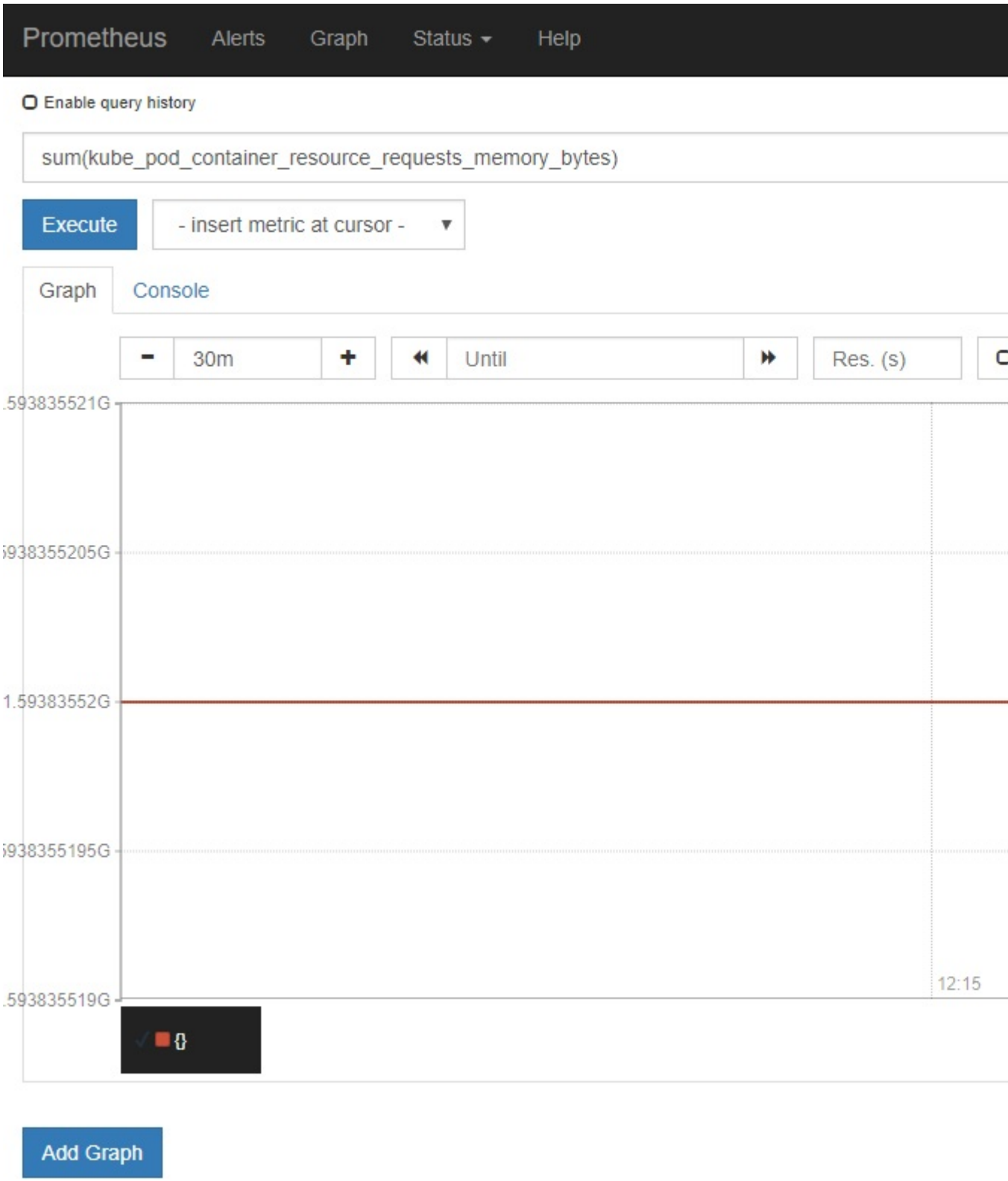


Resource Metric: Kubernetes Memory Requests

Expression:

```
sum(kube_pod_container_resource_requests_memory_bytes)
```

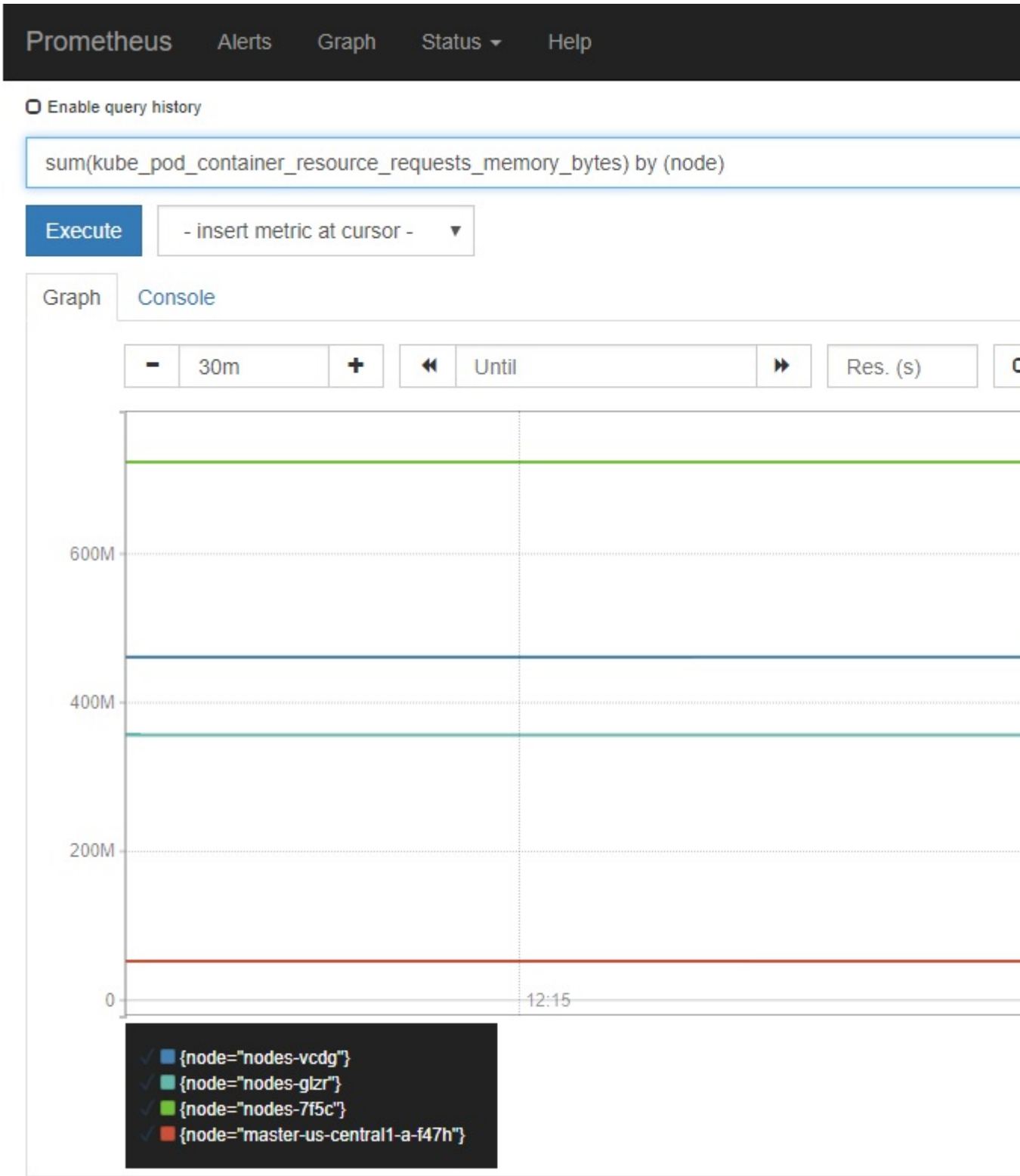
This will give us the total memory requests for all pods in the cluster



Resource Metric: Kubernetes Node Memory Requests

Expression:

```
sum(kube_pod_container_resource_requests_memory_bytes) by (node)
```

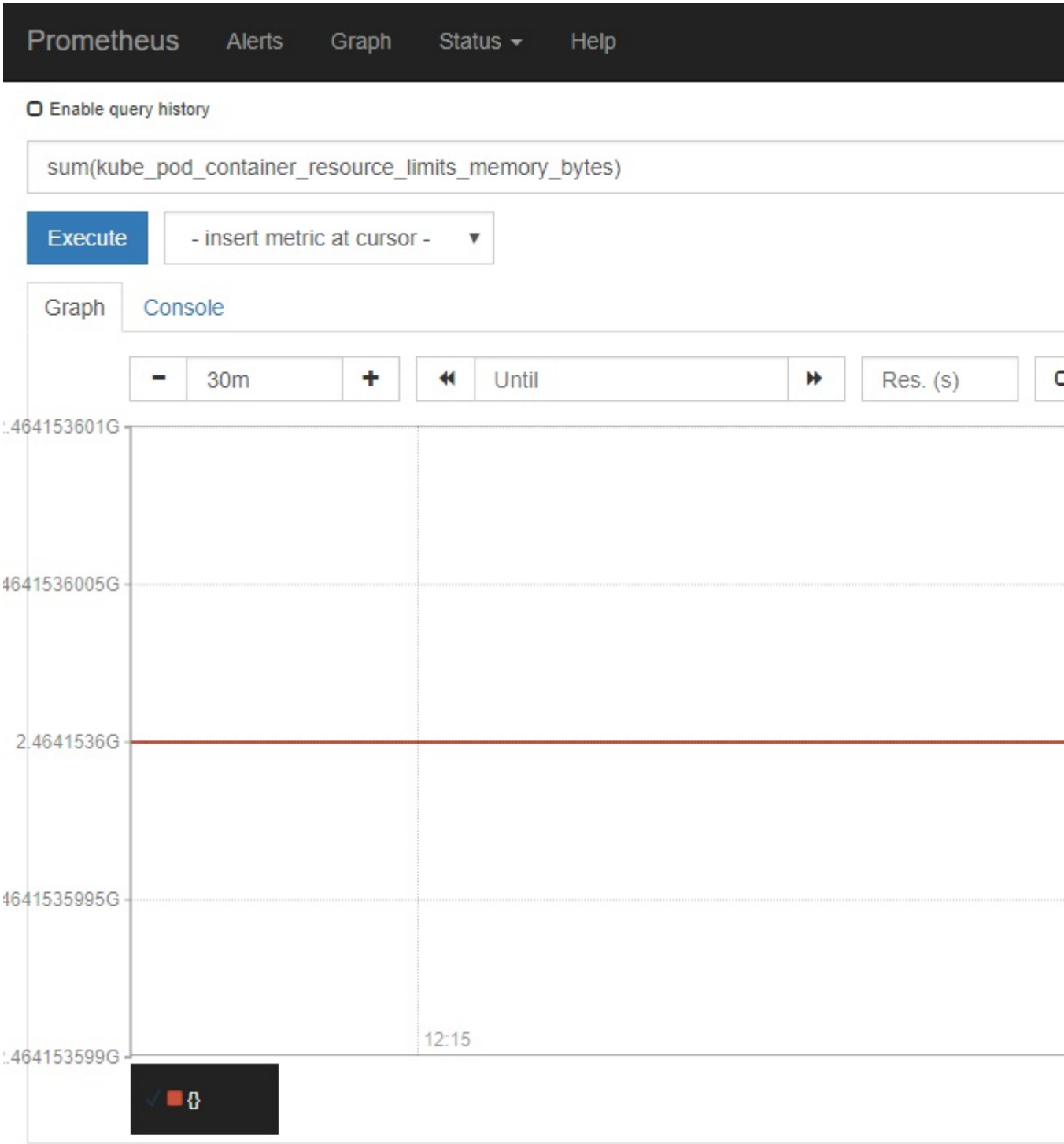


Resource Metric: Kubernetes Memory Limits

Expression:

```
sum(kube_pod_container_resource_limits_memory_bytes)
```

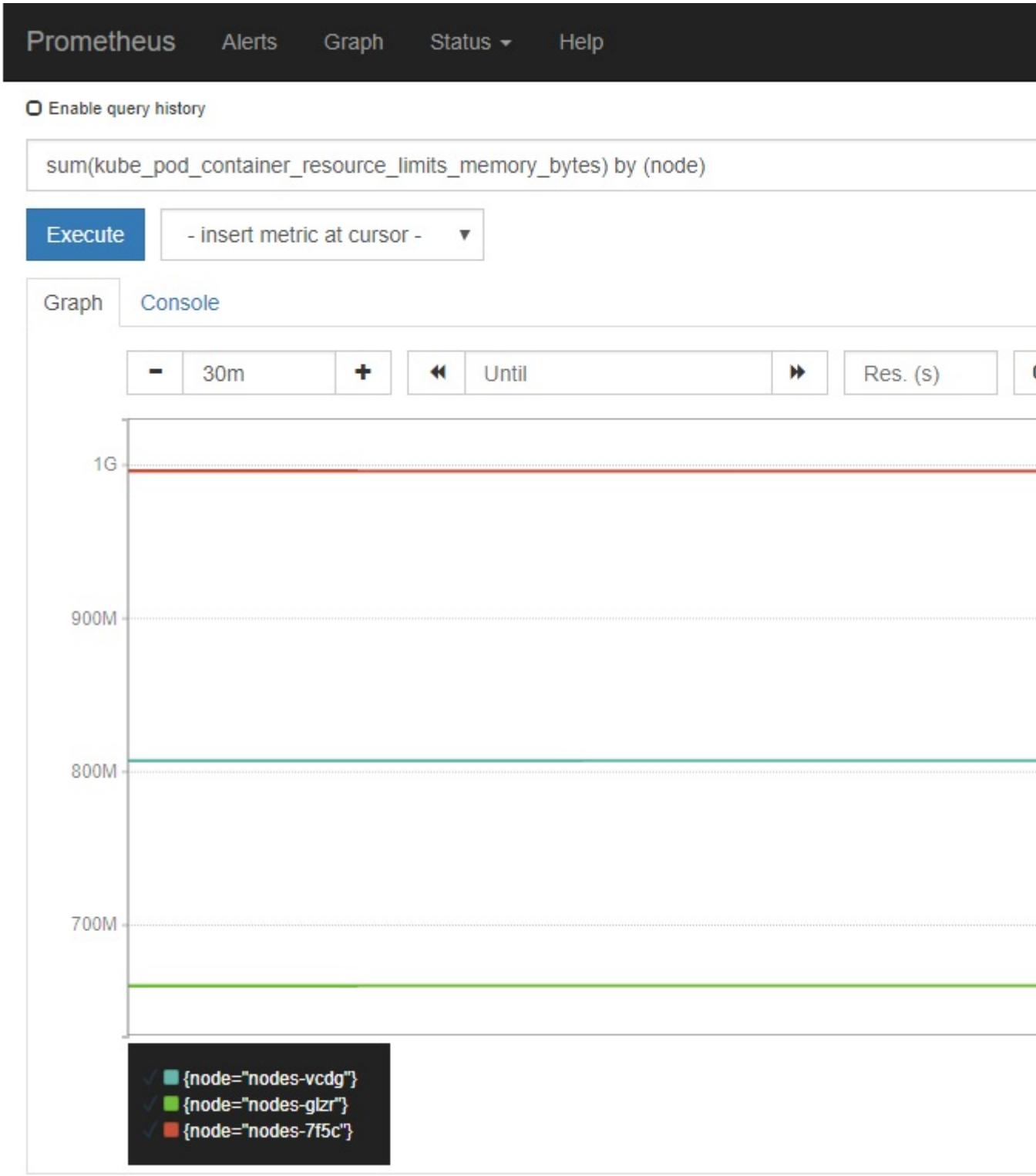
This will give us the sum of memory limits for all pods in the cluster



Resource Metric: Kubernetes Node Memory Limits

Expression:

```
sum(kube_pod_container_resource_limits_memory_bytes) by (node)
```

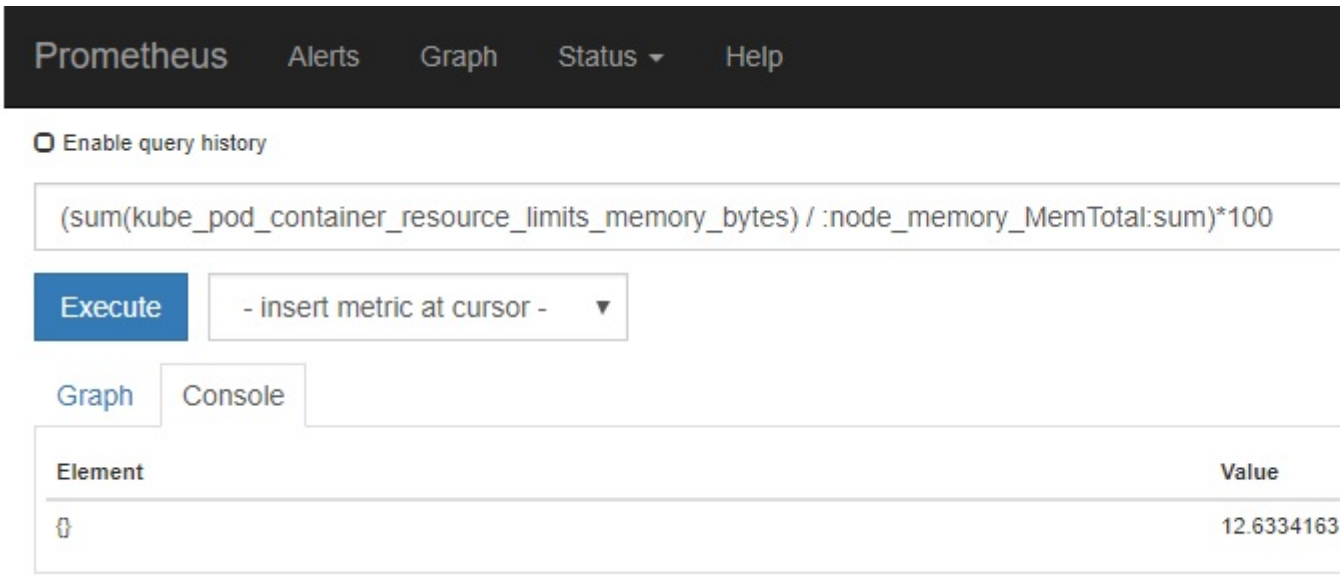


Resource Metric: Kubernetes Memory Limit Commitment

Expression:

```
(sum(kube_pod_container_resource_limits_memory_bytes)
/ :node_memory_MemTotal:sum)*100
```

This will give us the memory limit commitment for the entire cluster.



Conclusion

In this blog post we outlined an extensive list of Kubernetes resource metrics, the importance of monitoring them as well as setting up the tools required. In the next installment, we will look at further Prometheus resource metrics and also deploy Grafana for more user friendly dashboards.

The list of metrics we considered for this blog post are based on hardware and OS metrics, the distinct hardware and software abstractions Kubernetes introduces and it's resource management model. What we did not consider are organizational imperatives driven by the need to understand resource allocation, usage and utilization based on custom organizational groupings. These custom groupings can range from customers and clients to teams, applications and departments.

Replex drives visibility into metrics for these custom groupings, in addition to the ones based on native Kubernetes abstractions. It does this on two levels; Usage and cost. Replex drives cost transparency by co-relating usage with the costs of the underlying hardware Kubernetes pods are running on. This essentially means that IT managers can visualize the usage profiles as well as the associated costs of individual teams, applications or clients using the shared Kubernetes infrastructure.

Replex also collects and analyzes usage and utilization metrics in real-time and provides actionable intelligence on optimizing them. On average, this translates into cost savings of up-to 30% across clients using a wide range of infrastructure variants from public cloud to multi and hybrid cloud.

Ready for Production? Download the Complete Production Readiness Checklist with Checks, Recipes and Best Practices for Availability, Resource Management, Security, Scalability and Monitoring