Autoscaling Leases

Table of contents

> ⚠ **Attention**
>
> NVIDIA Triton Management Service (TMS) will reach the end of life on July 31, 2024. The version 1.4.0 is the last release.

TMS has the capability to automatically scale the number of Triton instances associated with a lease based on utilization. This means that as a lease becomes heavily utilized, TMS can transparently add more Triton instances to service inference requests, and as demand decreases, automatically remove unneeded instances.

This document covers what TMS administrators must do to enable this feature and how TMS users can leverage it to speed up inference. This include:

- Installing necessary third-party tools.
- Configuring TMS to enable autoscaling, as well as parameters that can be used to control it.
- Enabling autoscaling for a lease.

Installing Prerequisites

To make autoscaling work, TMS needs to be able to collect performance metrics and then make them available to Kubernetes for determining when to automatically scale leases. This requires two third-party tools to be installed in Kubernetes: Prometheus and the Prometheus Metrics Adapter. This guide will cover the basics of installation and configuration, but ultimately, the TMS administrator should follow the latest instructions for installing, configuring, and securing these tools as provided by the developers of the tools. Luckily, there are Helm charts available for both of these tools, so a basic installation in Kubernetes for testing purposes is fairly straight-forward.

Note: If your cluster is already using Prometheus and the Prometheus metrics adapter for other purposes, you do not need to install a separate copy of these for TMS. So long as the installed copy can monitor cods in the namespace where you're installing TMS, things should just work.

Installing Prometheus

For the most up-to-date instructions for installing Prometheus, see their installation guide. As noted above, for production clusters you should work with your system administrator to make sure you properly configure and secure Prometheus. At least for testing purposes, Prometheus can be easily installed in Kubernetes via a Helm chart available on Github. Note that the this Helm chart is currently in a beta state and is subject to change.

To install Prometheus via Helm, you can simply do this:

```
$ TARGET_NAMESPACE = ... # put your namespace name here
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
$ helm install -n $TARGET_NAMESPACE prometheus prometheus-community/kube-prometheus-stack
```

Verify that the installation was successful by running `kubectl get pods` and verify that the Prometheus pods are running and healthy (they may take a bit of time to start).

Installing the Prometheus Metrics Adapter

Once Prometheus is installed, you can proceed to install the Prometheus metrics adapter. Just like with installing Prometheus, in production clusters this should be done by or with the help of the system administrators to ensure any security concerns are properly addressed.

When installing via Helm, the first step is to find the name of the Prometheus service by running `kubectl get svc`. If you used the default options above, you should see a service named `prometheus-kube-prometheus-prometheus`. If you do not see a service named `prometheus-kube-prometheus-prometheus`, double-check if Prometheus was installed properly, or look to see if an update to the Prometheus Helm chart has changed the name of the service.

```
$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
prometheus-kube-prometheus-prometheus ClusterIP 10.152.183.39 <none>
9090/TCP 6h33m
```

```
prometheus-grafana ClusterIP 10.152.183.204 <none> 80/TCP 6h33m
prometheus-kube-prometheus-operator ClusterIP 10.152.183.197 <none> 443/TCP 6h33m
prometheus-prometheus-node-exporter ClusterIP 10.152.183.154 <none> 9100/TCP 6h33m
prometheus-kube-prometheus-alertmanager ClusterIP 10.152.183.155 <none> 9093/TCP 6h33m
prometheus-kube-state-metrics ClusterIP 10.152.183.117 <none> 8080/TCP 6h33m
alertmanager-operated ClusterIP None <none> 9093/TCP,9094/TCP,9094/UDP 6h33m
prometheus-operated ClusterIP None <none> 9090/TCP 6h33m
```

With the name of the Prometheus service, you can now install the Prometheus adapter.

```
$ TARGET_NAMESPACE = ... # put your namespace name here
$ helm install -n $TARGET_NAMESPACE prometheus-adapter prometheus-community/prometheus-adapter --set=prometheus.url=http://prometheus-kube-prometheus-prometheus
```

If everything installed successfully, Prometheus should start collecting metrics from the cluster within a few minutes. You can verify this by getting metrics the Kubernetes custom metrics API.

Note: if you don't have `jq` installed in your system, you can run without it – you'll just get all the output in a single line.

```
$ kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1 | jq | less
```

You should see output that look like the below. The actual entries don't matter so long as there are some entries.

```
{
"kind": "APIResourceList",
"apiVersion": "v1",
"groupVersion": "custom.metrics.k8s.io/v1beta1",
```

```
"resources": [
{
"name": "services/node_memory_KReclaimable_bytes",
"singularName": "",
"namespaced": true,
"kind": "MetricValueList",
"verbs": [
"get"
]
},
{
"name":
"services/prometheus_remote_storage_string_interner_zero_reference_releases",
"singularName": "",
"namespaced": true,
"kind": "MetricValueList",
"verbs": [
"get"
]
}
]
}
```


Server Configuration

Since it requires the installation of additional third-party components, lease autoscaling is disabled by default. To enable it, you must set the appropriate values in `values.yaml`. These options are in the `server.autoscaling` section of the file. The TMS administrator should set these values based on the hardware available in the cluster, the expected workloads for this particular installation, and the configuration options of Prometheus.

```
server:
autoscaling:
enable: false
replicas:
```

```
default:
minimum: 1
maximum: 5
limits:
maximum: 10
minimum:
lowerBound: 1
upperBound: 2
metrics:
cpuUtilization:
allowed: false
enabled: false
threshold:
default: 90
minimum: 50
maximum: 100
gpuUtilization:
allowed: false
enabled: false
threshold:
default: 90
minimum: 50
maximum: 100
queueTime:
allowed: false
enabled: false
threshold:
default: 10000
minimum: 10000
maximum: 0
prometheus:
podMonitorLabels:
release: prometheus
ruleLabels:
release: prometheus
```

The options in this section are as follows:

- `enable` (default `false`): Controls whether autoscaling is enabled. Valid values are `true` and `false`.
- `replicas` (dictionary): Controls the number of replicas that will be allowed for leases. See `values.yaml` for further details on all the options.
- `metrics` (dictionary): A set of metrics which can be used for autoscaling. This is defined in further detail later.
- `prometheus` (dictionary): Options which specify how Prometheus finds Kubernetes objects created by TMS that are used in autoscaling. These are described further detail below.

In the above, if `server.autoscaling.enable` is switched to `true`, the following would happen:

- If a user does not request autoscaling for a lease, their lease will not automatically scale.
- If a user requests autoscaling for a lease but does not specify a maximum number of replicas, the will will have at most `5` replicas (`server.autoscaling.replicas.default.maximum`).
- If a user requests autoscaling for a lease and they specify the maximum number of replicas, they can request up to `10` replicas (`server.autoscaling.replicas.limits.maximum`).

The section on requesting autoscaling leases describes how to make these requests.

Configuring Autoscaling Metrics

The dictionary `server.autoscaling.metrics` defines a series of metrics on which autoscaling may trigger. Each metrics consists of a threshold along with a boolean flag indicating whether or not the metric is configured. Based on this, each metric is used to calculate a target number of replicas. The largest number is then used. The details of how each target number is calculated can be found in the Kubernetes documentation.

The metrics are as follows:

- `cpuUtilization` (dictionary): Scale based on high CPU utilization. Values are expressed as a percentage.
- `gpuUtilization` (dictionary): Scale based on high GPU utilization. Values are expressed as a percentage.
- `queueTime` (dictionary): Scale based on inference requests spending a long amount of time in the queue before they are executed. Values are expressed in microseconds.

Each metric has the following entries:

- `enable` (default `false`): Whether to enable this metric by default.
- `allowed` (default `false`): Whether this metric can be enabled on a per-lease bases.
- `threshold` (dictionary): Values that determine when a lease should be scaled up and down.
- `threshold.default` (integer): the default value if not specified on a per-lease basis.
- `threshold.minimum` (integer): the minimum value allowed when specified on a per-lease basis.
- `threshold.maximum` (integer): the maximum value allowed when specified on a per-lease basis.

Configuring Prometheus Objects

When autoscaling is enabled, TMS will create a number of Kubernetes objects related to Prometheus. For autoscaling to work properly, Prometheus must be able to detect these objects. This is configured via the `server.autoscaling.prometheus` entry in `values.yaml`. This object has the following entries:

- `podMonitorLabels` (dictionary): A set of labels which will be added to `PodMonitor` objects so that Prometheus can monitor the metrics of the Triton pods. This must match the value of `.spec.podMonitorSelector` in your Prometheus configuration.

- `ruleLables` (dictionary): A set of labels which will be added to `PrometheusRule` objects so that Prometheus can detect rules used by TMS to define new metrics. This must match the value of `.spec.ruleSelector` in your Prometheus configuration.

In addition to the above, if your Prometheus installation has specified values for `.spec.podMonitorNamespaceSelector` or `.spec.ruleNamespaceSelector`, you need to ensure that the namespace into which you install TMS has matching labels applied to it.

Verifying Autoscaling Leases Are Working Properly

Requesting Autoscaling Leases

On a server that is properly configured, users may request a lease support autoscaling via the programmatic gRPC API, as well as the tmsctl command-line tool. The documentation for the API and tools contains complete details of the different flags and their usage. This section only gives an overview of how to request autoscaling via `tmsctl`.

To request autoscaling with the default parameters, users just need to add the `--enable-autoscaling` flag (below `$MODEL_OPTIONS` is a stand-in for whatever model the user wants to load:

```
$ tmscl lease create -m $MODEL_OPTIONS --enable-autoscaling
```

To specify the maximum number of replicas, simply use the `--autoscaling-max-replicas` option. For example, the below requests a maximum of four replicas.

```
$ tmscl lease create -m $MODEL_OPTIONS --enable-autoscaling --autoscaling-max-replicas 4
```

In both cases above, the leases will start with a single replica of Triton, and as inference requests increase, the number of Triton instance for the lease will increase until they reach their respective maximums.

Troubleshooting

Prometheus has many rules that determine how it searches for different Kubernetes objects in order to adjust its behavior dynamically. If these don't match how you

configured TMS, autoscaling will not work properly. The Prometheus documentation provides detailed information on all the option. This section covers some of the more common issues.

Symptom: You are not seeing any metrics collected for your Triton pods.

Things to Check:

- Make sure that you set `.server.autoscaling.prometheus.podMonitorLabels` in `values.yaml` to match the labels defined by `.spec.podMonitorSelector` in your Prometheus installation.
- If your Prometheus installation has set `.spec.podMonitorNamespaceSelector`, make sure that your namespace has matching labels (e.g. run `kubectl label ns tms_namespace someLabel=someValue`).

Symptom: The metric for autoscaling based on queue time (`tms_avg_request_queue_duration`) is not being collected.

Things to Check:

- Make sure that you set `.server.autoscaling.prometheus.ruleLabels` in `values.yaml` to match the labels defined by `.spec.ruleSelector` in your Prometheus installation.
- If your Prometheus installation has set `.spec.ruleNamespaceSelector`, make sure that your namespace has matching labels (e.g. run `kubectl label ns tms_namespace someLabel=someValue`).