Triton Management Service Deployment Guide (Latest)

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.

Triton Management Service (TMS) is an application that helps users manage and orchestrate a fleet of Triton Inference Servers in a Kubernetes cluster. Key features of TMS include:

- Easily creating and deleting Triton instances on-demand.
- Securely loading models from remote storage locations.
- Autoscaling Triton instances to meet dynamic workloads while minimizing resource utilization during times of low demand.
- Loading models into pooled Triton instances, allowing you to use less resources while maintaining quality-of-service metrics.

One of the main organizational units in TMS is the concept of a lease. A lease is a description of a model (or ensemble of models), along with a description of the hardware needed to run them (e.g. number of GPUs, amount of memory). To learn more about leases, see the description of leases.

To get started with TMS, first see the deployment guide to learn how to install and configure TMS. Once you have TMS running, see the basic operations tutorial to learn how to create Triton instances and load models into them. If you don't already have a Kubernetes cluster but want to try out TMS, see the minikube quickstart guide for an example of how to set up a test environment and install TMS in it.

Contents

TMS Basics Tutorial

⚠ **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.

This guide will walk you through the basics of creating a lease, running inference against it, and releasing the lease. This guide assumes the following:

- You are familiar with the basics of leases.
- You already have a TMS cluster up and running, with the appropriate secrets configured to get containers from NGC. If you do not, please see the deployment guide to learn how to configure and install TMS.
- You have `tmsctl`, the TMS CLI tool, already installed. This can be downloaded from NGC.
- You have a model repository configured that is hosting your models.
- You can run `kubectl` commands to communicate with your cluster. This is needed to run `kubectl port-forward` commands to open up ports to the Triton server which will host your models. Another option is to run the tutorial in a pod inside the same cluster running Kubernetes. In this case, you can skip the `kubectl port-forward` commands (you will also need to make some slight modifications to refer to the correct service rather than `localhost`).

Connecting to TMS

Before getting to the more interesting steps, you need to ensure you can communicate with the TMS server. This tutorials assumes you are running the steps outlined here outside the Kubernetes cluster hosting TMS and need to open a port to connect to it. The way to do this is to use the `kubectl port-forward`.

First, you need to locate the TMS service. By default, it should be named `tms` and be running on port 30345. The rest of the tutorials assumes this is the case for your installation. If not, you'll have to modify some commands. To check where TMS is running, run `kubectl get svc`:

```
% kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
tms ClusterIP 10.98.225.223 <none> 30345/TCP 7s
```

With the name of the service and port number on which it is listening, you can run a `kubectl port-forward` command so you can communicate with the service from your local machine. You need to leave this command running, so you will need to do it in separate terminal form the one one which you will be running the rest of the commands.

```
% kubectl port-forward svc/tms 30345:30345
Forwarding from 127.0.0.1:30345 -> 9345
Forwarding from [::1]:30345 -> 9345
```

You should now be able to communicate with TMS. To test it, run a `tmsctl lease list` command:

```
% tmsctl lease list -t http://localhost:30345
Lease State Expires Triton
Count: 0
```

Notice that above, you had to specify the address of the TMS via the `-t` flag. To avoid having to do that each time, run the `tmsctl target add` command to set a default target.

```
% tmsctl target add --set-default test-target http://localhost:30345
```

To inspect your set of named targets and see the default, you can run `tmsctl target list`

```
% tmsctl target list
* test-target -> http://localhost:30345
```

Now you can run `tmsctl` without specifying the `-t` option.

```
% tmsctl lease list
Lease State Expires Triton
Count: 0
```

The rest of this tutorial will assume you have set this and will not specify the `-t` option on each command.

Creating Your First Lease

Creating a lease requires two things:

1. The URI from which to fetch the model.
2. The name of the model which will be used by Triton.

Your TMS installation should be configured model repositories, such as one hosted in an S3 bucket, or in a Kubernetes persistent volume. If this has not already been done, please refer to the model repository documentation. You can also pull a model from an HTTP server.

The below assumes that you have set `$MODEL_URI` to the URI from which to fetch the model. For example, if the model is in a repository named `my_repo`, and is in a folder named `my_model`, you would set `MODEL_URI=model://my_repo/my_model`. If instead your model is hosted on an HTTP server, you would set `MODEL_URI=https://www.example.com/my_model.zip`.

In addition to `$MODEL_URI`, the below assumes you have set `$MODEL_NAME` to the name that your model should have in Triton. This will be used as part of the path for inference requests.

To create a lease, run the `tmsctl lease create` command. Specify a duration of at least 30 minutes (e.g. `--duration 30m`, `--duration 1h`) to have enough time to run the tutorial, but not so much that if you forget to release the lease you hold the resources for too long.

```
% tmsctl lease create -t $TMS_ADDRESS -m name=$MODEL_NAME,uri=$MODEL_URI
--duration 30m
Lease 9fd209b1f45f424c914ebc2967a3b591
State: Valid
Expires: 2023-10-12T23:27:19Z
Triton: triton-de245ce9.yournamespace.svc.cluster.local
<nvcr.io/nvidia/tritonserver:23.09-py3>
Models:
Name Url Status
$MODEL_NAME $MODEL_URI Ready
```

Assuming everything went well, you should see output similar to the above. A few things to note of importance:

- The lease ID is listed first. This is how you will refer to the lease for operations like getting its status, renewing it, or releasing it. In the example above, this is `9fd209b1f45f424c914ebc2967a3b591`. The rest of this tutorial will refer to this as `$LEASE_ID`.

- The line starting with `Triton:` gives the URL of the Triton server hosting your lease (with all it models). Above, it is `triton-de245ce9.yournamespace.svc.cluster.local`. The first component of this (`triton-de245ce9`), will be referred to as `$TRITON_SERVER` in the rest of the tutorial.

Running Inference

With your lease ready, you can run inference against any of its models (just one in this example). The details of the parameters will vary widely depending on your particular model. The below shows the overall idea, but you will have to adjust it for your model. In a common deployment scenario, you would likely have an application that is making inference requests rather than doing it manually. This is meant to simply demonstrate how to go from creating a lease to running inference.

An important thing to note is that the Kubernetes services associated with the leases are only available inside the cluster. To reach it externally, you need to run a `kubectl port-forward` command like you did for TMS.

```
% kubectl port-forward svc/$TRITON_SERVER 8000:8000
```

You can now run inference against the server. Again, the particulars of the parameters to your model will vary.

```
% curl -X POST -H "Content-Type: application/json"
http://localhost:8000/v2/models/$MODEL_NAME/infer \
--data '{ "inputs": [ {"name": "INPUT", "shape": [1], "datatype": "FP32", "data": [10] }]}'
```

You should see output like the below:

```
{"model_name":"$MODEL_NAME","model_version":"1","outputs":
[{"name":"OUTPUT","datatype":"FP32","shape":[1],"data":[10.0]}]}
```

Other Lease Operations

Now that you have a lease, you can perform many different operations on it. Below are a few basic ones.

List Leases

You can always run `tmsctl lease list` to see the state of leases in your TMS installation.

```
% tmsctl lease list
Lease State Expires Triton
9fd209b1f45f424c914ebc2967a3b591 Valid 2023-10-12T23:39:28 triton-
de245ce9.epauli.svc.cluster.local
Count: 1
```

Lease Status

To get detailed information about a lease, run `tmsctl lease status`.

```
% tmsctl lease status $LEASE_ID
Lease 9fd209b1f45f424c914ebc2967a3b591
State: Valid
Expires: 2023-10-12T23:39:28Z
Triton: triton-de245ce9.yournamespace.svc.cluster.local
<nvcr.io/nvidia/tritonserver:23.09-py3>
Readied: 2023-10-12T23:17:19Z
Models:
Name Url Status
$MODEL_NAME $MODEL_URI Ready
Events:
Type Source Age Message
Status Triton Manager 0s Creating Triton deployment.
Status Triton Manager 4s Triton deployment ready.
Status Triton Sidecar 5s identity cached; model size: 1930.
Status Triton Sidecar 7s identity is ready.
Status Lease Provider 9s Lease ready.
Status Lease Service 8m Lease renewed by request.
Status Lease Service 12m Lease renewed by request.
```

Renew

Your TMS installation will be configured with a default duration for leases. After that time elapses, TMS will automatically release the lease. If you still need it, you can run `tmsctl lease renew` to renew the lease.

```
% tmsctl lease renew $LEASE_ID
Renewed lease 9fd209b1f45f424c914ebc2967a3b591 [Valid]
Expires: 2023-10-12T23:34:44
```

Create a Custom Lease Name

In the section above where you ran inference, you had to use the name of the Triton instance in the URL. You can create additional names for a lease, which you can use to run inference. You can use this feature to provide more meaningful names to your Triton instances, as well as move the name from one lease to another so you can update your models without changing the URL your application uses.

To be able to use the name `myname` to refer to your lease, run the below.

```
% tmsctl lease name create myname $LEASE_ID
Lease name "myname".
Target lease: $LEASE_ID
```

You can now run inference using the `myname` hostname. You can still use the name of the Triton server as well.

To test the new name, kill your previous `kubectl port-forward` command, and run a new one. This time, use `myname` instead of the name previously provided by TMS.

```
% kubectl port-forward svc/myname 8000:8000
Forwarding from 127.0.0.1:8000 -> 8000
Forwarding from [::1]:8000 -> 8000
```

```
% curl -X POST -H "Content-Type: application/json"
http://localhost:8000/v2/models/$MODEL_NAME/infer \
--data '{ "inputs": [ {"name": "INPUT", "shape": [1], "datatype": "FP32", "data": [10] }]}'
```

Releasing the Lease

When you are done using a lease, you can release all resources associated with it by running `tmsctl lease release`.

```
% tmsctl lease release $LEASE_ID
Lease $LEASE_ID
State: Released
```

TMS Minikube Quickstart Guide

⚠ **Attention**

Let's give readers a helpful hint!

In this quickstart guide, we'll set up a single-node Kubernetes cluster with minikube and install TMS onto it for development and testing. We'll also create an NFS model repository on our host machine to load our models from.

The quickstart guide was written for an Ubuntu Linux machine with the bash shell – you may need to make some modifications depending on your dev environment.

Prerequisites

- Docker
- NGC CLI
- Root access to your server
- (Optional) A CUDA capable GPU and NVIDIA GPU Drivers, if deploying GPU models

Create minikube cluster

In order to deploy TMS, we need to have a Kubernetes cluster available to us. TMS works with a wide variety of Kubernetes flavors – in this guide we'll be using minikube, which makes it easy to deploy a single-node cluster for development and testing. If you already have a Kubernetes cluster available to you, you might not need to go through these steps.

Note

Note: Installation instructions for third party components are included in this guide for convenience, but we recommend looking at each tool's linked documentation for the most up-to-date information.

1. Install minikube

```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube_latest_amd6
sudo dpkg -i minikube_latest_amd64.deb
```

2. Install kubectl

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl
sudo curl -fsSLo /etc/apt/keyrings/Kubernetes-archive-keyring.gpg
https://dl.k8s.io/apt/doc/apt-key.gpg
echo "deb [signed-by=/etc/apt/keyrings/Kubernetes-archive-keyring.gpg]
https://apt.Kubernetes.io/ Kubernetes-xenial main" | sudo tee
/etc/apt/sources.list.d/Kubernetes.list
sudo apt-get update
sudo apt-get install -y kubectl
```

3. Install Helm

```
curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

4. Start minikube

```
minikube start
```

Create Model Repository

Our next step is to create and fill our model repository to hold the model artifacts that TMS will deploy. Several different kinds of model repositories are available – see the model repository documentation for details.

In this guide, we'll cover two different kinds of model repositories – NFS and HTTP. You'll only need to create one model repository to use TMS. In either case, we'll need to download the models onto our dev machine.

Note

In this example, we're using the image recognition model from the Triton quickstart guide.

```
mkdir -p model_repository/densenet_onnx/1
curl
https://contentmamluswest001.blob.core.windows.net/content/14b2744cf8d6418c87f
1.2.onnx \
-o model_repository/densenet_onnx/1/model.onnx
curl https://raw.githubusercontent.com/triton-inference-
server/server/main/docs/examples/model_repository/densenet_onnx/config.pbtxt \
-o model_repository/densenet_onnx/config.pbtxt
curl https://raw.githubusercontent.com/triton-inference-
server/server/main/docs/examples/model_repository/densenet_onnx/densenet_label
\
-o model_repository/densenet_onnx/densenet_labels.txt
```

You should now have the following directory structure at `./model_repository`:

```
model_repository/
    densenet_onnx
    1
       model.onnx
    config.pbtxt
    densenet_labels.txt
```

HTTP Model Repository

To serve a model repository over HTTP, all we need to do is create a zip file for each model and run an HTTP server.

1. Zip model

   ```
   cd model_repository
   zip -r densenet_onnx densenet_onnx
   ```

2. Serve over HTTP. In a separate terminal, run the following

   Note

   You can use any HTTP File server. Python's `http.server` defaults to port 8000.

   ```
   python -m http.server --directory .
   ```

NFS Model Repository

NFS Model repositories for TMS have the same structure as Triton model repositories. You can check out the Triton documentation to learn more.

1. Move model repository to directory for sharing

```
sudo cp -r model_repository /srv/model_repository
```

2. Enable NFS

```
sudo apt install nfs-kernel-server
sudo systemctl start nfs-kernel-server.service
```

3. Export NFS Share

    1. Add the following line to the file `/etc/exports`:

    ```
    /srv/model_repository *(rw,sync,no_subtree_check)
    ```

    2. Then, execute the following command

    ```
    sudo exportfs -arv
    ```

4. Create Kubernetes storage resources. We'll need a `PersistentVolume` to expose our NFS share to the cluster, and a `PersistentVolumeClaim` to allow Triton pods to mount it.

```
my_nfs_server = <nfs server IP address>
```

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: PersistentVolume
metadata:
name: repo0
spec:
capacity:
storage: 2Gi
accessModes:
- ReadWriteMany
nfs:
```

```
server: $my_nfs_server
path: "/srv/model_repository"
mountOptions:
- nfsvers=4.2
EOF
```

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: repo0-claim
spec:
accessModes:
- ReadWriteMany
storageClassName: ""
resources:
requests:
storage: 1Gi
volumeName: repo0
EOF
```

Deploy TMS

TMS is deployed with a Helm chart. To deploy it, we’ll need to install Helm, configure access to the chart on NGC, and modify the deployment values.

You’ll need your NGC API Key, which can be found here.

```
export NGC_CLI_API_KEY=<your key>
```

1. Add TMS Helm repository

```
helm repo add tms-helm https://helm.ngc.nvidia.com/nvaie --
username=\$oauthtoken --password=$NGC_CLI_API_KEY
```

2. Add container pull secret

   ```
   kubectl delete secret ngc-container-pull
   kubectl create secret docker-registry ngc-container-pull \
   --docker-server=nvcr.io --docker-username='$oauthtoken' --docker-password=$NGC_CLI_API_KEY
   ```

3. Install TMS. We'll need to use different deployment parameters depending on the model repository type.

   1. HTTP Model Repository

      ```
      helm install tms tms-helm/triton-management-service \
      --set images.secrets={"ngc-container-pull"} \
      --set server.apiService.type="NodePort"
      ```

   2. NFS Model Repository

      ```
      helm install tms tms-helm/triton-management-service \
      --set images.secrets={"ngc-container-pull"} \
      --set server.apiService.type="NodePort" \
      --set "server.modelRepositories.volumes[0].repositoryName=modelrepo" \
      --set "server.modelRepositories.volumes[0].volumeClaimName=repo0-claim"
      ```

Create Your First Lease

To deploy a model, we need to create a `lease` with TMS. This lease will include a unique identifier for the model(s) you want to deploy, along with some associated metadata. See the `tmsctl` for all of the available lease options.

1. Download `tmsctl`. Check the NGC Console to ensure you're getting the latest version. `tmsctl` is the command line tool for managing TMS.

```
ngc registry resource download-version "nvaie/triton-management-service-control:v1.4.0"
unzip triton-management-service-control_v1.4.0/tmsctl.zip
```

2. Set `tmsctl` target. By doing this, you won't need to specify the TMS URL in future commands.

```
tms_url=`minikube service tms --url`
./tmsctl target add --force --set-default tms $tms_url
```

3. Make lease creation request. With this, TMS will download the model you specify and create a Triton deployment that serves this model. Depending on the kind of model repository you used, the model URI might be different.

    1. HTTP Model Repository

    ```
    ./tmsctl lease create -m
    "name=densenet_onnx,uri=http://host.minikube.internal:8000/densenet_or
    --triton-resources gpu=0
    ```

    2. NFS Model Repository

    ```
    ./tmsctl lease create -m
    "name=densenet_onnx,uri=model://modelrepo/densenet_onnx" --triton-resources gpu=0
    ```

Note

Depending on your network speed, this command may time out due to minikube setting a low threshold for the time it allows for pulling images. If that occurs, you can pull the images manually with `minikube ssh docker image pull` for the respective `tritonserver` and `triton-management-sidecar` images.

4. Add lease name

   By default, TMS assigns a random URL to the created Triton server. To make it easier to address the models from other applications, we can choose a specific name to attach to the lease and use that as part of the URL.

   Let's first get the Lease ID of the lease we just created:

   ```
   lease_id=`./tmsctl lease list -z | grep -oP 'lease:\K[^\s]+'`
   ```

   Then use that to add a new name to it:

   ```
   ./tmsctl lease name create test-lease $lease_id
   ```

   Now we can use the url `test-lease.default.svc.cluster.local:8001` within the cluster to address this lease. Note that if you installed TMS into a namespace other than `default`, you should replace that part of the URL with the namespace you are using.

Make a Triton Request

To simplify the networking, we'll be making the Triton request from within the Kubernetes cluster. So

```
kubectl run -it --rm triton-client --image=nvcr.io/nvidia/tritonserver:23.03-py3-sdk
```

```
/workspace/install/bin/image_client -m densenet_onnx -c 3 -s INCEPTION -i grpc
/workspace/images/mug.jpg -u test-lease.default.svc.cluster.local:8001
```

Congrats! You've successfully deployed and served a model with TMS!

Clean-up

To delete the Triton deployments created by TMS, you can use the `lease list` command to find all existing leases, and the `lease release` command to delete them.

```
tmsctl lease release <lease-id>
```

To remove TMS you can uninstall with Helm

```
helm uninstall tms
```

To tear down your minikube cluster, you can use

```
minikube stop
```

Triton Management Service Deployment Guide

⚠ **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.

Triton Management Service (TMS) is a Kubernetes microservice, and expects to be deployed into a Kubernetes managed cluster. To more easily facilitate its deployment into your Kubernetes cluster, TMS provides a Helm chart designed to simplify the deployment, or installation, process.

In order to deploy TMS the `helm` tool (download) and the TMS Helm chart (download must be installed on the local system. Additionally, the local user will require cluster administrator privileges.

TMS Pre-deployment Configuration

Preparing Your Cluster

In order to run TMS, you will need a properly-configured Kubernetes cluster. Depending on which TMS features you wish to leverage and whether you plan to run inference on GPUs, you will need to install some additional dependencies over a default installation.

As a baseline, production TMS installations are recommended to have at least two nodes – one on which to run the API server and database, and one on which to run inference. Typical deployments will have many nodes on which to run inference. One important note about the inference nodes is that they need to be able to run large container images. The default images for Triton can exceed over fourteen gigabytes, so make sure your cluster is properly configured to handle that (also, be prepared for Triton to take a bit of time the first time it starts on each node, as it can take some time for the image to transfer).

If you will be running inference on GPUs, you need to ensure that your inference nodes properly recognize the GPUs and list them as resources. You can check whether this is the case by running `kubectl describe node $NODE_NAME` and seeing whether there is an entry with a key of `nvidia.com/gpu` in the `Capacity` and `Allocatable` sections. If your cluster is not already properly configured, please see the documentation for the GPU operator or your cloud service provider.

If your deployment requires the autoscaling feature, please see the autoscaling section below.

For the specifics about the versions of Kubernetes and other tools with which TMS was tested, please see the release notes for the version of TMS your are deploying.

Obtaining TMS Helm Chart

The TMS Helm chart can be downloaded from NVIDIA NGC. To do so, use the following command:

```
helm fetch https://helm.ngc.nvidia.com/nvaie/charts/triton-management-service-1.4.0.tgz --username='$oauthtoken' --password=<YOUR API KEY>
```

Extracting the `values.yaml` file from the downloaded chart's TAR file is easy. To do so, use the following command:

```
helm show values triton-management-service-1.4.0.tgz > values.yaml
```

This will create a `values.yaml` file the current directory, which can modified to meet deployment needs.

Note

Download TMS Helm Chart from NGC

See Helm Chart Values for a listing of the configurable values.

Configuring the API Server Pod

By default, TMS requests minimal CPU and memory resources from Kubernetes to run the pod containing the API server and database. While this works fine for initial testing of TMS's features and for smaller, more stable deployments, it is likely to be insufficient if many clients are expected to be making concurrent API calls. In that situation, it is highly recommended that system administrators change the default settings.

To change the default settings, use the configuration options in `server.resources` in the `values.yaml` file. The amount of CPU and memory resources is relatively low compared to that of the database. For that reason, it is recommended that initially the database be allocated 75% of the available resources, and the API server the other 25%. Below is a sample configuration which would do this on a node with 8 CPUs and 16Gi of memory.

```
resources:
apiServer:
cpu: 2
memory: 4Gi
database:
cpu: 6
memory: 12Gi
```

Kubernetes Secrets

Setting up secrets in Kubernetes for TMS is fairly straightforward, and we'll cover the basics here.

Note that creation of Kubernetes secrets requires sufficient cluster privileges, and therefore might, if you lack sufficient privileges, require a cluster administrator to create them on your behalf.

Container Pull Secrets

TMS Helm chart will include any secrets listed under `values.yaml#images.secrets`. The default `values.yaml` file contains an example secret named "ngc-container-pull".

To create an image-pull secret, use:

```
kubectl create secret docker-registry <secret-name> --docker-server=<docker-server-urn> --docker-username=<username> --docker-password=<password>
```

Then which ever value was chosen for `&lt;secret-name&gt;` add to the `values.yaml#images.secrets` list.

Configuring Model Repositories

To connect to a model repository, see the model repository page.

Configuring Autoscaling

To enable and configure autoscaling, see the separate autoscaling configuration guide.

Configuring Triton Containers

TMS allows the TMS administrator to configure some aspect of the containers that will be created for Triton instances. These can be configured via the top-level `triton` object in `values.yaml`.

Currently, only resource constraints are specified in this section. These are all listed under `resources`. TMS admins may specify both the `default` resources that Triton containers will get, as well as the `limits.maximum` values that users may request on a per-lease basis.

A sample configuration is shown below.

```
triton:
resources:
default:
cpu: 2
gpu: 1
systemMemory: 4Gi
sharedMemory: 256Mi
limits:
minimum:
cpu: 1
```

```
gpu: 1
systemMemory: 1Gi
sharedMemory: 128Mi
maximum:
cpu: 4
gpu: 2
systemMemory: 8Gi
sharedMemory: 512Mi
```

The fields in both `default`, `minimum` and `maximum` sections are defined as follows.

Each value in the `maximum` section must be at least as large as the `default` and `minimum` value.

Each value in the `minimum` section must be smaller than the `default` and `maximum` value.

- `cpu`: The number of whole or factional CPUs assigned to Triton. Can be specified either a number of cores (e.g. `4`), or a number followed by `m`, which represents milli-CPUs (e.g. `1500m`).
    - Minimum value: `1` (or `1000m`).
    - Default: `2`
- `gpu`: The number of whole GPUs assigned to Triton. Must be a whole number – GPUs cannot be fractionally assigned.
    - Minimum value: `0`
    - Default: `1`
- `repositorySize`: The amount of disk space allocated for Triton model repository, as a number plus units (e.g. `4Gi`).
    - Units allowed: `Mi`, `Gi`, `Ti`

    - Minimum value: `256Mi`
    - Default: `2Gi`

- `systemMemory`: The amount of system memory, as a number plus units (e.g. `4Gi`).
    - Units allowed: `Ki`, `Mi`, `Gi`, `Ti`
    - Minimum value: `256Mi`, and at least `128Mi` more than `sharedMemory`.
    - Default: `4Gi`

- `sharedMemory`: The amount of shared memory, as number plus units (same units as `memory`).
    - Minimum value: `32Mi`
    - Default: `256Mi`
    - *Note*: Some backends (e.g. PyTorch) allow the user to use shared memory to allocate tensors.

      If you plan on using this, make sure you set a higher value.

Configuring Persisted Database

To enable and configure TMS to persist database contents, a volume claim bounded to a sizeable kuberenetes persistent volume must be provided to `values.yaml#server.databaseStorage.volumeClaimName`.

In the case of server failure or restart, TMS will be able to reload the contents of the database from this volume.

It should be noted that server performance can be affected by slow or unreliable storage solutions used for the persisted volume.

TMS Deployment Using Helm

Assuming you've followed the steps above, and downloaded the TMS Helm chart, exported its `values.yaml` file, and modified it as necessary, use the following command to install (aka deploy) TMS:

```
helm install <name-of-tms-installation> -f values.yaml triton-management-service-1.0.tgz
```

Security Considerations

The Kubernetes cluster where TMS is installed should be properly secured according to best practices and the security posture of your organization.

Any additional, optional services connected to TMS such as Prometheus and Prometheus adapter should also be secured. We recommend the cluster administrator properly secure access to any S3 or other external model repositories which TMS will utilize. We reccomend leverating encryption in transit and at rest, scoping access to cluster resources following the principle of least privilege, as well as configuring audit logging for your cluster.

TMS default configuration does not allow connections from outside of the Kubernetes cluster. The user assumes responsibility for securing any external connections when changing the default configuration values.

Useful Links & Additional Resources

- NVIDIA GPU Cloud
- Kubernetes
    - Secrets
- Helm
    - Download & Installation
    - Commands
    - Charts
- Triton User Guide

Model Repositories

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.

Model repositories hold the model artifacts that will be loaded into and served by the deployed Triton Inference Servers. Model repositories for Triton Management Service are similar in structure and content to Triton Inference Server model repositories, but there are different options and configurations for the available locations.

In general, model repositories are configured by specifying the remote location of the repository (where the method of specifying the location is dependent on the type of repository) as well as a *Repository Name* when you deploy TMS. TMS operations requiring references to the model repository (i.e. lease creation requests) will use the configured Repository Names. Several different types of model repository are available.

HTTP(S)

TMS Configuration

HTTP(S) model repositories are not required to be pre-specified in the TMS `values.yaml` file. However, you can associate Kubernetes Secret with a particular HTTP url in the `values.yaml` file, in which case TMS will provide the contents of the secret in the `Authorization` request header:

```
# values.yaml
server:
modelRepositories:
https:
```

```
- secretName: Name of the Kubernetes secret to read and provide as a
Authorization header for download requests.
targetUri: URL of the remote web-sever in \<domain_label_or_ip_address\>/\<path\>
format, used to determine if secrets apply to a model request or not.
```

The default `values.yaml` file contains an example secret named “ngc-model-pull”.

The `targetUri` is used to determine which secret is best suited to be used to download a given model based on the model’s URN. URN matching is broken up into two parts:

1. Match the DNS label right to left, or absolute match of an IP Address. For example: `models.company.com` would match `cdn.models.company.com`, but would not match `models.cdn.company.com`.

2. Match the path portion of URN from left-to-right. For example: `internal-cdn/repository` would match `internal-cdn/repository/ai_models`, but would not match `internal-cdn/ai_models/repository`.

To create a model-pull secret, use:

```
kubectl create secret generic <secret-name> --from-file <secret-name>
```

Then which ever value was chosen for `&lt;secret-name&gt;` add to the `values.yaml#server.modelRepositories.https` list with the corresponding `targetUri` value.

Setting up the Repository

Models in HTTP(S) repositories should be zipped versions of the directories in a Triton Model Repository, served by some kind of web server and accessible through HTTP GET requests.

For example, if the triton model repository is structured as follows:

```
model_repository/
    my_model
    1
    model.onnx
  config.pbtxt
```

Then you should serve a file `my_model.zip` that contains one of the following file layouts:

1.
   ```
   $ unzip -l my_model.zip
   Archive: my_model.zip
   Length Date Time Name
   --------- ---------- ----- ----
   0 2022-04-28 22:27 1/
   356 2022-04-28 22:27 1/model.onnx
   59 2022-06-01 21:12 config.pbtxt
   --------- -------
   415 3 files
   ```

2.
   ```
   $ unzip -l my_model.zip
   Archive: my_model.zip
   Length Date Time Name
   --------- ---------- ----- ----
   0 2022-07-08 00:23 my_model/
   59 2022-06-01 21:12 my_model/config.pbtxt
   0 2022-04-28 22:27 my_model/1/
   356 2022-04-28 22:27 my_model/1/model.onnx
   --------- -------
   415 4 files
   ```

The `my_model.zip` file – and any other zip files with a similar structure – can be served by a wide variety of web server. One approach is to use the `http.server` module in the Python standard library. In a directory containing the zip file, execute the command

```
python -m http.server --directory .
```

This will serve the model with a URI `http://localhost:8000/my_models.zip`.

Model URI

To refer to a model in an HTTP(S) repository, use the full URL of the server. For example:

```
tmsctl lease create -t ${tms_address} -m
"name=my_model,uri=http://www.example.com/models/my_model.zip"
```

Persistent Volume Claim

TMS Configuration

TMS enables TMS administrators to provide model repositories from Kubernetes Persistent Volume Claims for requested Triton instances.

To enable requested Triton instances to load models from a persistent volume claim, provide the name of the particular Kubernetes persistent volume claim in an entry under `values.yaml#server.modelRepositories.volumes`, along with a valid name for the repository. The Persistent Volume Claim will then be mounted as a volume onto any Triton pod launched by TMS.

```
# values.yaml
server:
modelRepositories:
volumes:
# Name used to reference this model repository as part of lease acquisition.
# May contain only lowercase alphanumeric characters (without spaces, hyphens `-` are
permitted).
- repositoryName: volume-models
# Kubernetes persistent volume claim (pvc) used to fetch models.
volumeClaimName: example-volume-claim
```

Setting up the Repository

Persistent Volumes in Kubernetes are cluster resources that can be consumed – like a node, while a Persistent Volume Claim is particular request to use that resource – like a pod. Since model repositories in TMS are used by multiple Triton instances, you'll need to create a specific PVC for your repository that can then be mounted onto multiple pods.

One way to set up the repository is to create the model repository outside of kubernetes in a piece of storage that can be consumed as a Persistent Volume. Then, you can define that piece of storage as a Persistent Volume, and then attach a Persistent Volume Claim to it that allows kubernetes pods to consume that particular piece of storage. The NFS Model Repository path in the quickstart guide gives an example of this.

Persistent Volume Claims are generally exposed directly as file systems, so to create a model repository you can use the same structure as a Triton Inference Server model repositories – for example:

```
model_repository/
    my_model
    1
      model.onnx
    config.pbtxt
```

See the following resources for creating Persistent Volumes and Claims backed by various types of storage:

- NFS: TMS Quickstart Guide
- AWS Elastic Block Storage: AWS Documentation. Only supported on Amazon EKS clusters.
- Azure Blob Storage: Azure Documentation. Only supported on Azure Kubernetes Service clusters.
- Azure Files: Azure Documentation. Only supported on Azure Kubernetes Service clusters.

Model URI

To refer to a model in a PVC repository, prefix the model name with `model://` and the name of the model repository configured in the `values.yaml` file. For example:

```
tmsctl lease create -t ${tms_address} -m "name=my_model,uri=model://volume-models/my_model"
```

S3 Object Store

TMS Configuration

To configure access to an S3 compatible object store, you must specify a Repository Name, a Bucket Name, and an S3 service Endpoint.

```
#values.yaml
server:
modelRepositories:
s3:
# Name used to reference this model repository as part of lease acquisition.
# May contain only lowercase alphanumeric characters (without spaces, hyphens `-` are permitted).
- repositoryName: repo0
# Name of the S3 bucket used to fetch models.
bucketName: tms-models
# Service URL of the S3 bucket.
# If both 'endpoint' and 'awsRegion' fields are specified, TMS will default to using the value from 'endpoint'.
# Must be a valid URL designating to an existing endpoint (eg. "http:/s3.us-west-2.amazonaws.com" or "http:/play.min.io:9000").
# Refer here to learn more:
https://docs.aws.amazon.com/general/latest/gr/s3.html#amazon_s3_website_endpoints.
endpoint: "https://s3.us-west-2.amazonaws.com"
```

If your S3 Object Store is an actual AWS S3 bucket, you can provide the AWS Region of your bucket instead of the explicit endpoint

```
#values.yaml
server:
modelRepositories:
s3:
- repositoryName: repo0
bucketName: tms-models
# Service region code of AWS S3 Bucket.
# Field is for S3 buckets exclusively deployed through AWS.
# Non-AWS S3 Buckets should be configured through the `endpoint` field.
# Must be a valid code designating to existing AWS region (eg. "us-west-2").
# Refer here to learn more:
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-
zones.html).
awsRegion: "us-west-2"
```

If your model repository is in a private S3 bucket that requires access credentials, you have two options.

First, you can create a Kubernetes Secrets containing an *access key ID* and one containing a *secret access key* that represent the authority to list and retrieve the objects in the bucket. Then, you specify those secrets in the `values.yaml` file:

```
#values.yaml
server:
modelRepositories:
S3:
- repositoryName: repo0
bucketName: tms-models
endpoint: "https://s3.us-west-2.amazonaws.com"
# Name of the Kubernetes secret to read and provide as the access key ID to download
objects from the S3 bucket.
# Optional value when IAM or default AWS environment variables are not used for
authorizing TMS to read from an S3 bucket.
accessKey: "access-key-secret-name"
# Name of the Kubernetes secret containing the secret access key to read from the S3
bucket
# Optional value when IAM or default AWS environment variables are not used for
authorizing TMS to read from an S3 bucket.
secretKey: "secret-key-secret-name"
```

The other option is applicable when you are using AWS S3 buckets and when TMS is to be deployed on EKS. Instead of explicitly providing the key ID and secret key, you must associate an AWS IAM role which has `s3:ListBucket` and `s3:GetObject` permissions for that bucket with the TMS kubernetes service account. You can do this by providing the Amazon Resource Name of that IAM role in the `values.yaml` file.

```
#values.yaml
server:
security:
aws:
# AWS IAM role used read models S3 buckets configured in `modelRepositories.S3`.
role: arn:aws:iam::00000000:role/Tms-s3-role
```

You should also ensure that the role you provide here has a trust policy that allows the `tms-triton` service account to assume that role. For example, you can create this IAM role with the following `eksctl` command:

```
eksctl create iamserviceaccount --cluster tms-cluster --name=tms-server --attach-
policy-arn=arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess --role-only --role-
name=Tms-s3-role --approve
```

Please see the documentation on Configuring a Kubernetes service account to assume an IAM role to learn more.

Setting up the Repository

S3 model repositories should be set organized into folders that are similar to the following structure:

```
tms-models #bucket name
      my_model #S3 folder
      1
        model.onnx
      config.pbtxt
```

All model folders (like the `my_model` folder above) should either be at the top level of your bucket, or contained in a single parent directory. If your model repository is not at the top level folder of your bucket you should include the full path when referring to the model in `lease` commands.

You should also ensure that you have an IAM role available that has access to the bucket (and folder) containing the models or that the bucket is publicly accessible.

Model URI

To refer to a model in an S3 repository, prefix the model name with `model://` and the name of the model repository configured in the `values.yaml` file. TMS will internally resolve this to the correct S3 url. For example:

```
tmsctl lease create -t ${tms_address} -m "name=my_model,uri=model://aws-models/my_model"
```

Triton Image Allowlist

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.

Overview

The Triton image allowlist is a management feature used to controls which Triton images can be used to created Triton pools and bespoke Triton instances. This feature gives administrators some basic controls over which images can and cannot be used when creating Triton instances. Simply put, if an image is not in the allowlist, it cannot be used to create a new Triton instance. The Triton allowlist service is used to inspect and modify the allowlist.

Initially, the allowlist only contains the default Triton image, as configured during installation. This can be seen by running `tmsctl allowlist list`. Right after installing TMS, it should look like this:

```
$ tmsctl allowlist list
nvcr.io/nvidia/tritonserver:23.09-py3
```

In this state, new Triton instances can only be created with `nvcr.io/nvidia/tritonserver:23.09-py3` as the Triton image. For example, trying to use `nvcr.io/nvidia/tritonserver:23.08-py3` will fail.

```
$ tmsctl lease create --triton-image nvcr.io/nvidia/tritonserver:23.08-py3 -m
"name=$MODEL_NAME,uri=$MODEL_URI"
```

```
fatal: Requested Triton container image ("nvcr.io/nvidia/tritonserver:23.08-py3") is
unreachable or not provided in a supported format. Unreachable container images
either do not exist or require privileges not granted to the server.
(triton_options_bespoke.triton.container_image @ Acquire)
```

New entries can be added via `tmsctl allowlist add`.

```
$ tmsctl allowlist add nvcr.io/nvidia/tritonserver:23.08-py3
Added nvcr.io/nvidia/tritonserver:23.08-py3
$ tmsctl allowlist list
nvcr.io/nvidia/tritonserver:23.08-py3
nvcr.io/nvidia/tritonserver:23.09-py3
```

After running the above, we can create new bespoke Triton instances and Triton pools specifying `nvcr.io/nvidia/tritonserver:23.08-py3` as the image.

```
$ tmsctl lease create --triton-image nvcr.io/nvidia/tritonserver:23.08-py3 -m
"name=$MODEL_NAME,uri=$MODEL_URI"
Lease da21b2c0e68b49ffa8f0f6db0b030128
State: Valid
Expires: 2023-10-18T15:43:53Z
Triton: triton-6d8c9d13.tmsns.svc.cluster.local
<nvcr.io/nvidia/tritonserver:23.08-py3>
Models:
Name Url Status
<model_name> <model_url> Ready
```

Entries can be removed via `tmsctl allowlist rm`.

```
$ tmsctl allowlist rm nvcr.io/nvidia/tritonserver:23.10-py3
Removed nvcr.io/nvidia/tritonserver:23.10-py3
$ tmsctl allowlist list
nvcr.io/nvidia/tritonserver:23.09-py3
```

After running the above, any attempts to create new bespoke Tritons or Triton pools specifying `nvcr.io/nvidia/tritonserver:23.08-py3` as the image will fail.

Further details about the individual operations are given below.

Triton Image Allowlist Operations

Following are the list of operations with the Triton Allowlist Service:

1. `TritonAllowlist/Append` appends a Triton container image to the list containing the allowed Triton container images.

   The server will return success or failure depending on whether the requested image could be added to the allowlist.

   Attempting to add images which are already present in the allowlist will not result in any changes.

2. `TritonAllowlist/List` lists the allowed Triton container images.

   This RPC begins streaming a response once the request has been received.

   Each response message contains a Triton image that belongs to the list.

3. `TritonAllowlist/Remove` removes a Triton container image from the list containing the allowed Triton container images.

   The server will return success or failure depending on whether the image could be removed from the allowlist.

Leases

⚠ **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.

Overview

A lease is the primary organizational unit used by TMS. Leases allow users to describe which models to load, as well as control their lifecycle. Leases provide a convenient means of describing the Triton instance where the models should be loaded, without having to manage the Triton instance directly. Additionally, leases allow you to configure features like autoscaling or sharing Triton servers.

A lease consists of the following information:

- A model or ensemble of models that will be loaded together in a Triton instance.
- A description of the Triton instance where to load the models. This can be either:
    - A bespoke Triton instance which will be created just for this lease and will not be shared. Bespoke Triton instances support autoscaling.
    - The name of a pre-existing [Triton pool](./triton-pools.md], where the lease may share a Triton instance with other leases.
- Information about the duration of the lease, including whether it supports automatic renewal based on usage.

For example, a simple lease might have the following characteristics:

- It consists of a single model named `model_A`, along with the URI from which to fetch it.
- Uses a bespoke Triton instance with the default configuration as set by the system administrator.
- Lasts for the default duration as set by the system administrator.

A more complex lease might look as follows:

- It consists of an ensemble of models. The models are named `model_A`, `model_B`, and `model_C`, and each is specified along with the URI from which to fetch it. These models will be loaded in the specified order to ensure the ensemble works properly.
- It describes a bespoke Triton instance on which to run with custom resource requirements:
  - The Triton instance supports autoscaling up to four copies, scaling up when inference queue time exceeds 200ms.
  - Each Triton requires 2 GPUs, 4 CPUs, and 32 GB of main memory.
- The lease should remain active for 8 hours, and automatically renew for another four hours so long as it has served inference requests in the 30 minutes before it would expire.

The exact lifecycle of a lease will vary depending on the application requirements, but they will all follow this general outline:

- Create a lease via a `Lease.Acquire()` API call and get the URL of the Triton instance where the models were loaded.
- Run inference against the models in the lease using the Triton inference API.
- Potentially renew the lease via `Lease.Rewnew()` calls, or let it automatically renew if it is configured to renew while still being used.
- Either manually release the lease via a `Lease.Release()` call, or let it expire. Either way, this will either free the associated Triton instance for bespoke leases or mark the resources as available for leases running on pooled Triton instances.

For more details on the available operations, see the section below. There is also a tutorial that will guide you through the basics of of working with leases.

Lease Operations

The Lease Service exposes a number of RPC end-points: `Acquire`, `Release`, `Renew`, and `Status`. The Triton Allowlist Service exposes the following RPC end-points: `Append`, `List` and `Remove`. Each of the end-point accept a single structured request and respond with a structured response.

> **Note**
>
> The gRPC protocol supports streaming requests and/or responses. This means that one or both sides of the interaction can stream data to the other. Functionally, this allows the server to being sending response data before the client has finished sending request data.

The expected order of operations with regards to the Lease Service are as follows:

1. `Lease/Acquire` to create a new lease with a specified set of AI models.

   Assuming the request is successful, the response will include a unique identifier and an expiration date for the new lease.

   All models in a lease acquire request are considered bundled. They cannot be loaded or unloaded separately. Additionally, all models in a lease will be loaded into the same instance of Triton Inference Server. If it is impossible to do so (e.g. insufficient memory), then the lease will be marked as invalid and any models successfully loaded models will be unloaded after the first model load failure is detected. TMS does not support partially loaded leases.

   A lease can created as part of a Triton Pool or using a bespoke Triton instance. This is determined by the use of the `triton_options` value in the gRPC API.

   This RPC begins streaming a response once the request has been received. The server will send a series of model status updates to the caller to show continued progress as the lease’s models are deployed. Model status updates will be sparse (not include status of every model every time).

The final response from the server will include status for every model in the request as well as data for the lease itself.

2. `Lease/Renew` to extend the lease’s duration. Once a lease is renewed it assigned a new expiration date.

   Once a lease has expired, it is no longer valid and any associated models will be unloaded and become unavailable. Any resources consumed by the lease are returned to the hosting Triton Inference Server to be used by future leases. In the case that a Triton Inference Server instance becomes unnecessary, it will be deleted and its resources returned to the cluster.

3. `Lease/Status` to get the current status of a specific lease.

   Requesting the status of an expired or released lease is a valid operation.

4. `Lease/Release` to terminate a lease before its expiration is reached.

   Once a lease has been released, it is no longer valid and any associated models will be unloaded and become unavailable. Any resources consumed by the lease are returned to the hosting Triton Inference Server to be used by future leases. In the case that a Triton Inference Server instance becomes unnecessary, it will be deleted and its resources returned to the cluster.

Autoscaling Leases

⚠ **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`).

Triton Pools & Quota Base Shared Tritons

⚠ **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.

Triton Pools enable TMS administrators to create a set of Triton instances which can be shared by any leases created and assigned to the pool. Multiple pools can exist simultaneously with each pool having its own definition and purpose.

Pool definitions allow for the specification of the container image used to deploy Triton instances, and the specification of the resources reserved for and assigned assigned to each TIS instance present in the pool. In addition, a pool definition includes a minimum and maximum pool size (aka. number of concurrent Triton instances the pool supports), and a per-instance quota value used by TMS to determine which Triton instances are best candidates for new leases to be assigned to.

Note: Triton Pools work best in clusters with homogeneous GPUs. TMS does not take GPU SKU into consideration when determining the capacity of Triton instances.

Triton Pool Options

Name

Triton Pools must be given a name. A pool's name must be unique among all other existing pools. A name can be reused once the name's previous pool has been deleted. This name is used to identify the pool when creating leases, or interacting with the pool.

Per-Instance Quota

Triton Pools must have a per-instance quota value. The quota value is used to determine which Triton instances have available “space” for new leases to be assigned to.

The meaning of the units the quota is defined as is determined by the pool’s creator. TMS applies no specific meaning to the value.

For example, a pool could be created in a cluster with GPUs which all have 40Gi of memory. The TMS administrator could then decide that each Triton instance should be assigned a per-instance quota value of 40, and expect that all leases deployed into the pool specify the amount of GPU memory they require in gigabytes.

In the above case, TMS would only compare a lease’s quota request against any available quota on Triton instances in the pool. TMS would not inspect Triton or the GPU to determine the actual amount of available GPU memory. There is no enforcement of quota values at runtime.

For example, it might be known ahead of time that each Triton instance in a pool is capable of hosting four leases. Therefore, the pool could be created with a per-instance quota value of 4, and each lease would specify a quota need value of 1.

In the above case, each unit of per-instance quota is equal to a single “computing slot” and each lease would consume one, or more, of them.

Instance Limits

Triton Pools are defined with a **minimum** and **maximum** number of Triton instances they’re allowed to create and host. When the minimum value is greater than zero, the pool will attempt to always have at least that number of Triton instances available.

These limits are used to determine when a pool scales the number of Triton instances present in the pool. When a lease is created and assigned to the pool and there are no available Triton instances with sufficient available quota to host the lease, the pool will attempt to add a new Triton instance only if it has not already reached it maximum capacity. When a pool has insufficient capacity and cannot add a new Triton instance, any lease creation attempt will be rejected.

Note: When attempting to create new Triton instances, pools are limited by available cluster resources.

Triton Definition

Triton Pool definitions include a definition for each Triton instance in the pool. Triton definitions include the Triton container image used, the number of logical CPU cores, the number of GPUs, and the amount of memory reserved and assigned to each Triton instance created by the pool.

Enforcement of Triton Backend Uniqueness

Triton Pool definitions include an option to not enforce that each Triton instance be restricted to the Triton backends used by the first lease deployed on it. Enforcement is enabled by default because the mixing of Triton backends is discouraged due to issues with memory management.

For example, a Triton instance is deployed with enforcement enabled. The first least deployed to the instance is an ensemble of two models; the first is a TensorFlow model, the second is a PyTorch model. The TensorFlow and PyTorch backends will each allocate as much memory as possible, effectively splitting the available memory between them.

From this point on, because enforcement is enabled, only leases which depend on the TensorFlow and/or PyTorch backends will be deployed to this Triton instance. When a second lease is deployed to this Triton instance, it will only contain models with backends meeting this requirement.

The first time TMS encounters a model, its backend is considered *unknown* and therefore cannot be assigned to any existing Triton instance. Once deployed, TMS will learn which Triton backend the model depends on and will update its record of the Triton instance to reflect the correct mix of backends active on the instance. Additionally, TMS will record the Triton backend information of the model such that all future deployments of the same model will be able to correctly select which Triton instances match the model's Triton backend requirements.

When enforcement is disabled, TMS will select Triton instances without taking into consideration for model backends and which Triton backends are active on instances. This simplifies instance selection, but incurs the risk of attempting to load a model with a backend that's not present on a Triton instance and the instance having insufficient memory available to the load the model.

Enabling enforcement is recommended unless extensive testing with a restricted set of models has been done to ensure Triton instance stability.

Quota Based Shared Triton Leases

Quota based shared Triton (QBST) leases are defined as one or more models with a specified quota consumption value and assigned to a Triton Pool. The specified quota consumption value, or quota, is used to determine how the lease’s models will be hosted. Leases with multiple models will always have all of its models hosted by a single Triton instance.

Quota

The **quota** value of a QBST lease defines the amount of “space” or resources the lease will consume. The units or meaning of the value the quota is defined as is determined by the pool’s creator. TMS applies no specific meaning to the value.

For example, a Triton Pool might define units in terms of “compute fraction” with each Triton instance being assigned a denominator value. For this example, we’ll assume each Triton instance is assigned a quota capacity of 8. Any lease assigned to this pool must specify what fraction of a while Triton instance the lease will consume. This is done by the lease’s quota value.

Continuing with the example, lease could be created which expects to consume a quarter of the capacity of a Triton instance would be assigned a quota value of 2. This lease could share the Triton instance with any combination of other leases whose quota sum is less than or equal to 6 (the remaining quota capacity).

It is up to the pool’s administrator to determine the units and meaning of a pool’s quota, and the measures by which a lease is expected to determine the amount quota it will consume.

TMS uses lease and pool quota values to determine how and where to place leases within a pool. TMS does not enforce any kind of resource utilization after a lease has been assigned to a Triton instance.

Extending the example above, a lease creator specifies that their lease consumes 2 quota units. In actuality the lease consumes 8 quota units (i.e. an entire Triton instance). Because the lease consumes significantly more resources than advertised, several of the loaded models, including models loaded for other leases, experience significant performance degradation and out of memory errors.

It is important to test the quota consumption values of leases before creating them in a production environment. Undervaluing leases can lead to performance degradation, out of memory errors, and Triton instance instability. Overvaluing leases, while often safer, can leave hardware under utilized and potentially cause capacity issues due external processes being forced to wait for available AI cycles.

TMS Metrics

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 provides a metrics endpoint from which Prometheus formatted runtime metrics can be retrieved.

The following Helm chart options can be used to affect how metrics are reported:

- `server.metrics.enabled` can be used to enable/disable metrics endpoint.
- `server.metrics.reportingWindow` can be used to configure the reporting window for metrics endpoint.
- `server.metrics.minimumVisibility` can be used to configure the which metrics are collected and reported.

By default, only high visibility are reported when metrics reporting is enabled.

Standard (high visibility) metrics are reported for each of the server's endpoints.

Metric Name	Description
tms_error_count	Number of errored requests during the reporting window.
tms_duration_avg_seconds	Average duration of successful requests during the reporting window.
tms_grpc_request_count	Number of gRPC requests made during the reporting window.

Additional metrics are available by adjusting the minimum-visibility Helm chart value. These metrics are self-describing as part of the Prometheus formatted output.

TMS GRPC API Package

⚠ **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.

With every release of Triton Management Service a zip-compressed archive of the gRPC IDL files is provided. These packages can be downloaded from NVIDIA's NGC Catalog.

Versions

It is important to use the version of the TMS gRPC API that matches the version of the TMS Server being communicated with. NVIDIA provides no guarantee of forward/backward compatibility for the programmatic interfaces of beta software. The TMS interface is still in development and expected to fluctuate.

Using the Package

1. Download the gRPC API Package, either using the web interface (provided above) or using the NGC CLI with the following command `ngc registry resource download-version "nvaie/triton-management-service_grpc-api-bundle:1.4.0"`. Notice that the desired version is the last component of the command, and can be adjusted to match the version of TMS as necessary.

2. Extract the contents of the downloaded package.

   Linux

   ```
   $ unzip ./files.zip
   Archive: ./files.zip
   ```

```
inflating: files/triton-management-service_grpc-api-bundle-v1.4.0.zip
$ unzip triton-management-service_grpc-api-bundle-v1.4.0.zip
Archive: triton-management-service_grpc-api-bundle-v1.4.0.zip
inflating: bespoke-triton.proto
inflating: pooled-triton.proto
inflating: lease-state.proto
inflating: triton-allowlist-service.proto
inflating: model.proto
inflating: model-state.proto
inflating: triton-pool-service.proto
inflating: triton.proto
inflating: lease-service.proto
inflating: lease-name-service.proto
inflating: common.proto
inflating: lease-duration.proto
inflating: error-code.proto
inflating: lease-event.proto
inflating: triton-state.proto
```

Windows Extracting the IDL on Windows will require a tool like 7-Zip which can handle compressed TAR files.

```
> 7z x .\files.zip
7-Zip 23.01 (x64) : Copyright (c) 1999-2023 Igor Pavlov : 2023-06-20
Scanning the drive for archives:
1 file, 18579 bytes (19 KiB)
Extracting archive: .\files.zip
--
Path = .\files.zip
Type = zip
Physical Size = 18579
Everything is Ok
Size: 19324
Compressed: 18579
> 7z x ./triton-management-service_grpc-api-bundle-v1.4.0.zip
```

```
7-Zip 23.01 (x64) : Copyright (c) 1999-2023 Igor Pavlov : 2023-06-20
Scanning the drive for archives:
1 file, 19324 bytes (19 KiB)
Extracting archive: triton-management-service_grpc-api-bundle-v1.4.0.zip
--
Path = triton-management-service_grpc-api-bundle-v1.4.0.zip
Type = zip
Physical Size = 19324
Everything is Ok
Files: 15
Size: 65633
Compressed: 19324
```

Once extracted you should have the following list of files:

```
bespoke-triton.proto
lease-duration.proto
lease-service.proto
model.proto
triton-pool-service.proto
common.proto
lease-event.proto
lease-state.proto
pooled-triton.proto
triton-state.proto
error-code.proto
lease-name-service.proto
model-state.proto
triton-allowlist-service.proto
triton.proto
```

3. Use the `protoc` compiler to generate language specific code files from the provided IDL (*.proto) files. The necessary compiler and tools can be downloaded from the Protocol Buffers Release Page on GitHub. The latest, release version is v23.4. Download the version of the tools that best suite your platform.

Once all tools are downloaded, use them to generate code in your language of choice. Use Protocol Buffers Getting Started as a guide as needed.

Example for JavaScript code generation:

```
$ protoc --proto_path=proto --
js_out=library=tms_model_state,binary:js_autogen proto/model-state.proto
```

Will create a JavaScript file named `tms_model_state.js` from `proto/model-state.proto`, and output the results to a folder named `js_autogen` (must exist before running `protoc`). Notice that the above assumes the *.proto file are contained in a folder named `proto` which is a child of the current working folder.

Triton Management Service Control

⚠ **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.

Triton Management Service Control (`tmsctl`) is a command line utility for interacting with Triton Management Service (TMS). It provides commands for interacting with TMS, creating and managing leases, and managing service configuration.

Download tmsctl from NGC

This document includes a reference of all commands, as well as an explanation of some configuration options that can be used to control the behavior of `tmsctl`.

Command Reference

- allowlist
  - allowlist add
  - allowlist list
  - allowlist rm
- lease
  - lease create

    - lease list
    - lease release
    - lease renew
    - lease status
- lease name
    - lease name create
    - lease name delete
    - lease name list
- pool
    - pool create
    - pool delete
    - pool list
    - pool status
- target
    - target add
    - target list
    - target remove
    - target set

Common Options

Many commands share a few common options. These are documented here.

`(--target|-t):&lt;target&gt;`

Specify the instances of TMS on which to perform the operation.

Valid targets are URLs beginning with `http://` or `https://`, or the name of a named instance (see the target command).

Examples:

- `--target https://www.example.com:30345`: will connect to TMS at the specified URL.
- `--target my_tms`: will connect to a TMS instance named `my_tms` previously specified via tmsctl target add.

Unless a default target is specified via tsmctl target commands, all commands require the `--target` option.

`(--porcelain|-z)`

Formats output in an easy-to-parse format for scripts; avoids fancy formatting for human readers.

Porcelain output does not attempt to colorize output or insert unnecessary whitespace to improve readability.

This output is not guaranteed to be stable between releases.

Lease

The `tmsctl lease` commands allow you to perform operations on leases, such as creating, renewing, and releasing them.

Lease Create

```
tmsctl lease create [(--target|-t):&lt;target&gt;] [(--porcelain|-z)] (--model|-m):&lt;model&gt;    [{Duration Options}] [{Automatic Renewal Options}] [{Autoscaling Options}] [{Triton Options}]
```

```
tmsctl lease create [(--target|-t):&lt;target&gt;] [(--porcelain|-z)] (--model|-m):&lt;model&gt;   (--triton-pool|-p):&lt;name&gt; --quota:&lt;quota&gt; [{Duration Options}] [{Automatic Renewal Options}]
```

Connects to `&lt;target&gt;` and creates a lease for Triton Inference Server to serve one or more `&lt;model&gt;`.

Provides a `&lt;lease-id&gt;` for the newly created lease when successful.

An error code will be returned when no default `&lt;target&gt;` exists and `(--target|-t)` has not been specified.

To learn about how to package models, please see the model repository documentation.

Options

`--model|-m:&lt;model&gt;`

`&lt;model&gt;` is a comma-separated list of `&lt;name&gt;=&lt;value&gt;` pairs describing a model.

This option can be included multiple times, once for each unique model required. All models in a lease will be loaded and provided by a single Triton Inference Server. If the set of models is too large or requires too many resources, Triton may fail to load them. In the event of a failure an error will be returned and the lease made invalid.

The allowed `&lt;name&gt;=&lt;value&gt;` pairs are:

- `name` (required): the name of the model. Must match the name expected by Triton.
- `uri` (required): the URI from which to get the model.
- `count` (optional, default=0): the number of instances of the model to load, or 0 to use the model's default count.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Automatic Renewal Options

`--auto-renew`

Makes new new lease eligible for automatic renewal. Auto-renewal rules are determined by the server.

`--auto-renew-activity-window`

The time window during which a lease must be used before expiration for it to be automatically renewed. Durations must be specified in `(&lt;hours&gt;h)(&lt;minutes&gt;m)(&lt;seconds&gt;s)` format (e.g. `1h30m15s`, `1h30m`, `1h`, `1m30s`).

Autoscaling Options

Options related to automatically scaling the number of instances of a lease.

Note: Autoscaling and using pooled Triton instances are mutually exclusive. If any of these options are used at the same time pool options are used, an error is reported.

`--enable-autoscaling`

Enable autoscaling for this lease. This is automatically turned on if any of the other options related to autoscaling are set.

`--autoscaling-max-replicas`

Set the maximum number of replicas when autoscaling. Valid values are any positive integer. Implies `--enable-autoscaling` when provided.

`--autoscaling-min-replicas`

Set the minimum number of replicas when autoscaling. Valid values are any non-negative integer. Implies `--enable-autoscaling` when provided.

`--autoscaling-metric-cpu-utilization`

Set the state of autoscaling based on CPU utilization. Valid values are `enable`, `disable`, and `server-default` (default).

`--autoscaling-metric-cpu-utilization-threshold`

Set the threshold for autoscaling based on CPU utilization. The value must be a number between 0 (exclusive) and 100 (inclusive).

`--autoscaling-metric-gpu-utilization`

Set the state of autoscaling based on CPU utilization. Valid values are `enable`, `disable`, and `server-default` (default).

`--autoscaling-metric-gpu-utilization-threshold`

Set the threshold for autoscaling based on GPU utilization. The value must be a number between 0 (exclusive) and 100 (inclusive).

`--autoscaling-metric-queue-time`

Set the state of autoscaling based on queue time. Valid values are `enable`, `disable`, and `server-default` (default).

`--autoscaling-metric-queue-time-threshold`

Set the threshold for autoscaling based on queue time. Durations must be specified in `(&lt;hours&gt;h)(&lt;minutes&gt;m)(&lt;seconds&gt;s)` format (e.g. `1h30m15s`, `1h30m`, `1h`, `1m30s`).

`--autoscaling-metric-queue-time-percentage`

Set the state of autoscaling based on the percentage of time inference requests spend in the queue. Valid values are `enable`, `disable`, and `server-default` (default).

`--autoscaling-metric-queue-time-percentage-threshold`

Set the threshold for autoscaling based on the percentage of time inference requests spend in the queue. The value must be a number between 0 (exclusive) and 100 (inclusive).

Duration Options

`--duration`

The initial duration of the lease. Durations must be specified in `(&lt;hours&gt;h)(&lt;minutes&gt;m)(&lt;seconds&gt;s)` format (e.g. `1h30m15s`, `1h30m`, `1h`, `1m30s`).

`--renewal-duration`

The duration for which the lease renews when renewed. Durations must be specified in `(&lt;hours&gt;h)(&lt;minutes&gt;m)(&lt;seconds&gt;s)` format (e.g. `1h30m15s`, `1h30m`, `1h`, `1m30s`).

Triton Options

Options related to how the Triton instance is created.

Note: Specifying Triton options and using pooled Triton instances are mutually exclusive. If any of these options are used at the same time pool options are used, an error is reported.

`--triton-image|-i`

Specifies the Triton container image to be used to deployment the lease. `&lt;triton-image&gt;` must be in the allowed list of Triton container images, managed by the TMS administrator.

`--triton-resources`

Specifies the hardware resources to allocate to the Triton server for this lease.

Expected format:
```
cpu=&lt;count&gt;,gpu=&lt;count&gt;,repository-size=&lt;memory&gt;,system-memory=&lt;memory&gt;,shared-memory=&lt;memory&gt;
```
, where `&lt;count&gt;` is expected to be a positive integer, and `&lt;memory&gt;` is expected to be a positive number followed by Ki, Mi, or Gi to indicate the amount of memory.

When not provided and a pool is not specified, server-configured defaults are used.

Triton Pool Options

Options related to the creation of Triton pools.

Note: Specifying pool options is mutually exclusive with autoscaling options and Triton options. If any of these options are used at the same time as those, an error is reported.

`--triton-pool|-p`

Specifies the Triton Pool, by name, that the lease should be deployed into. Must be specified along with the `--quota` option.

`--quota`

Specifies the amount of available quota the lease will consume from a single instance of Triton in the target pool. Must be specified along with `(--triton-pool|-p):&lt;name&gt;`. Must be greater than zero.

Lease List

`tmsctl lease list [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

`tmsctl leases [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Connects to `&lt;target&gt;` and list all active and pending leases.

By default, a summary of each lease will be listed. Adding the `--verbose` flag will increase the amount of output.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Lease Release

`tmsctl lease release &lt;lease-id&gt; [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Connects to `&lt;target&gt;` and release lease `&lt;lease-id&gt;`.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

`&lt;lease-id&gt;`

Unique identifier of a lease.

Lease Renew

`tmsctl lease renew &lt;lease-id&gt; [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Connects to `&lt;target&gt;` and renew lease `&lt;lease-id&gt;`.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

`&lt;lease-id&gt;`

Unique identifier of a lease.

Lease Status

`tmsctl lease status &lt;lease-id&gt; [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Connects to `&lt;target&gt;` to get the current status of a lease.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

`&lt;lease-id&gt;`

Unique identifier of a lease.

Lease Name

Provides functionality for managing names associated with leases.

Lease Name Create

tmsctl lease name create (--name:)&lt;lease-name&gt; (--lease:)&lt;lease-id&gt; [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]

Creates a new `&lt;lease-name&gt;` for an existing lease `&lt;lease-id&gt;`.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`--lease`

The unique identifier of a lease to which the name should refer. May be be specified without the `--lease` flag if the name is specified first.

`--name`

The name of a lease to create. May be be specified without the `--name` flag if it is the first positional argument.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

`&lt;lease-id&gt;`

The unique identifier of a lease to which the name should refer.

`&lt;lease-name&gt;`

The name of a lease to create.

Lease Name Delete

`tmsctl lease name delete (--name:)&lt;lease-name&gt; ((--lease:)&lt;lease-id&gt;|--force) [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Deletes an existing `&lt;lease-name&gt;` for a specified lease `&lt;lease-id&gt;`.

The lease `&lt;lease-id&gt;` associated with the `&lt;lease-name&gt;` does not have to be specified if the `--force` flag is provided.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`--force`

Delete the name regardless of what lease it currently refers to.

`--lease`

The unique identifier of a lease to which the name should refer. If this does not match what the name actually refers to, an error occurs. May be be specified without the `--lease` flag if the name is specified first.

`--name`

The name of a lease to delete. May be be specified without the `--name` flag if it is the first positional argument.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

`&lt;lease-id&gt;`

The unique identifier of a lease to which the name should refer. If this does not match what the name actually refers to, an error occurs. May be be specified without the

`--lease` flag if the name is specified first.

`&lt;lease-name&gt;`

The name of a lease to delete.

Lease Name List

`tmsctl lease name list [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Connects to `&lt;target&gt;` and lists all lease names associated with existing leases.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

`tmsctl lease name move (--name:)&lt;lease-name&gt; ((--source-lease:)&lt;source-lease-id&gt;|--force) (--target-lease:)&lt;target-lease-id&gt; [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Moves a `&lt;lease-name&gt;` from one lease `&lt;source-lease-id&gt;` to another `&lt;target-lease-id&gt;`.

The lease `&lt;source-lease-id&gt;` associated with the `&lt;lease-name&gt;` does not have to be specified if the `--force` flag is provided.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`--force`

Move the name regardless of what lease it currently refers to.

`--name`

The name of a lease to move. May be be specified without the `--name` flag if it is the first positional argument.

`--source-lease` The unique identifier of the lease to which the name should currently refer. If this does not match what the name actually refers to, an error occurs. May be be specified without the `--source-lease` flag if the name is specified first.

`--target-lease` The unique identifier of the new lease to which the name should refer. May be be specified without the `--target-lease` flag if the name and source lease are specified first.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Pool

Commands for managing Triton pools.

Pool Create

```
tmsctl pool create &lt;name&gt; (--instance-quota|-q):&lt;quota&gt;   [--disable-backend-uniqueness] [(--instances|-c):&lt;count&gt;]   [--triton-resources:&lt;resources&gt;] [--triton-container:&lt;image-name&gt;]   [(--target|-t):&lt;target&gt;][(--porcelain|-z)]
```

An error code will be returned when no default exists and (–target|-t) has not been specified.

Options

`--disable-backend-uniqueness`

Disables Triton backend uniqueness enforcement.

By default, Triton pools segregates Triton instances by the Triton backend(s) used by models loaded. Disabling this segregation enables leases with models with differing Triton backend requirements to be collocated. The mixing of Triton backends can lead to runtime out-of-memory errors.

`--instance-quota|-q`

Specifies the maximum allocatable quota per Triton instance in the pool as an integer. This value limits the number of leases which can be assigned to a single Triton instance in the pool based. Leases deployed into the pool must specify the amount of quota the consume and will only be place on Triton instances with sufficient remaining quota. Specifying a quota larger than is physically available can lead to resource exhaustion errors and server crashes. When the provided value is outside the configured server limits, the pool creation request will fail. Value is required and must be a value greater than zero.

`--instances|-c`

Specifies the minimum and maximum number of Triton instances allowed to exist in the pool. Expected format: `&lt;minimum&gt;,&lt;maximum&gt;` where `&lt;minimum&gt;` and/or `&lt;maximum&gt;` can be replaced with `*` to use the configured server default value. When not provided the configured server defaults will be used.

`--triton-container`

Specifies the Triton container image to be used for all Triton instances in the pool. `&lt;image-name&gt;` must be in the allowed list of Triton container images managed by the TMS administrator.

`--triton-resources`

Specifies the hardware resources to allocate to the Triton server for this lease.

Expected format:
`cpu=&lt;count&gt;,gpu=&lt;count&gt;,repository-size=&lt;memory&gt;,system-memory=&lt;memory&gt;,shared-memory=&lt;memory&gt;`
, where `&lt;count&gt;` is expected to be a positive integer, and `&lt;memory&gt;` is expected to be a positive number followed by Ki, Mi, or Gi to indicate the amount of memory.

When not provided, configured server defaults are used.

`&lt;name&gt;`

Unique name of the pool used to reference the pool when creating leases which make use of the pool. Pool names can contain only alphanumeric, hyphen, and underscore characters. Pool names are case insensitive and must not conflict with any existing, active pool. Value is required, maximum allowed size is 512 characters, and minimum size is 8 characters.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Pool Delete

```
tmsctl pool delete (&lt;triton-pool-name&gt;|&lt;triton-pool-name&gt;) [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]
```

An error code will be returned when no default exists and (–target|-t) has not been specified.

Options

`&lt;triton-pool-name&gt;` Unique identifier of the pool. Represented as 32 character UUID.

`&lt;triton-pool-name&gt;` Unique name of the pool.

Pool names can contain only alphanumeric, hyphen, and underscore characters. Pool names are case insensitive. Maximum allowed size is 1024 bytes.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Pool List

`tmsctl pool list [(--verbose|-v)] [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

An error code will be returned when no default exists and (–target|-t) has not been specified.

Options

`--verbose|-v`

Whether to produce more verbose details about the pools.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Pool Status

`tmsctl pool status (&lt;triton-pool-name&gt;|&lt;triton-pool-id&gt;) [(--verbose|-v)]   [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

An error code will be returned when no default exists and (–target|-t) has not been specified.

Options

`--verbose|-v`

Whether to produce more verbose details about the pools.

`&lt;triton-pool-id&gt;`

Unique identifier of the pool. Represented as 32 character UUID.

`&lt;triton-pool-name&gt;`

Unique name of the pool. Pool names can contain only alphanumeric, hyphen, and underscore characters. Pool names are case insensitive. Maximum allowed size is 1024 bytes.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Allowlist

Allowlist Add

`tmsctl allowlist add &lt;image&gt; [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Connects to `&lt;target&gt;` and adds a Triton container image to the Triton allowlist.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`&lt;image&gt;`

Container image to add the list of allowed Triton container images.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Allowlist List

`tmsctl allowlist list [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Connects to `&lt;target&gt;` and lists the Triton container images new leases are allowed to be created with.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Allowlist Remove

`tmsctl allowlist rm &lt;image&gt; [(--target|-t):&lt;target&gt;] [(--porcelain|-z)]`

Connects to `&lt;target&gt;` and removes a Triton container image from the Triton allow-list.

When no default `&lt;target&gt;` exists and (`--target|-t`) has not been specified, an error will occur.

Options

`&lt;image&gt;`

Container image to remove from the list of allowed Triton container images.

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

`--target|-t`

Determines the Triton Management Service to connect to. See Common Options.

Target

Target Add

`tmsctl target add [--force] [--set-default] &lt;name&gt; &lt;url&gt;`

Adds a new `&lt;target&gt;` to the set of configured Triton Management Services.

`&lt;url&gt;` is required to be prefixed with "http://" or "https://".

When `&lt;target&gt;` already exists in the list of configured Triton Management Services, an error will occur unless `--force` is specified.

Options

`--force`

Allows for the replacement of an existing configured Triton Management Service, when specified, with `&lt;target&gt;` and `&lt;url&gt;`.

`--set-default`

Sets as default target for future commands which require a connection to Triton Management Service.

Target List

`tmsctl target list [(--porcelain|-z)]` `tmsctl targets [(--porcelain|-z)]`

Lists all configured Triton Management Server targets.

Options

`--porcelain|-z`

Formats output in an easy-to-parse format for scripts. See Common Options.

Target Remove

`tmsctl target rm [--force] &lt;name&gt;`

Removes `&lt;target&gt;` from the set of configured Triton Management Services.

If the target is the default, it is not removed unless the `--force` flag is used.

Options

`--force`

Removes `&lt;target&gt;` regardless if it has been set to default or not.

Target Set

`tmsctl target set &lt;name&gt;`

`tmsctl target &lt;target&gt;`

Sets `&lt;target&gt;` as the default target for future commands which require a connection to Triton Management Service.

`&lt;target&gt;` must have already been added to list of possible targets.

see `tmsctl target add` for additional details.

Configuring tmsctl

On startup, `tmsctl` will read a configuration if name `.tmsctlconfig` from the user's home directory. This file contains information about any named targets (see the target command) as well as options to configure the output of `tmsctl`.

The format of `.tmsctlconfig` is not guaranteed to be stable and this point and may change in the future. For now, it is a JSON file.

Configuring Output Colors

By default, `tmsctl` outputs everything in the default colors of the console. This can be changed by adding an entry named `"console"` at the top level of the configuration file and setting is `"enable-colors"` property to `"true"`. The example configuration below will tell `tmsctl` to enable colors with its default color scheme:

```
{
"console": {
"enable-colors": "true"
}
}
```

If the default `tmsctl` color scheme does not work well with your preferred terminal settings, you can customize the set of colors that `tmsctl` will use. When colors are enabled, `tmsctl` will read the additional properties from the `"console"` object to control text color:

- `"color"`: used for most output.
- `"emphasis"`: used for lines that add emphasis (e.g. lease IDs in `tmsctl lease create`).
- `"error"`: used for errors

- `"header"`: used for header lines.
- `"understated"`: used for output that can often be ignored.
- `"warning"`: used for warnings.

In addition to the above, some option can have `"-back"` added to it to control the background color of the corresponding entry. The options that support `"-back"` are `"emphasis"`, `"error"`, `"warining"` and `"understated`”.

Allowed values for colors are those listed by the .NET class (ConsoleColor) [https://learn.microsoft.com/en-us/dotnet/api/system.consolecolor]. Colors must be provided in lower case, with words separated by a `-`. For example, to use `ConsoleColor.DarkGreen`, you would specify `"dark-green"` in the configuration file.

Helm Chart Values

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.

The TMS helm-chart contains a `values.yaml` file which contains all of the deployment configuration options available. TMS configuration is broken into three sections:

- `images`: Container image information used by TMS.
  - `server`: Name of the container image containing TMS Server. *(required)*
  - `sidecar`: Name of the container image containing TMS Triton Sidecar. *(required)*
  - `triton`: Name of the container image containing Triton Inference Server. *(required)*
  - `mongodb`: Name of the container image containing MongoDB database used by TMS Server. *(required)*
  - `rest`: Name of the container image containing TMS HTTP API Server. *(optional)*
  - `secrets`: Name(s) of Kubernetes secrets used to pull container image during pod deployment.
- `kubernetes`: Configuration options affecting how TMS deploys objects with Kubernetes. *(optional)*

    - `customAnnotations`: Custom annotations added to the metadata of pods deployed by TMS.

    - `customLabels`: Custom labels added to the metadata of pods deployed by TMS.

    - `partOf`: Name of a higher level application TMS is a part of, applied as label 'app.kubernetes.io/part-of'.

- `server`: Configuration options related to how TMS Server is deployed an operates.

    - `apiService`: Configuration options related to the network services provided by the TMS Server.

        - `port`: Port to use to connect to the gRPC API service when external to the Kubernetes cluster (default: `30345`).

          External ports must be in the range [30000, 32767].

        - `type`: Type of Kubernetes service connection used by the server to provide network API services (default: `ClusterIP`).

          Valid options are ExternalName, ClusterIP, NodePort, and LoadBalancer.

    - `resources`: Defines the computing and memory resources allocated and reserved for the pod hosting the API server and database. If many concurrent requests are expected to the API server from many different clients, it is highly recommended to change these values. A starting suggestion is to dedicate 25% of the resources to the API server, and 75% to the database.

        - `apiServer`: Defines the resources allocated and reserved for the API server's container.

            - `cpu`: The number of CPUs to be allocated and reserved for the API server container (default: 0). If 0, no CPUs will be requested, but a limit of 1 will be set. If it is any other value, that value will be used used as both the request and limit.

        - `memory`: The amount of memory to be allocated and reserved for the API server container (default: 1Gi). Must be a number with memory units (e.g. Mi, Gi).

    - `database`: Defines the resources allocated and reserved for by the database container.

        - `cpu`: The number of CPUs to be allocated and reserved for the database container (default: 1). This will be used as both the request and limit.

        - `memory`: The amount of memory to be allocated and reserved for the database container (default: 2Gi). Must be a number with memory units (e.g. Mi, Gi).

- `lease`: Configuration options for the creation and management of leases.

    - `timeout`: Configures amount of time a lease is allowed to attempt loading before timing out.

    - `duration`: Configuration options related to lease durations.

        - `initial`: Configuration options related to initial requested duration of leases.

            - `default`: Default requested duration of a lease (default: `10m`).

            - `maximum`: Maximum requested duration of a lease (default: `30m`).

        - `renewal`: Configures options related to requested renewal duration of leases.

            - `default`: Default requested renewal duration of a lease (default: `10m`).

            - `maximum`: Maximum requested renewal duration of a lease (default: `30m`).

    - `automaticRenewal`: Configuration options related to automatic renewal of leases. *(optional)*
        - `enabled`: Determines if the service supports automatically renewed leases or not (default: `true`).

          When not enabled, leases will not be allowed to request to be automatically renewed.
        - `window`: Configuration options related to the time since a lease has last been active to be automatically renewed.
            - `default`: Default requested amount of time since a lease has last been active to be automatically renewed (default: `5m`).
            - `maximum`: Maximum requested amount of time since a lease has last been active to be automatically renewed (default: `5m`).
    - `databaseStorage`: Configuration option to define persistent storage for the server's database.
        - `volumeClaimName` : Kubernetes persistent volume claim (pvc) attached to the volume where TMS server's database will be stored.
    - `shareTriton`: Configuration options related to the sharing of Triton Server instances by leases. *(optional)*
        - `enabled`: Determines if the service supports the sharing of Triton Server instances by leases or not. (default: `false`)
        - `byDefault`: Default value applied to lease requests when not specified (default: `false`).
- `modelRepositories`: Configuration options related to model repositories with models available to instances of Triton.
    - `s3`: Model repositories which contain models stored in a S3 bucket.

Access is managed by the ARN specified by `server.security.aws.role`.

- `repositoryName`: Name used to reference this model repository as part of lease acquisition.

  May contain only lowercase alphanumeric characters (without spaces, hyphens `-` are permitted).

- `bucketName`: Name of the S3 bucket used to fetch models.

- `awsRegion`: Region code of the S3 Bucket.

  Must be a valid code designating to existing AWS region (eg. “us-west-2”).

  For additional information, refer to https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

- `endpoint`: Service URL of the S3 bucket.

  When both ‘endpoint’ and ‘awsRegion’ fields are specified, the ‘endpoint’ value will be used instead of the `awsRegion` value.

  Must be a valid URL designating to an existing endpoint (eg. “http:/s3.us-west-2.amazonaws.com” or “http:/play.min.io:9000”).

  For additional information, refer to https://docs.aws.amazon.com/general/latest/gr/s3.html#amazon_s3

- `accessKey`: Name of the Kubernetes secret to read and provide as the access key ID to download objects from the S3 bucket.

  Optional value when IAM or default AWS environment variables are not used for authorizing TMS to read from an S3 bucket.

- `accessSecret`: Name of the Kubernetes secret containing the secret access key to read from the S3 bucket.

        Optional value when IAM or default AWS environment variables are not used for authorizing TMS to read from an S3 bucket.

    - `https`: Model repositories which provide models as compressed archive downloads via web-service using HTTP GET.

        - `secretName`: Name of the Kubernetes secret to read and provide as a Authorization header for download requests.

        - `targetUri`: URL of the remote web-sever in <domain_label_or_ip_address>/<path> format, used to determine if secrets apply to a model request or not.

    - `volumes`: Model repositories which contain models stored in a file-system-like structure.

        - `repositoryName`: Name used to reference this model repository as part of lease acquisition.

          May contain only lowercase alphanumeric characters (without spaces, hyphens `-` are permitted).

        - `volumeClaimName`: Kubernetes persistent volume claim (pvc) used to fetch models.

- `autoscaling`: Configuration options related to autoscaling Triton instances. If this section is missing, autoscaling will be disabled. *(optional)*

    - `enabled`: Determines if TMS Server supports autoscaling Triton instances or not (default: `false`).

      When not enabled, requests for autoscaling leases will not be allowed.

    - `replicas`: Configuration options related to replication of autoscaling Triton instances.

        - `default`: Values used for autoscaling Triton instances when values are not provided during lease acquisition.

        - `maximum`: The maximum number of replicas.Must be within the limits specified in the “limits” section (default: 5).

        - `minimum`: The minimum number of replicas. Must be within the limits specified in the “limits” section (default: 1).

          Must be a positive integer.

    - `limits`: Defines the limits imposed on the number of replicas that may be requested for a lease.

        - `maximum`: The maximum number of replicas (default: 10). Must be a non-negative number greater than or equal to `maximum-idle`.

        - `maximum-idle`: The maximum number of idle instances that are allowed (default: 1). In other words, the maximum value for the minimum number of replicas a user may request. Must be a non-negative number less than or equal to `maximum`.

        - `minimum`: The minimum number of replicas (default: 1). Must be a non-negative number less than or equal to `maximum-idle`.

- `metrics`: Configuration options related to how metrics are used by autoscaling Triton instances to determine availability and scale.

  At least one metric must be enabled when support for autoscaling Triton instances is enabled.

    - `cpuUtilization`: Metric used to determine scaling based on CPU utilization.

        - `allowed`: Determines whether autoscaling based on CPU utilization is allowed (default: `false`).

        - `enabled`: Determines if scaling based on CPU utilization is enabled by default (default: `false`).

    - `threshold`: Threshold, expressed as a percentage, used to determine scaling (default: `90`).

      Must be a positive integer in the exclusive range (0, 100).

        - `default`: Default value used for the threshold, as a percentage (default: `90`).
        - `minimum`: Minimum value for the threshold, as a percentage (default: `50`).
        - `maximum`: Maximum value for the threshold, as a percentage (default: `100`).

- `gpuUtilization`: Metric used to determine scaling based on GPU utilization.
    - `allowed`: Determines whether autoscaling based on GPU utilization is allowed (default: `false`).
    - `enabled`: Determines if scaling based on GPU utilization is enabled by default (default: `false`).
    - `threshold`: Threshold, expressed as a percentage, used to determine scaling (default: `90`).

      Must be a positive integer in the exclusive range (0, 100).

        - `default`: Default value used for the threshold, as a percentage (default: `90`).
        - `minimum`: Minimum value for the threshold, as a percentage (default: `50`).
        - `maximum`: Maximum value for the threshold, as a percentage (default: `100`).

- `queueTime`: Metric used to determine if scaling based on Triton inference-query queue times.
  - `allowed`: Determines whether autoscaling based on Triton inference-query queue times is allowed (default: `false`).
  - `enabled`: Determines if scaling based on Triton inference-query queue times is enabled by default (default: `false`).
  - `threshold`: Threshold, in microseconds, used to determine scaling.
    - `default`: Default value for the threshold, as a time in microseconds (default: `10000`).
    - `minimum`: Minimum value for the threshold, as a time in microseconds (default: `10000`).
    - `maximum`: Maximum value for the threshold, as a time in microseconds, with 0 and negative numbers meaning no limit (default: `0`).
- `queueTimePercentage`: Metric used to determine scaling based on the percentage of the total inference time that requests spend in the queue.
  - `allowed`: Determines whether autoscaling based on the percentage of time spent in the queue is allowed (default: false).
  - `enabled`: Determines whether autoscaling based on the percentage of time spent in the queue is enabled by default (default: false).
  - `threshold`: Threshold, as a percentage, used to determine scaling.
    - `default`: Default value for the threshold, as a percentage (default: 50).

        - `minimum`: Minimum value for the threshold, as a percentage (default: 1).

        - `maximum`: Maximum value for the threshold, as a percentage (default: 100)

  - `metrics`: Configuration options for the collection and reporting of runtime metrics by TMS Server.

    - `verbosity`: Verbosity (volume of total metrics) of metrics collected and reported (default: `0`). *(optional)*

      Must be in the range [0, 3].

    - `reportingWindow`: Period of time from the time of request used when determining metric values reported (default: `60s`).

    - `port`: Port used to connect to the metrics service when external to the Kubernetes cluster (default: `30543`).

      Must be in the range [30000, 32767].

    - `models`: Configuration options controlling model deployment (fetching, loading into Triton, etc.) metrics collection.

      - `verbosity`: Verbosity (volume of total metrics) of metrics collected and reported (default: `0`).

        Must be in the range [0, 3].

      - `reportingWindow`: Frequency which model metrics are pushed from Triton sidecar to TMS Server (default: `15s`).

  - `security`: Configuration options related to Transport Layer Security (TLS) connection encryption and security.

    - `aws`: Configuration options for instances deployed using Amazon EKS.

        - `role`: AWS IAM role used read models S3 buckets configured in `server.modelRepositories.awsS3`.

- `tls`: Configuration options related to Transport Layer Security (TLS) connection encryption.

    - `enabled`: Determines if TLS is expected to be enabled or not (default: `false`).

      When enabled, TMS will provision a certificate issuer as part of its deployment. The issuer will be used to issue TLS certificates for each Triton Inference Server instance deployed by TMS.

    - `certManager`: Configuration options related to cert-manager supplied TLS certificate(s) used to encrypt network traffic.

      TMS manages and applies certificates for TLS based secure communications using cert-manager.

        - `group`: Kubernetes resource group of the CA issuer to use when creating service certificates (default: `cert-manager.io`).

        - `kind`: Kubernetes resource kind of the CA issuer to use when creating service certificates (default: `ClusterIssuer`).

        - `name`: Name of the issuer to use when creating service certificates.

        - `privateKey`: Configuration options related to the creation of certificate private keys.

            - `algorithm`: Algorithm of the private key for the certificate (default: `RSA`).

              Supported values are `RSA` , `ECDSA`, or `Ed25519`.

            - `size`: Size, in bits, of the corresponding private key for the certificate (default: `4096`).

Supported values depend on the value of `algorithm`:

      - RSA: `2048`, `4096` or `8192`
      - ECDSA: `256`, `384` or `521`
      - Ed25519: *(property is ignored)*

  - `traceLevel`: Configures the verbosity of the logging produced by the server. *(optional)*

    TMS will produce logs for Kubernetes to collect via standard output and standard error normally when this value is not provided.

- `triton`: Configuration options related to the deployment of Triton Inference Server.

  Values can be customized based on capacity of your cluster's hardware and expected workload characteristics.

  - `enableRestrictedAccess`: Determines if the Triton API and protocols have restricted access enabled (default: true). This feature relies on Limited Endpoint Access feature in Triton.Since it is a BETA feature, it may result in compatability issues in the future.

  - `resources`: Configuration options related to default and maximum resource requests per Triton instance.

    - `default`: Values used to determine the resources assigned to a Triton instance when not provided during lease acquisition.

      - `cpu`: Number of logical CPU cores to assign to a Triton instance (default: `2`).

        Must be a positive integer.

      - `gpu`: Number of logical GPU devices to assign to a Triton instance (default: `1`).

        Must be a positive integer.

    - `sharedMemory`: Amount of a Triton instance’s memory to reserve for shared-memory (default: `256Mi`).

      Must be a positive integer, followed by a scale suffix of Ki, Mi, or Gi.

    - `systemMemory`: Amount of main memory to assign to a Triton instance (default: `4Gi`).

      Must be a positive integer, followed by a scale suffix of Ki, Mi, or Gi.

- `limits`: Range restrictions on resources allowed to be assigned to a Triton instance.

    - `minimum`: Minimum resources allowed to be assigned to a Triton instance.

        - `cpu`: Number of logical CPU cores to assign to a Triton instance (default: `2`).

          Must be a positive integer.

        - `gpu`: Number of logical GPU devices to assign to a Triton instance (default: `1`).

          Must be a positive integer.

        - `sharedMemory`: Amount of a Triton instance’s memory to reserve for shared-memory (default: `128Mi`).

          Must be a positive integer, followed by a scale suffix of Ki, Mi, or Gi.

        - `systemMemory`: Amount of main memory to assign to a Triton instance (default: `1Gi`).

          Must be a positive integer, followed by a scale suffix of Ki, Mi, or Gi.

- `maximum`: Maximum resources allowed to be assigned to a Triton instance.

  - `cpu`: Number of logical CPU cores to assign to a Triton instance (default: `16`).

    Must be a positive integer.

  - `gpu`: Number of logical GPU devices to assign to a Triton instance (default: `4`).

    Must be a positive integer.

  - `sharedMemory`: Amount of a Triton instance's memory to reserve for shared-memory (default: `2Gi`).

    Must be a positive integer, followed by a scale suffix of Ki, Mi, or Gi.

  - `systemMemory`: Amount of main memory to assign to a Triton instance (default: `32Gi`).

    Must be a positive integer, followed by a scale suffix of Ki, Mi, or Gi.

Release Notes for Triton Management Service

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.

Version 1.4

New Features

- Added the ability to restrict access to Triton Control Operations and allow only the Triton Sidecar to perform such operations. This feature relies on Limited Endpoint Access feature in Triton. Since it is a BETA feature, it may result in compatability issues in the future.

Bug Fixes

- Triton Pools feature can now be disabled by setting `triton.pools.enabled` to `false` in helm `values.yaml`. Previously, even after setting the above value to `false`, the Triton Pools feature remained enabled.

Version 1.3

New Features

- Updated the default Triton image from 23.10 to 23.11.

Bug Fixes

- When a model is deleted from a pooled Triton instance, the files are now deleted. Previously, they remained in the pod consuming space, which would lead to the pod eventually running out of storage.

Version 1.2

New Features

- Updated the default Triton image from 23.09 to 23.10.
- Added the ability to autoscale based on the percentage of total inference request time spent in the queue. This is in addition to the ability to scale on the absolute time spent in the queue. See the autoscaling options of tmsctl for more details.

Bug Fixes

Version 1.1

New Features

- Updated the default Triton image from 23.08 to 23.09. Since 23.09 lowers the memory requirements of Triton, we also lowered the default value for the minimum amount of memory used by Triton pods.
- In order to improve visibility on different console color schemes, the output of `tmsctl` no longer includes colors by default. Colors can be enabled and configured via `tmsctl` configuration options.
- Added a feature to allow the TMS administrator to configure the CPU and memory resources allocated and reserved for the TMS API server and database. See the deployment guide for more information.
- Model repository size for Triton instances can be configured during lease acquisition (initially fixed at 2GB prior to v1.1). See the section under `--triton-resources` in Triton Options and Configuring Triton Containers to learn more.

Bug Fixes

- Configured colors for headers in `tmsctl` output are used properly now.

- More informative message returned by LeaseService when a header is missing.
- `Lease.List()` method now works with the `verbose` option enabled.
- Leases are no longer terminated by the kubernetes startup probe when loading larger models (particularly LLMs).

Version 1.0

New Features

- Support for Generic S3 Model Repositories (including MinIO and Google Cloud Storage). (See updated documentation for S3 Configuration).
- Ability to configure longer timeout for loading larger models in `values.yaml` through `server.lease.timeout`.
- Finalized our TMS v1.0 API with the intent of retaining forward/backward compatibility with all TMS v1.x releases. Be aware that several v0.x interfaces were changed and are incompatible with the v1.0 API. Please make sure to upgrade to the latest `tmsctl` and to upgrade any custom clients to avoid compatibility issues.

Bug Fixes

- Fixed several small bugs in the Triton pool feature, including in the scheduling algorithm.
- Fixed bugs which were causing leases which were cancelled during creation to remain in a pending state. There are still some situations when this may happen, but such leases can now be released via `tmsctl lease release`.
- Fixed a bug which allowed Triton pods to report they were ready before they were fully operational.

Test Environments

This version of TMS was tested with the below versions of different packages. Other versions may work as well, but have not been tested.

- Kubernetes:

    - CNS v9.0 (Kubernetes 1.26 – see the CNS documentation for more details).
    - EKS with Kubernetes 1.26 - 1.27
    - AKS with Kubernetes 1.26
- Triton: 23.03
- Helm: 3.11
- Prometheus:
    - Prometheus: 2.45 (Prometheus community Helm chart version 48.1.1)
    - Prometheus Adapter: 0.10 (Prometheus community Helm chart version 4.2.0)

Version 0.11

New Features

- New configuration options were added in `values.yaml` to allow custom installations of Prometheus to work properly with autoscaling in TMS.
- When configuring queue time thresholds for autoscaling in `values.yaml`, units must now be specified. Previously, the values were specified as a number of microseconds (e.g. 10000). Now, units are required (e.g. “10ms”, “100us”).
- Option to configure TMS to persist data in the case of server failure or restart. (See Configuring Persisted Database)
- New feature: Triton Pools has been added. Provides improved methods for maximizing utilization of Triton Servers deployed by TMS.

  See: Triton Pools & Quota Based Shared Tritons for additional information.
- REST service has been removed.

Fixes

- Fixed a bug which prevented loading models in an S3 bucket which were in nested directories.

Version 0.10

New Features

Triton Recovery

- Triton instances can now recover from failures. If a pod hosting Triton instance is killed, or if the Triton server dies, the pod will be restarted and all the models will be reloaded when the new pod starts.

Lease Names

- Leases will now be able to have custom labels associated with them. These names can be used to interact with a lease in place of the default `"Triton-####"` label for kubernetes services. See the `lease-name` section under tmsctl to learn more.

Fixes

- Fixed a bug which prevented large models from being loaded due to using too much ephemeral storage in the Triton Sidecar containers.

Known Issues

Version 0.9

New Features

- Added the ability to set the minimum number of replicas in autoscaling leases. Administrator may configure thresholds and defaults for this value. Users may control it on a per-lease basis via the gRPC API and tmsctl.

Lease Events

- Triton Management Service will now record events related to the lifecycle of a lease.

  Lease event information will be provided during the creation of a new lease, as well as when requesting the status of a lease.

  Lease events provide a mechanism for TMS to report Triton related, model loading errors.

- Updated the output of `tmsctl lease create`, `tmsctl lease list`, and `tmsctl lease status`.

  - `tmsctl lease create` now includes lease event information. The pretty and porcelain (`-z`) output have both been updated to include event information.

    The pretty print version of the output now reports the status of all models in the lease and attempt to avoid scrolling by resetting the cursor position for every update received from the server.

  - `tmsctl lease status` now includes lease event information.

  - `tmsctl lease list` no longer includes model information. This was removed to reduce load on the server's database when large numbers of leases were being returned.

Support for TLS Encrypted Connections

- Added support for TLS encrypted connections to TMS and Triton Servers.

- With the initial implementation, certificate validation has been disabled for TMS Server, Triton Sidecar, and TMS Control (`tmsctl`). Certificate validation will be enabled in a future version of TMS.

Support for Public S3 Model Repositories without IAM

- Added support for model repositories residing in public S3 buckets.

- This provides an alternative option for accessing S3 models without IAM.

Horizontal Pod Autoscaling on Average Queue Time

- Modified leases to scale on average inference queue time with Triton Server.

- Prior to this change, the queue time metric was processed incorrectly and leases would not scale correctly.

Fixes

- Fixed an issue where the URL of a lease's Triton Server was not provided when reading a lease's status.
- Fixed an issue where the deployment of Triton Servers would fail when services that depend on init-containers were injected into the deployment due the "PodInitializing" container status not being correctly handled.
- Fixed an issue where models in a multi-model lease were loaded at random into Triton Server. This issue resulted in failures for ensemble models, but is now fixed to load models in the ordering from the lease request.
- Fixed an issue where the TMS deployment was failing due to bugs in parsing helm chart values.

Known Issues

- Canceling `tmsctl lease create` by using Ctrl+C to terminate the process, correctly prevents the deployment of the requested lease, but leaves the lease in a `Pending` "zombie" state. Leases caught in this state are metadata artifacts that no impact on the behavior of TMS or any deployed Triton Servers.

Version 0.8

New Features

- Added a `tmsctl lease renew` command to renew leases via `tmsctl`.
- Added options to control lease duration on a per-lease basis. Users may specify the duration via the gRPC API and via tmsctl. Administrators can configure limits for these values.
- Added options to control autoscaling parameters on a per-lease basis. Users may specify the duration via the gRPC API and via tmsctl. Administrators can configure limits for these values.
- Added `tmsctl target set` and `tmsctl target rm` commands.
- Added `tmsctl lease list` command.
- Added official support for Azure Blob and Azure File model repositories.

Known Issues

- Autoscaling on GPU utilization does not function correctly when Triton has been deployed to an Ampere MIG partition. Autoscaling on CPU utilization and queue time does work on such systems.

- TMS Clean-up job can hang when uninstalling a TMS deployment using `helm delete [tms-instance-name]`. This can result in Helm timing out the uninstallation. `helm delete [tms-instance-name] --no-hooks` can be used as a workaround.

  When using the workaround, cluster administrators might have to manually delete abandoned TMS pods, deployments, service, secrets, and/or certificates due to the clean-up job not having been run.

Version 0.7.1

Fixes

Lease Requests with Multiple Models

- Fix has been added that enables users to request Triton leases with multiple models in a single request.

Known Issues

- Autoscaling on GPU utilization does not function correctly when Triton has been deployed to an Ampere MIG partition. Autoscaling on CPU utilization and queue time does work on such systems.

Version 0.7

New Features

Improvements in Autoscaling Leases

- Autoscaling leases now support automatic renewal, just like non-autoscaling leases.

- The metrics that control scaling can be configured by the TMS administrator.

Expanded & Improved Metrics

TMS now reports model loading metrics and over 200 runtime metrics.

Model metrics are related to the loading of models into Triton Inference Server. Any reported model metrics will include the URL used to acquire the model.

Collected metrics have a “visibility score” assigned to them based on their importance and utility. When reporting metrics, TMS will only report the collected metrics that have a “visibility score” equal to or greater than the configured “minimum visibility”.

The minimum visibility value can be changed in the `values.yaml` file of the Helm chart.

Support for AWS S3 Model Repositories

TMS now supports reading models from AWS S3 model repositories. TMS administrators should take a look at the section of the model repository docs for assistance on this.

Known Issues

- Autoscaling on GPU utilization does not function correctly when Triton has been deployed to an Ampere MIG partition. Autoscaling on CPU utilization and queue time does work on such systems.

Version 0.6

New Features

Lease Autoscaling

TMS users can now request that lease automatically scale the number of Triton instances servicing them based on utilization. For full details, see the autoscaling configuration and usage instructions.

Persistent Volume Repositories

Administrators can now attach model repositories in persistent volumes to their TMS instance. To learn more, please refer to the Persistent Volume Claims section in the model repository guide.

Additionally, TMS now supports AWS EBS persistent volumes. Refer to the Persistent Volume Claims section in the model repository guide.

Bug Fixes

Known Issues

Version 0.5

New Features

NFS Model Repositories

TMS administrators can now configure TMS with model repositories hosted on NFS servers which Triton instances can load models from.

Unlike `http` model repositories, NFS hosted repositories provide TMS the benefit of being able to now consume decompressed Triton models in lease requests.

To use an NFS model repository, TMS administrators will have to create a Kubernetes persistent volume with a respective persistent volume claim (in the same namespace as TMS) for the NFS server. The persistent volume claim name should be provided in TMS's helm charts.

There is a guide in the quickstart guide providing more elaborate instructions. Additionally, see the default values under `values.yaml#sidecar.modelRepositories.nfs` to learn more.

Bug Fixes

Known Issues