

# Chapter 9. Advanced Model Deployments with TensorFlow Serving

In the previous chapter, we discussed the efficient deployment of TensorFlow or Keras models with TensorFlow Serving. With the knowledge of a basic model deployment and TensorFlow Serving configuration, we now introduce advanced use cases of machine learning model deployments in this chapter. The use cases touch a variety of topics, for example, deploying model A/B testing, optimizing models for deployment and scaling, and monitoring model deployments. If you haven't had the chance to review the previous chapter, we recommend doing so because it provides the fundamentals for this chapter.

## Decoupling Deployment Cycles

The basic deployments shown in [Chapter 8](#) work well, but they have one restriction: the trained and validated model needs to be either included in the deployment container image during the build step or mounted into the container during the container runtime, as we discussed in the previous chapter. Both options require either knowledge of DevOps processes (e.g., updating Docker container images) or coordination between the data science and DevOps teams during the deployment phase of a new model version.

As we briefly mentioned in [Chapter 8](#), TensorFlow Serving can load models from remote storage drives (e.g., AWS S3 or GCP Storage buckets). The standard loader policy of TensorFlow Serving frequently polls the model storage location, unloads the previously loaded model, and loads a newer model upon detection. Due to this behavior, we only need to deploy our model serving container once, and it continuously updates the model versions once they become available in the storage folder location.

## Workflow Overview

Before we take a closer look at how to configure TensorFlow Serving to load models from remote storage locations, let's take a look at our proposed workflow.

[Figure 9-1](#) shows the separation of workflows. The model serving container is deployed once. Data scientists can upload new versions of models to storage buckets either through the buckets' web interface or through command-line copy operations. Any changes to model versions will be discovered by the serving instances. A new build of the model server container or a redeployment of the container is not necessary.

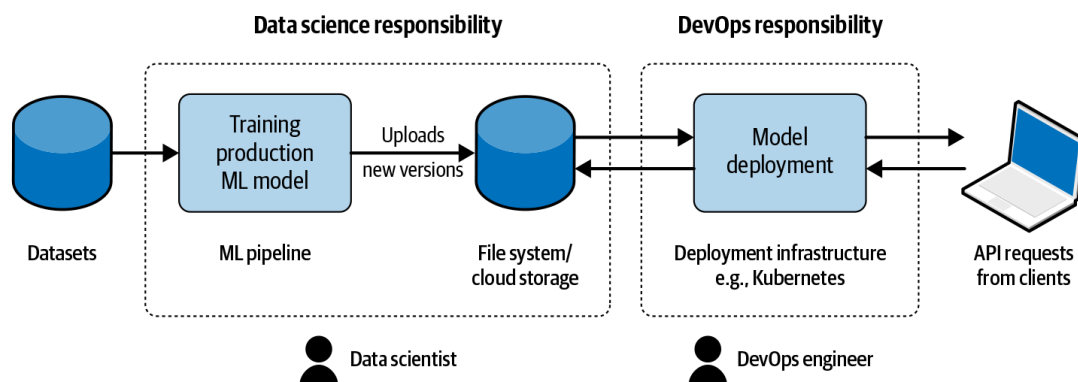


Figure 9-1. Split of the data science and DevOps deployment cycles

If your bucket folders are publicly accessible, you can serve the remote models by simply updating the model base path to the remote path:

```
docker run -p 8500:8500 \
  -p 8501:8501 \
  -e MODEL_BASE_PATH=s3://bucketname/model_path/ \ ❶
  -e MODEL_NAME=my_model \ ❷
  -t tensorflow/serving
```

- ❶ Remote bucket path
- ❷ Remaining configuration remains the same

If your models are stored in private cloud buckets, you need to configure TensorFlow Serving a bit more to provide access credentials. The setup is provider specific. We will cover two provider examples in this chapter: AWS and GCP.

## Accessing private models from AWS S3

AWS authenticates users through a user-specific access key and access secret. To access private AWS S3 buckets, you need to create a user access key and secret.<sup>1</sup>

You can provide the AWS access key and secret as environment variables to the `docker run` command. This allows TensorFlow Serving to pick up the credentials and access private buckets:

```
docker run -p 8500:8500 \
  -p 8501:8501 \
  -e MODEL_BASE_PATH=s3://bucketname/model_path/ \
  -e MODEL_NAME=my_model \
  -e AWS_ACCESS_KEY_ID=XXXXX \ ❶
  -e AWS_SECRET_ACCESS_KEY=XXXXX \
  -t tensorflow/serving
```

❶ The name of the environment variables is important.

TensorFlow Serving relies on the standard AWS environment variables and its default values. You can overwrite the default values (e.g., if your bucket isn't located in the `us-east-1` region or if you want to change the S3 endpoint).

You have the following configuration options:

- `AWS_REGION=us-east-1`
- `S3_ENDPOINT=s3.us-east-1.amazonaws.com`
- `S3_USE_HTTPS=1`
- `S3_VERIFY_SSL=1`

The configuration options can be added as environment variables or added to the `docker run` command as shown in the following example:

```
docker run -p 8500:8500 \
  -p 8501:8501 \
  -e MODEL_BASE_PATH=s3://bucketname/model_path/ \
  -e MODEL_NAME=my_model \
  -e AWS_ACCESS_KEY_ID=XXXXX \
```

```
-e AWS_SECRET_ACCESS_KEY=XXXXX \
-e AWS_REGION=us-west-1 \ ❶
-t tensorflow/serving
```

- ❶ Additional configurations can be added through environment variables.

With these few additional environment variables provided to TensorFlow Serving, you are now able to load models from remote AWS S3 buckets.

## Accessing private models from GCP Buckets

GCP authenticates users through *service accounts*. To access private GCP Storage buckets, you need to create a service account file.<sup>2</sup>

Unlike in the case of AWS, we can't simply provide the credential as an environment variable since the GCP authentication expects a JSON file with the service account credentials. In the GCP case, we need to mount a folder on the host machine containing the credentials inside a Docker container and then define an environment variable to point TensorFlow Serving to the correct credential file.

For the following example, we assume that you have saved your newly created service account credential file under `/home/your_username/.credentials/` on your host machine. We downloaded the service account credentials from GCP and saved the file as `sa-credentials.json`. You can give the credential file any name, but you need to update the environmental variable `GOOGLE_APPLICATION_CREDENTIALS` with the full path inside of the Docker container:

```
docker run -p 8500:8500 \
-p 8501:8501 \
-e MODEL_BASE_PATH=gcp://bucketname/model_path/ \
-e MODEL_NAME=my_model \
-v /home/your_username/.credentials/:/credentials/ ❶
-e GOOGLE_APPLICATION_CREDENTIALS=/credentials/sa-credentials.json \
-t tensorflow/serving
```

- ❶ Mount host directory with credentials.
- ❷ Specify path inside of the container.

With a couple steps, you have configured a remote GCP bucket as a storage location.

## Optimization of Remote Model Loading

By default, TensorFlow Serving polls any model folder every two seconds for updated model versions, regardless of whether the model is stored in a local or remote location. If your model is stored in a remote location, the polling operation generates a bucket list view through your cloud provider. If you continuously update your model versions, your bucket might contain a large number of files. This results in large list-view messages and therefore consumes a small, but over time not insignificant, amount of traffic. Your cloud provider will most likely charge for the network traffic generated by these list operations. To avoid billing surprises, we recommend reducing the polling frequency to every 120 seconds, which still provides you up to 30 potential updates per hour but generates 60 times less traffic:

```
docker run -p 8500:8500 \  
    ...  
    -t tensorflow/serving \  
    --file_system_poll_wait_seconds=120
```

TensorFlow Serving arguments need to be added after the image specification of the `docker run` command. You can specify any polling wait time greater than one second. If you set the wait time to zero, TensorFlow Serving will not attempt to refresh the loaded model.

## Model Optimizations for Deployments

With the increasing size of machine learning models, model optimization becomes more important to efficient deployments. Model quantization allows you to reduce the computation complexity of a model by reducing

the precision of the weight's representation. Model pruning allows you to implicitly remove unnecessary weights by zeroing them out of your model network. And model distillation will force a smaller neural network to learn the objectives of a larger neural network.

All three optimization methods aim for smaller models that allow faster model inferences. In the following sections, we will explain the three optimization options further.

## Quantization

The weights of a neural network are often stored as float 32-bit data types (or, as the IEEE 754 standard calls it, single-precision binary floating-point format). A floating-point number is stored as the following: 1 bit storing the sign of the number, 8 bits for the exponent, and 23 bits for the precision of the floating number.

The network weights, however, can be expressed in `bfloat16` floating-point format or as 8-bit integers. As shown in [Figure 9-2](#), we still need 1 bit to store the sign of the number. The exponent is also still represented through 8 bits when we store weights as `bfloat16` floating points because it is used by TensorFlow. However, the fraction representation is reduced from 23 bits to 7 bits. The weights can even sometimes be represented as integers using only 8 bits.

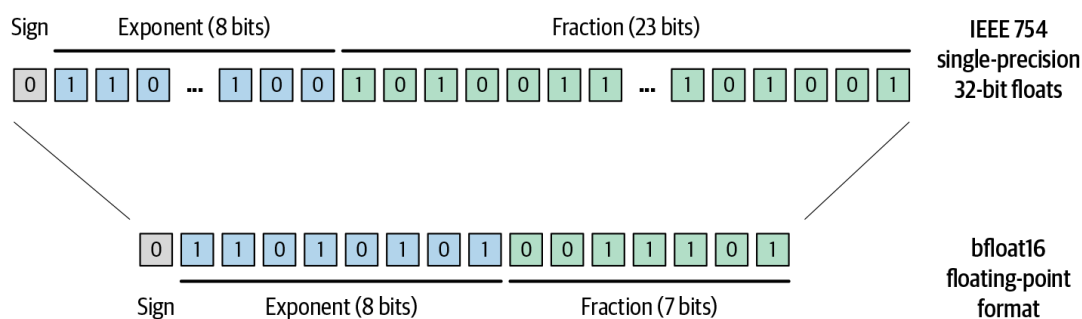


Figure 9-2. Reduction of the floating precision

By changing the network's weight representation to 16-bit floating points or integers, we can achieve the following benefits:

- The weights can be represented with fewer bytes, requiring less memory during the model inference.

- Due to the weights' reduced representation, predictions can be inferred faster.
- The quantization allows the execution of neural networks on 16-bit, or even 8-bit, embedded systems.

Current workflows for model quantization are applied after model training and are often called *post-training quantization*. Since a quantized model can be underfitted due to the lack of precision, we highly recommend analyzing and validating any model after quantization and before deployment. As an example of model quantizations, we discuss Nvidia's TensorRT library (see [“Using TensorRT with TensorFlow Serving”](#)) and TensorFlow's TFLite library (see [“TFLite”](#)).

## Pruning

An alternative to reducing the precision of network weights is *model pruning*. The idea here is that a trained network can be reduced to a smaller network by removing unnecessary weights. In practice, this means that “unnecessary” weights are set to zero. By setting unnecessary weights to zero, the inference or prediction can be sped up. Also, the pruned models can be compressed to smaller models sizes since sparse weights lead to higher compression rates.

---

### HOW TO PRUNE MODELS

Models can be pruned during their training phase through tools like TensorFlow's model optimization package `tensorflow-model-optimization`.<sup>3</sup>

---

## Distillation

Instead of reducing the network connections, we can also train a smaller, less complex neural network to learn trained tasks from a much more extensive network. This approach is called *distillation*. Instead of simply training a smaller machine learning model with the same objective as the bigger model, the predictions of the bigger model (the teacher neural network) influence the update of the smaller model's (the student neural network) weights, as shown in [Figure 9-3](#). By using the predictions from the teacher and student neural networks, the student network can be *forced*

to learn an objective from the teacher neural network. Ultimately, we can express the same model objective with fewer weights and with a model architecture that wouldn't have been able to learn the objective without the teacher forcing it.

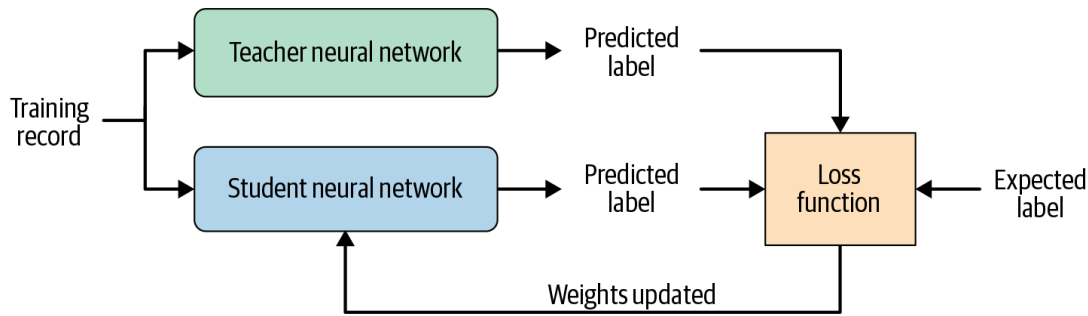


Figure 9-3. Student network learning from a teacher network

## Using TensorRT with TensorFlow Serving

One option for performing quantization on a trained TensorFlow model before deploying it to production is converting the model with Nvidia's TensorRT.

If you are running computationally intensive deep learning models on an Nvidia GPU, you can use this additional way of optimizing your model server. Nvidia provides a library called TensorRT that optimizes the inference of deep learning models by reducing the precision of the numerical representations of the network weights and biases. TensorRT supports int8 and float16 representations. The reduced precision will lower the inference latency of the model.

After your model is trained, you need to optimize the model with TensorRT's own optimizer or with `saved_model_cli`.<sup>4</sup> The optimized model can then be loaded into TensorFlow Serving. At the time of writing this chapter, TensorRT was limited to certain Nvidia products, including Tesla V100 and P4.

First, we'll convert our deep learning model with `saved_model_cli`:



```
$ saved_model_cli convert --dir saved_models/ \
                          --output_dir trt-savedmodel/ \
                          --tag_set serve tensorrt
```

After the conversion, you can load the model in our GPU setup of TensorFlow Serving as follows:

```
$ docker run --runtime=nvidia \
  -p 8500:8500 \
  -p 8501:8501 \
  --mount type=bind,source=/path/to/models,target=/models/my_model \
  -e MODEL_NAME=my_model \
  -t tensorflow/serving:latest-gpu
```

If you are inferring your models on Nvidia GPUs, the hardware is supported by TensorRT. Switching to TensorRT can be an excellent way to lower your inference latencies further.

## TFLite

If you want to optimize your machine learning model but you're not running Nvidia GPUs, you can use TFLite to perform optimizations on your machine learning.

TFLite has traditionally been used to convert machine learning models to smaller model sizes for deployment to mobile or IoT devices. However, these models can also be used with TensorFlow Serving. So instead of deploying a machine learning model to an edge device, you can deploy a machine learning model with TensorFlow Serving that will have a low inference latency and a smaller memory footprint.

While optimizing with TFLite looks very promising, there are a few caveats: at the time of writing this section, the TensorFlow Serving support for TFLite models is only in experimental stages. And furthermore, not all TensorFlow operations can be converted to TFLite instructions. However, the number of supported operations is continuously growing.

## Steps to Optimize Your Model with TFLite

TFLite can also be used to optimize TensorFlow and Keras models. The library provides a variety of optimization options and tools. You can either convert your model through command-line tools or through the Python library.

The starting point is always a trained and exported model in the `SavedModel` format. In the following example, we focus on Python instructions. The conversion process consists of four steps:

1. Loading the exported saved model
2. Defining your optimization goals
3. Converting the model
4. Saving the optimized model as a TFLite model

```
import tensorflow as tf

saved_model_dir = "path_to_saved_model"
converter = tf.lite.TFLiteConverter.from_saved_model(
    saved_model_dir)

converter.optimizations = [
    tf.lite.Optimize.DEFAULT ❶
]
tflite_model = converter.convert()

with open("/tmp/model.tflite", "wb") as f:
    f.write(tflite_model)
```

- ❶ Set the optimization strategy.

TFLite provides predefined optimization objectives. By changing the optimization goal, the converter will optimize the models differently. A few options are `DEFAULT`, `OPTIMIZE_FOR_LATENCY`, and `OPTIMIZE_FOR_SIZE`.

In the `DEFAULT` mode, your model will be optimized for latency and size, whereas the other two options prefer one option over the other. You can set the convert options as follows:

```
...
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
converter.target_spec.supported_types = [tf.lite.constants.FLOAT16]
tflite_model = converter.convert()
...
```

---

If your model includes a TensorFlow operation that is not supported by TFLite at the time of exporting your model, the conversion step will fail with an error message. You can enable an additional set of selected TensorFlow operations to be available for the conversion process. However, this will increase the size of your TFLite model by ca. 30 MB. The following code snippet shows how to enable the additional TensorFlow operations before the converter is executed:

```
...
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS,
                                       tf.lite.OpsSet.SELECT_TF_OPS]
tflite_model = converter.convert()
...
```

If the conversion of your model still fails due to an unsupported TensorFlow operation, you can bring it to the attention of the TensorFlow community. The community is actively increasing the number of operations supported by TFLite and welcomes suggestions for future operations to be included in TFLite. TensorFlow ops can be nominated via the [TFLite Op Request form](#).

## Serving TFLite Models with TensorFlow Serving

The latest TensorFlow Serving versions can read TFLite models without any major configuration change. You only need to start TensorFlow Serving with the enabled `use_tflite_model` flag and it will load the optimized model as shown in the following example:

```
docker run -p 8501:8501 \
  --mount type=bind,\
    source=/path/to/models,\
    target=/models/my_model \
  -e MODEL_BASE_PATH=/models \
  -e MODEL_NAME=my_model \
  -t tensorflow/serving:latest \
  --use_tflite_model=true ❶
```

- ❶ Enable TFLite model loading.

TensorFlow Lite optimized models can provide you with low-latency and low-memory footprint model deployments.

---

#### DEPLOY YOUR MODELS TO EDGE DEVICES

After optimizing your TensorFlow or Keras model and deploying your TFLite machine learning model with TensorFlow Serving, you can also deploy the model to a variety of mobile and edge devices; for example:

- Android and iOS mobile phones
- ARM64-based computers
- Microcontrollers and other embedded devices (e.g., Raspberry Pi)
- Edge devices (e.g., IoT devices)
- Edge TPUs (e.g., Coral)

If you are interested in deployments to mobile or edge devices, we recommend the publication *Practical Deep Learning for Cloud, Mobile, and Edge* by Anirudh Koul et al. (O'Reilly) for further reading. If you are looking for materials on edge devices with a focus on TFMicro, we recommend *TinyML* by Pete Warden and Daniel Situnayake (O'Reilly).

---

# Monitoring Your TensorFlow Serving Instances

TensorFlow Serving allows you to monitor your inference setup. For this task, TensorFlow Serving provides metric endpoints that can be consumed by Prometheus. Prometheus is a free application for real-time event logging and alerting, currently under Apache License 2.0. It is used extensively within the Kubernetes community, but it can easily be used without Kubernetes.

To track your inference metrics, you need to run TensorFlow Serving and Prometheus side by side. Prometheus can then be configured to pull metrics from TensorFlow Serving continuously. The two applications communicate via a REST endpoint, which requires that REST endpoints are enabled for TensorFlow Serving even if you are only using gRPC endpoints in your application.

## Prometheus Setup

Before configuring TensorFlow Serving to provide metrics to Prometheus, we need to set up and configure our Prometheus instance. For the simplicity of this example, we are running two Docker instances (TensorFlow Serving and Prometheus) side by side, as shown in [Figure 9-4](#). In a more elaborate setup, the applications would be Kubernetes deployments.

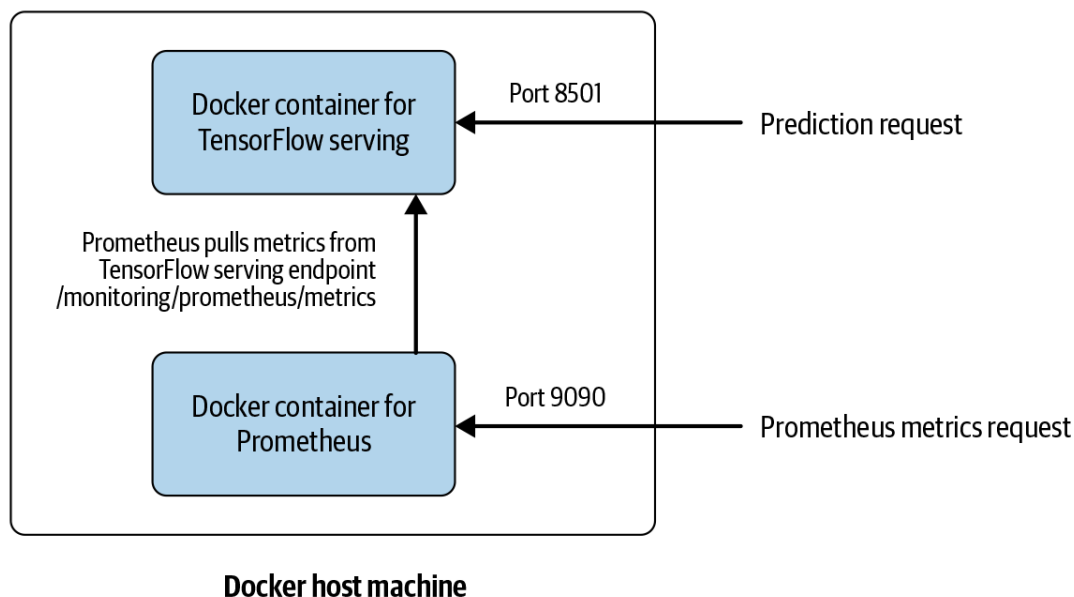


Figure 9-4. Prometheus Docker setup

We need to create a Prometheus configuration file before starting up Prometheus. For this purpose, we will create a configuration file located at `/tmp/prometheus.yml` and add the following configuration details:

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s
  external_labels:
    monitor: 'tf-serving-monitor'

scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s ❶
    metrics_path: /monitoring/prometheus/metrics ❷
    static_configs:
      - targets: ['host.docker.internal:8501'] ❸
```

- ❶ Interval when metrics are pulled.
- ❷ Metrics endpoints from TensorFlow Serving.
- ❸ Replace with the IP address of your application.

In our example configuration, we configured the target host to be `host.docker.internal`. We are taking advantage of Docker's domain name resolution to access the TensorFlow Serving container via the host machine. Docker automatically resolves the domain name `host.docker.internal` to the host's IP address.

Once you have created your Prometheus configuration file, you can start the Docker container, which runs the Prometheus instance:

```
$ docker run -p 9090:9090 \ ❶
    -v /tmp/prometheus.yml:/etc/prometheus/prometheus.yml \ ❷
    prom/prometheus
```

- ❶ Enable port 9090.

❷

Mount your configuration file.

Prometheus provides a dashboard for the metrics, which we will later access via port 9090.

## TensorFlow Serving Configuration

Similar to our previous configuration for the inference batching, we need to write a small configuration file to configure the logging settings.

With a text editor of your choice, create a text file containing the following configuration (in our example, we saved the configuration file to */tmp/monitoring\_config.txt*):

```
prometheus_config {  
  enable: true,  
  path: "/monitoring/prometheus/metrics"  
}
```

In the configuration file, we are setting the URL path for the metrics data. The path needs to match the path we specified in the Prometheus configuration that we previously created (*/tmp/prometheus.yml*).

To enable the monitoring functionality, we only need to add the path of the `monitoring_config_file` and TensorFlow Serving will provide a REST endpoint with the metrics data for Prometheus:

```
$ docker run -p 8501:8501 \  
  --mount type=bind,source=`pwd`,target=/models/my_model \  
  --mount type=bind,source=/tmp,target=/model_config \  
  tensorflow/serving \  
  --monitoring_config_file=/model_config/monitoring_config.txt
```

## PROMETHEUS IN ACTION

With the Prometheus instance running, you can now access the Prometheus dashboard to view the TensorFlow Serving metrics with the Prometheus UI, as shown in [Figure 9-5](#).

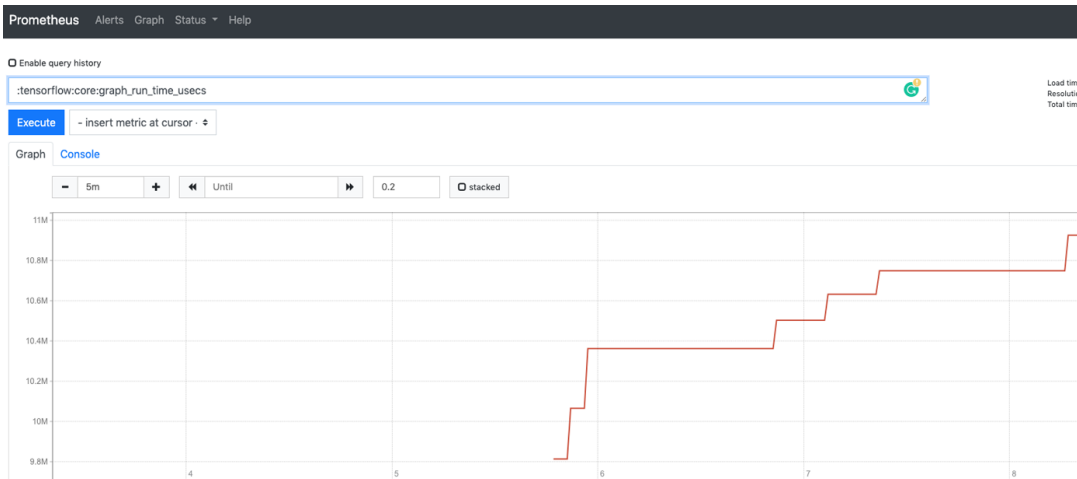


Figure 9-5. Prometheus dashboard for TensorFlow Serving

Prometheus provides a standardized UI for common metrics. Tensorflow Serving provides a variety of metric options, including the number of session runs, the load latency, or the time to run a particular graph, as shown in [Figure 9-6](#).

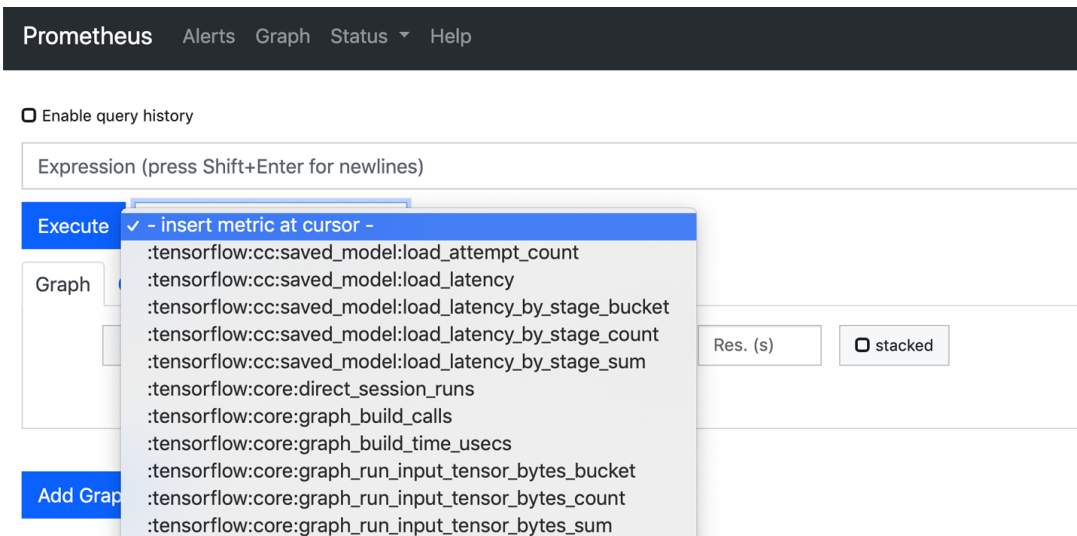


Figure 9-6. Prometheus metric options for TensorFlow Serving



# Simple Scaling with TensorFlow Serving and Kubernetes


So far, we have discussed the deployment of a single TensorFlow Serving instance hosting one or more model versions. While this solution is sufficient for a good number of deployments, it isn't enough for applications experiencing a high volume of prediction requests. In these situations, your single Docker container with TensorFlow Serving needs to be replicated to reply to the additional prediction requests. The *orchestration* of the container replication is usually managed by tools like Docker Swarm or Kubernetes. While it would go beyond the scope of this publication to introduce Kubernetes in depth, we would like to provide a small glimpse of how your deployment could orchestrate through Kubernetes.

For the following example, we assume that you'll have a Kubernetes cluster running and that access to the cluster will be via `kubectl`. Because you can deploy TensorFlow models without building specific Docker containers, you will see that, in our example, we reused Google-provided Docker containers and configured Kubernetes to load our models from a remote storage bucket.

The first source code example highlights two aspects:

- Deploying via Kubernetes without building specific Docker containers
- Handling the Google Cloud authentication to access the remote model storage location

In the following example, we use the GCP as the cloud provider for our deployment:<sup>5</sup>

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: ml-pipelines
  name: ml-pipelines
spec:
  replicas: 1 
  selector:
```

```

matchLabels:
  app: ml-pipelines
template:
  spec:
    containers:
      - args:
          - --rest_api_port=8501
          - --model_name=my_model
          - --model_base_path=gs://your_gcp_bucket/my_model ❷
        command:
          - /usr/bin/tensorflow_model_server
        env:
          - name: GOOGLE_APPLICATION_CREDENTIALS
            value: /secret/gcp-credentials/user-gcp-sa.json ❸
        image: tensorflow/serving ❹
        name: ml-pipelines
        ports:
          - containerPort: 8501
        volumeMounts:
          - mountPath: /secret/gcp-credentials ❺
            name: gcp-credentials
    volumes:
      - name: gcp-credentials
        secret:
          secretName: gcp-credentials ❻

```

- ❶ Increase replicas if needed.
- ❷ Load model from remote location.
- ❸ Provide cloud credentials here for GCP.
- ❹ Load the prebuilt TensorFlow Serving image.
- ❺ Mount the service account credential file (if Kubernetes cluster is deployed through the GCP).
- ❻ Load credential file as a volume.

With this example, we can now deploy and scale your TensorFlow or Keras models without building custom Docker images.

You can create your service account credential file within the Kubernetes environment with the following command:

```
$ kubectl create secret generic gcp-credentials \
  --from-file=/path/to/your/user-gcp-sa.json
```

A corresponding service setup in Kubernetes for the given model deployment could look like the following configuration:

```
apiVersion: v1
kind: Service
metadata:
  name: ml-pipelines
spec:
  ports:
    - name: http
      nodePort: 30601
      port: 8501
  selector:
    app: ml-pipelines
  type: NodePort
```

With a few lines of YAML configuration code, you can now deploy and, most importantly, scale your machine learning deployments. For more complex scenarios like traffic routing to deployed ML models with Istio, we highly recommend a deep dive into Kubernetes and Kubeflow.

---

#### FURTHER READING ON KUBERNETES AND KUBEFLOW

Kubernetes and Kubeflow are amazing DevOps tools, and we couldn't provide a holistic introduction here. It requires its own publications. If you are looking for further reading on the two topics, we can recommend the following publications:

- *Kubernetes: Up and Running*, 2nd edition by Brendan Burns et al. (O'Reilly)
  - *Kubeflow Operations Guide* by Josh Patterson et al. (O'Reilly)
  - *Kubeflow for Machine Learning* (forthcoming) by Holden Karau et al. (O'Reilly)
-

# Summary

In this chapter, we discussed advanced deployment scenarios, such as splitting the data science and DevOps deployment life cycles by deploying models via remote cloud storage buckets, optimizing models to reducing the prediction latency and the model memory footprint, or how to scale your deployment.

In the following chapter, we now want to combine all the individual pipeline components into a single machine learning pipeline to provide reproducible machine learning workflows.

- <sup>1</sup> More details on managing AWS access keys can be found [in the documentation](#).
- <sup>2</sup> More details on how to create and manage service accounts can be found [in the documentation](#).
- <sup>3</sup> See TensorFlow's website for more information about [optimzation methods](#) and [an in-depth pruning example](#).
- <sup>4</sup> See [Nvidia's documentation about TensorRT](#).
- <sup>5</sup> Deployments with AWS are similar; instead of the credential file, the AWS `secret` and `key` need to be provided as environment variables.