



INFERENCE SERVER BETA RELEASE

DU-08994-001 _v0.4 | October 2018

User Guide



TABLE OF CONTENTS

Chapter 1. Overview Of The Inference Server	1
1.1. Contents Of The Inference Server Container.....	1
Chapter 2. Pulling The Inference Server Container	2
Chapter 3. Running The Inference Server Container	3
Chapter 4. Verifying The Inference Server	5
Chapter 5. Model Store	6
5.1. Model Versions.....	7
5.2. Model Definition Files.....	7
5.3. Model Configuration Schema.....	8
5.3.1. TensorFlow GraphDef Models.....	9
5.3.2. TensorFlow SavedModel Models.....	10
5.3.3. TensorRT PLAN Models.....	10
5.3.4. Caffe2 NetDef Models.....	11
5.3.5. ONNX Models.....	11
Chapter 6. Inference Server HTTP API	12
6.1. Server Status API.....	12
6.2. Infer.....	13
Chapter 7. Inference Server gRPC API	15
Chapter 8. Support	16

Chapter 1.

OVERVIEW OF THE INFERENCE SERVER

The NVIDIA® TensorRT™ Inference Server provides a cloud inferencing solution optimized for NVIDIA GPUs. The server provides an inference service via an HTTP or gRPC endpoint, allowing remote clients to request inferencing for any model being managed by the server. The Inference Server provides the following features:

Multiple model support

The server can manage any number and mix of models (limited by system disk and memory resources). Supports TensorRT, TensorFlow GraphDef, TensorFlow SavedModel and Caffe2 NetDef model formats. Also supports TensorFlow-TensorRT integrated models.

Multi-GPU support

The server can distribute inferencing across all system GPUs.

Multi-tenancy support

Multiple models (or multiple instances of the same model) can run simultaneously on the same GPU.

Batching support

The Inference Server itself is provided as a pre-built container. External to the server, API schemas, C++ and Python client libraries and examples, and related documentation are provided in source at: [GitHub Inference Server](#).

1.1. Contents Of The Inference Server Container

This image contains the inference server in `/opt/inference_server`. The executable is `/opt/inference_server/bin/inference_server`.

Chapter 2.

PULLING THE INFERENCE SERVER CONTAINER

You can pull (download) an NVIDIA container that is already built, tested, tuned, and ready to run. Each NVIDIA deep learning container includes the code required to build the framework so that you can make changes to the internals. The containers do not contain sample data-sets or sample model definitions unless they are included with the source for the framework.

Currently, you can access NVIDIA GPU accelerated containers in one of two ways depending upon where you doing your training. If you own a DGX-1™ or a DGX Station™, then you should use the NVIDIA® DGX™ container registry located at <https://compute.nvidia.com>. You can pull the containers from there and you can also push containers there into your own account on the nvidia-docker repository, nvcr.io.

If you are accessing the NVIDIA containers from a Cloud Server Provider such as Amazon Web Services (AWS), then you should first create an account at the NGC located at <https://ngc.nvidia.com>. After you create an account, the commands to use containers are the same for the DGX-1 and the DGX Station. However, currently, you cannot save any containers to the NVIDIA® GPU Cloud™ (NGC) container registry, nvcr.io if you are using NVIDIA® GPU Cloud™ (NGC). Instead you have to save the containers to your own Docker repository.



The containers are exactly the same, whether you pull them from the NVIDIA DGX container registry or the NGC container registry.

Before you can pull a container you must have Docker and nvidia-docker installed as explained in [Preparing to use NVIDIA Containers Getting Started Guide](#). You must also have access and logged into the NGC container registry as explained in [NGC Getting Started Guide](#).

For step-by-step instructions, see [Container User Guide](#).

Chapter 3.

RUNNING THE INFERENCE SERVER CONTAINER

Before running the Inference Server, you must first set up a model store containing the models that the server will make available for inferencing. The [Model Store](#), describes how to create a model store. For this example, assume the model store is created on the host system directory `/path/to/model/store`. The following command will launch the inference server using that model store.

```
$ nvidia-docker run --rm --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 -p8000:8000 -p8001:8001 -v/path/to/model/store:/tmp/models inference_server:18.xx-py<x> /opt/inference_server/bin/inference_server --model-store=/tmp/models
```

Where `inference_server:18.xx-py<x>` is the container that was pulled from the NVIDIA DGX or NGC container registry as described in [Pulling The Inference Server Container](#).

The `nvidia-docker -v` option maps `/path/to/model/store` on the host into the container at `/tmp/models`, and the `--model-store` option to the Inference Server is used to point to `/tmp/models` as the model store.

The Inference Server listens for:

- ▶ HTTP requests on port 8000
- ▶ gRPC requests on port 8001

and the above command uses the `-p` flag to map container ports 8000, 8001 to host ports 8000, 8001. A different host port can be used by modifying the `-p` flag, for example `-p9000:8000` will cause the Inference Server HTTP endpoint to be available on host port 9000.

The `--shm-size` and `--ulimit` flags are recommended to improve Inference Server performance. For `--shm-size` the minimum recommended size is 1g but larger sizes may be necessary depending on the number and size of models being served.

After starting, the Inference Server will log initialization information to the console. Initialization is complete and the server is ready to accept requests after the console shows the following:

```
Starting server 'inference:0' listening on http://localhost:8000
```

Chapter 4.

VERIFYING THE INFERENCE SERVER

The simplest way to verify that the Inference Server is running correctly is to use the Server Status API to query the server's status. For more information about the Inference Server API, see [Inference Server HTTP API](#). From the host system use `curl` to the HTTP endpoint to request server status. The response is protobuf text showing the status for the server and for each model being served, for example:

```
$ curl localhost:8000/api/status
id: "inference:0"
version: "1.1.0"
uptime_ns: 23322988571
model_status {
  key: "resnet50_netdef"
  value {
    config {
      name: "resnet50_netdef"
      platform: "caffe2_netdef"
    }
    ...
    version_status {
      key: 1
      value {
        ready_state: MODEL_READY
      }
    }
  }
}
ready_state: SERVER_READY
```

This status shows configuration information as well as indicating that version 1 of the `resnet50_netdef` model is `MODEL_READY`, indicating the Inference Server is ready to accept inferencing requests for version 1 of that model. A model version `ready_state` will show up as `MODEL_UNAVAILABLE` if the model failed to load for some reason or if it was unloaded due to the [model version policy](#) discussed here.

Chapter 5.

MODEL STORE

The Inference Server accesses models from a locally accessible file path. This path is specified when the server is started using the `--model-store` option. The model store must be organized as follows:

```
<model-store path>/
  model_0/
    config.pbtxt
    output0_labels.txt
    1/
      model.plan
    2/
      model.plan
  model_1/
    config.pbtxt
    output0_labels.txt
    output1_labels.txt
    0/
      model.graphdef
    7/
      model.graphdef
  model_2/
    ...
  model_n/
```

Any number of models may be specified, however, after the Inference Server is started, models cannot be added to or removed from the model store. To add or remove a model:

1. Stop the Inference Server.
2. Update the model store.
3. Start the Inference Server.

The name of the model directory (for example, `model_0`, `model_1`) must match the name of the model specified in the required configuration file, `config.pbtxt`. This model name is used in the client and server APIs to identify the model. Each model directory must have at least one numeric subdirectory (for example, `model_0/1`). Each of these subdirectories holds a version of the model with the version number corresponding to the directory name. Version subdirectories can be added and removed from the model store while the Inference Server is running.

For more information about how the model versions are handled by the server, see [Model Versions](#). Within each version subdirectory, there is one or more model definition files. For more information about the model definition files contained in each version subdirectory, see [Model Definition Files](#).

The configuration file, `config.pbtxt`, for each model must be **protobuf** text adhering to the **ModelConfig** schema defined and explained below. The `*_labels.txt` files are optional and are used to provide labels for outputs that represent classifications.

5.1. Model Versions

Each model can have one or more versions available in the model store. Each version is stored in its own, numerically named, subdirectory where the name of the subdirectory corresponds to the version number of the model. Version subdirectories can be added and removed while the Inference Server is running to add and remove the corresponding model versions. Each model specifies a *version policy* that controls which of the versions in the model store are made available by the Inference Server at any given time. The **ModelVersionPolicy** portion, as described in [Model Configuration Schema](#) specifies one of the following policies.

All

All versions of the model that are specified in the model store are available for inferencing.

Latest

Only the latest `n` versions of the model specified in the model store are available for inferencing. The latest versions of the model are the numerically greatest version numbers.

Specific

The specifically listed versions of the model are available for inferencing.

If no version policy is specified, then **Latest** (with `num_version = 1`) is used as the default, indicating that only the most recent version of the model is made available by the Inference Server. In all cases, the addition or removal of version subdirectories from the model store can change which model version is used on subsequent inference requests.

5.2. Model Definition Files

Each model version subdirectory must contain at least one model definition file. By default, the name of this file must be:

- ▶ `model.plan` for TensorRT models
- ▶ `model.graphdef` for TensorFlow GraphDef models
- ▶ `model.savedmodel` for TensorFlow SavedModel models
- ▶ `model.netdef` / `init_model.netdef` for Caffe2 Netdef models

The default can be overridden using the `default_model_filename` property in [Model Configuration Schema](#).

Optionally, a model can provide multiple model definition files, each targeted at a GPU with a different [Compute Capability](#). Most commonly, this feature is needed for TensorRT and TensorFlow to TensorRT integrated models where the model definition is valid for only a single compute capability. See the `trt_mnist` configuration in [Model Configuration Schema](#) for an example.

An example model store is available at [Deep Learning Inference Server Clients](#).

5.3. Model Configuration Schema

Each model in the model store must include a file called `config.pbtxt` that contains the configuration information for the model. The model configuration must be specified as protobuf text using the ModelConfig schema described at [GitHub: Inference Server model_config.proto](#).

The following example configuration file is for a TensorRT MNIST model that accepts a single “data” input tensor of shape [1,28,28] and produces a single “prob” output vector. The output vector is a classification and the labels associated with each class are in `mnist_labels.txt`. The Inference Server will run two instances of this model on GPU 0 so that two `trt_mnist` inference requests can be handled simultaneously. Batch sizes up to 8 will be accepted by the server. Two model definition files are provided for each version of this model, one for compute capability 6.1 called `model6_1.plan` and another for compute capability 7.0 called `model7_0.plan`.

```
name: "trt_mnist"
platform: "tensorrt_plan"
max_batch_size: 8
input [
  {
    name: "data"
    data_type: TYPE_FP32
    format: FORMAT_NCHW
    dims: [ 1, 28, 28 ]
  }
]
output [
  {
    name: "prob"
    data_type: TYPE_FP32
    dims: [ 10, 1, 1 ]
    label_filename: "mnist_labels.txt"
  }
]
cc_model_filenames [
  {
    key: "6.1"
    value: "model6_1.plan"
  },
  {
    key: "7.0"
    value: "model7_0.plan"
  }
]
instance_group [
  {
    count: 2
    gpus: [ 0 ]
  }
]
```

```
}]
```

The next example configuration file is for a TensorFlow ResNet-50 GraphDef model that accepts a single input tensor named “input” in HWC format with shape [224,224,3] and produces a single output vector named “output”. The Inference Server will run two instances of this model, one on GPU 0 and one on GPU 1. Batch sizes up to 128 will be accepted by the server.

```
name: "resnet50"
platform: "tensorflow_graphdef"
max_batch_size: 128
input [
  {
    name: "input"
    data_type: TYPE_FP32
    format: FORMAT_NHWC
    dims: [ 224, 224, 3 ]
  }
]
output [
  {
    name: "output"
    data_type: TYPE_FP32
    dims: [ 1000 ]
  }
]
instance_group [
  {
    gpus: [ 0 ]
  },
  {
    gpus: [ 1 ]
  }
]
```

5.3.1. TensorFlow GraphDef Models

The configuration platform value for TensorFlow GraphDef models must be `tensorflow_graphdef` and the model definition file must be named `model.graphdef` (unless the `default_model_filename` property is set in the model configuration).

TensorFlow 1.7 and later integrates TensorRT to enable TensorFlow models to benefit from the inference optimizations provided by TensorRT. Because the Inference Server supports GraphDef models that have been optimized with TensorRT, it can serve those models just like any other TensorFlow model. The Inference Server's TensorRT version (available in the [Inference Server Container Release Notes](#) must match the TensorRT version that was used when the GraphDef model was created.

The following example configuration file is for a TensorFlow ResNet-50 GraphDef model that accepts a single input tensor named `input` in `HWC` format with shape `[224, 224, 3]` and produces a single output vector named `output`. The Inference Server will run two instances of this model, one on GPU 0 and one on GPU 1. Batch sizes up to 128 will be accepted by the server.

```
name: "resnet50"
platform: "tensorflow_graphdef"
max_batch_size: 128
```

```

input [
  {
    name: "input"
    data_type: TYPE_FP32
    format: FORMAT_NHWC
    dims: [ 224, 224, 3 ]
  }
]
output [
  {
    name: "output"
    data_type: TYPE_FP32
    dims: [ 1000 ]
  }
]
instance_group [
  {
    count: 1
    gpus: [ 0, 1 ]
  }
]

```

5.3.2. TensorFlow SavedModel Models

The configuration platform value for TensorFlow SavedModel models must be `tensorflow_savedmodel` and the saved-model directory must be named `model.savedmodel` (unless the `default_model_filename` property is set in the model configuration).

TensorFlow 1.7 and later integrates TensorRT to enable TensorFlow models to benefit from the inference optimizations provided by TensorRT. Because the Inference Server supports SavedModel models that have been optimized with TensorRT, it can serve those models just like any other TensorFlow model. The Inference Server's TensorRT version (available in the Inference Server Container Release Notes (<https://docs.nvidia.com/deeplearning/sdk/inference-release-notes/index.html>)) must match the TensorRT version that was used when the SavedModel model was created.

5.3.3. TensorRT PLAN Models

The configuration platform value for TensorRT PLAN models must be `tensorrt_plan` and the model definition file must be named `model.plan` (unless the `default_model_filename` property is set in the model configuration).

The following example configuration file is for a TensorRT MNIST model that accepts a single `data` input tensor of shape `[1, 28, 28]` and produces a single `prob` output vector. The output vector is a classification and the labels associated with each class are in `mnist_labels.txt`. The Inference Server will run two instances of this model on GPU 0 so that two `trt_mnist` inference requests can be handled simultaneously. Batch sizes up to 8 will be accepted by the server. Two model definition files are provided for each version of this model, one for compute capability 6.1 called `model16_1.plan` and another for compute capability 7.0 called `model17_0.plan`.

```

name: "trt_mnist"
platform: "tensorrt_plan"
max_batch_size: 8
input [

```

```

{
  name: "data"
  data_type: TYPE_FP32
  format: FORMAT_NCHW
  dims: [ 1, 28, 28 ]
}
]
output [
  {
    name: "prob"
    data_type: TYPE_FP32
    dims: [ 10, 1, 1 ]
    label_filename: "mnist_labels.txt"
  }
]
cc_model_filenames [
  {
    key: "6.1"
    value: "model6_1.plan"
  },
  {
    key: "7.0"
    value: "model7_0.plan"
  }
]
instance_group [
  {
    count: 2
    gpus: [ 0 ]
  }
]

```

The model store for this model would look like:

```

model_store/
  trt_mnist/
    config.pbtxt
    mnist_labels.txt
  1/
    model6_1.plan
    model7_0.plan

```

5.3.4. Caffe2 NetDef Models

The configuration platform value for Caffe2 NetDef models must be `caffe2_netdef`. NetDef model definition is split across two files: the initialization network and the predict network. These files must be named `init_model.netdef` and `model.netdef` (unless the `default_model_filename` property is set in the model configuration).

5.3.5. ONNX Models

The Inference Server cannot directly perform inferencing using [ONNX](#) models. An ONNX model must be converted to either TensorRT PLAN or Caffe2 NetDef to be served by the Inference Server. To convert your ONNX model to a TensorRT PLAN use either the [ONNX Parser](#) included in TensorRT or the open-source [TensorRT backend for ONNX](#). Another option is to convert your ONNX model to Caffe2 NetDef [as described here](#).

Chapter 6.

INFERENCE SERVER HTTP API

The Inference Server can be accessed directly using two exposed HTTP endpoints:

`/api/status`

The server status API for getting information about the server and about the models being served.

`/api/infer`

The inference API that accepts model inputs, runs inference and returns the requested outputs.

The HTTP endpoints can be used directly as described in this section, but for most use-cases, the preferred way to access the Inference Server is via the C++ and Python client API libraries. The libraries are available at [GitHub: Inference Server](#).

6.1. Server Status API

Performing an **HTTP GET** to `/api/status` returns status information about the server and all the models being served. Performing an **HTTP GET** to `/api/status/<model name>` returns information about the server and the single model specified by `<model name>`. An example is shown in [Verifying The Inference Server](#).

The server status is returned in the HTTP response body in either text format (the default) or in binary format if query parameter **format=binary** is specified (for example, `/api/status?format=binary`). The status schema is defined by the protobuf schema given in `server_status.proto` defined at [GitHub: Inference Server server_status.proto](#).

The success or failure of the server status request is indicated in the HTTP response code and the **NV-Status** response header. The **NV-Status** response header returns a text protobuf formatted status following the `status.proto` schema defined at [GitHub: Inference Server status.proto](#). If the request is successful the HTTP status is 200 and the **NV-Status** response header will indicate no failure:

```
NV-Status: code: SUCCESS
```

If the server status request fails, the response body will be empty, a non-200 HTTP status will be returned and the **NV-Status** header will indicate the failure reason, for example:

```
NV-Status: code: NOT_FOUND msg: "no status available for unknown model \'x\'"
```

6.2. Infer

Performing an HTTP POST to `/api/infer/<model name>` performs inference using the latest available version of `<model name>` model. The latest available version is the numerically greatest version number. Performing an HTTP POST to `/api/infer/<model name>/<model version>` performs inference using a specific version of the model.

In either case, the request uses the `NV-InferRequest` header to communicate an `InferRequestHeader` protobuf message that describes the input tensors and the requested output tensors as defined at [GitHub: Inference Server api.proto](#). For example, for the ResNet-50 example shown in [Model Configuration Schema](#) the following `NV-InferRequest` header indicates that a batch-size 1 request is being made with input size of `602112 bytes (3 * 224 * 224 * sizeof(FP32))`, and that the result of the “output” tensor should be returned as the top-3 classification values.

```
NV-InferRequest: batch_size: 1 input { name: "input" byte_size: 602112 } output
  { name: "output" byte_size: 4000 cls { count: 3 } }
```

The input tensor values are communicated in the body of the HTTP POST request as raw binary in the order as the inputs are listed in the request header.

The inference results are returned in the body of the HTTP response to the POST request. For outputs where full result tensors were requested, the result values are communicated in the body of the response in the order as the outputs are listed in the request header. After those, an `InferResponseHeader` message is appended to the response body. The `InferResponseHeader` message is returned in either text format (the default) or in binary format if query parameter `format=binary` is specified (for example, `/api/infer/foo?format=binary`).

For example, assuming outputs specified in the `InferRequestHeader` in order are `output0`, `output1`, ..., `outputn`, the response body would contain:

```
<raw binary tensor values for output0, if raw output was requested for output0>
<raw binary tensor values for output1, if raw output was requested for output1>
...
<raw binary tensor values for outputn, if raw output was requested for outputn>
<text or binary encoded InferResponseHeader proto>
```

The success or failure of the inference request is indicated in the HTTP response code and the `NV-Status` response header. The `NV-Status` response header returns a text protobuf formatted status following the `status.proto` schema. If the request is successful the HTTP status is 200 and the `NV-Status` response header will indicate no failure:

```
NV-Status: code: SUCCESS
```

If the inference request fails, a non-200 HTTP status will be returned and the `NV-Status` header will indicate the failure reason, for example:

```
NV-Status: code: NOT_FOUND msg: "no status available for unknown model \'x\'"
```


Chapter 7.

INFERENCE SERVER GRPC API

The Inference Server can be accessed directly using gRPC endpoints defined at [grpc_service.proto](#).

- ▶ **GRPCServer.Status:** The server status API for getting information about the server and about the models being served.
- ▶ **GRPCServer.Infer:** The inference API that accepts model inputs, runs inference and returns the requested outputs.

The gRPC endpoints can be used via the gRPC generated client (demonstrated in the image classification example at [grpc_image_client.py](#) or via the C++ and Python client API libraries. Build instructions for the gRPC client libraries are available at [Deep Learning Inference Server clients](#).

Chapter 8. SUPPORT

For questions and feature requests, use the [Inference Server Devtalk forum](#).

Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, cuFFT, cuSPARSE, DALI, DIGITS, DGX, DGX-1, Jetson, Kepler, NVIDIA Maxwell, NCCL, NVLink, Pascal, Tegra, TensorRT, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2018 NVIDIA Corporation. All rights reserved.