# Host Hundreds of NLP Models Utilizing SageMaker Multi-Model Endpoints Backed By GPU Instances

Integrate Triton Inference Server With Amazon SageMaker

Ram Vegiraju
Published in

Towards Data Science

.
7 min read
.
3 days ago
60

Image from Unsplash

In the past we've explored SageMaker Multi-Model Endpoints (MME) as a cost effective option to host multiple models behind a singular endpoint. While hosting smaller models is possible on MME with CPU based instances, as these models get larger and more complex in nature sometimes GPU compute may be necessary.

MME backed by GPU based instances is a specific SageMaker Inference feature that we will harness in this article to showcase how we can host hundreds of NLP models efficiently on a single endpoint. Note that at the time of this article, MME GPU on SageMaker currently

supports the following single GPU based instance families: p2, p3, g4dn, and g5.

MME GPU is currently also powered by two model serving stacks:

1. Nvidia Triton Inference Server
2. TorchServe

For the purpose of this article we will be utilizing Triton Inference Server with a PyTorch backend to host BERT based models on our GPU instance. If you are new to Triton, we will have a slight primer, but I would recommend referencing my starter article here.

**NOTE**: This article assumes an intermediate understanding of SageMaker Deployment and Real-Time Inference in particular. I would suggest following this article for understanding Deployment/Inference more in depth. We will also overview Multi-Model Endpoints, but to understand further please reference this documentation.

**DISCLAIMER**: I am a Machine Learning Architect at AWS and my opinions are my own.

## What is MME? Solution Overview

Why Multi-Model Endpoints and when would you use them? MME is a cost and management effective hosting option. A traditional SageMaker Endpoint setup will look like the following:

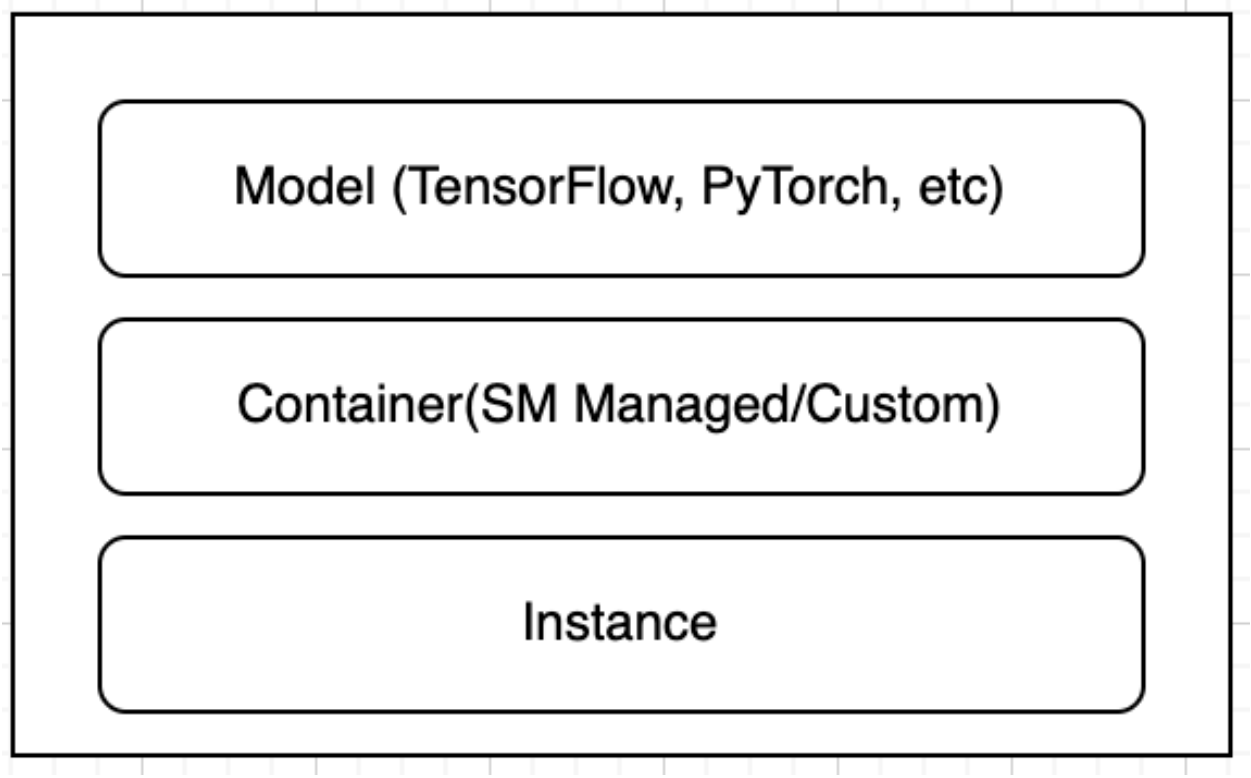



Image by Author

When you have hundreds or even thousands of models, it becomes hard to manage so many different endpoints and you have to pay for the hardware behind each persistent endpoint. With MME this becomes simplified as you have one endpoint and one set of hardware behind it for you to manage:

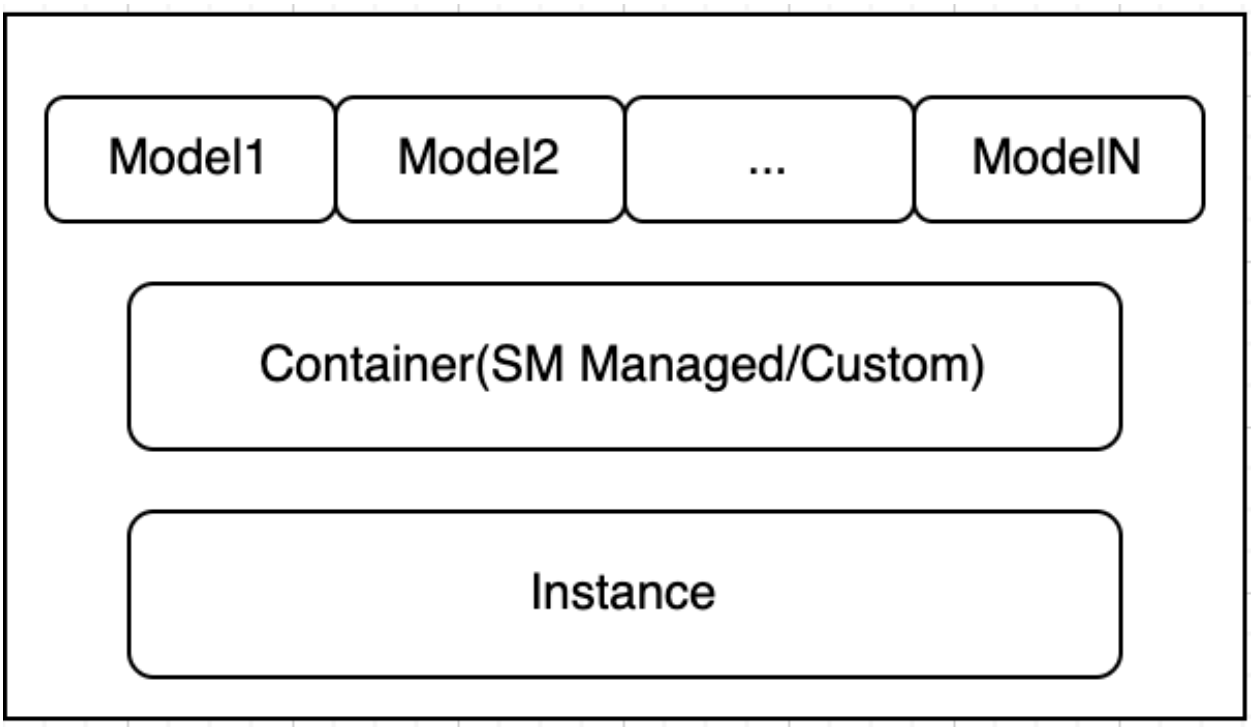
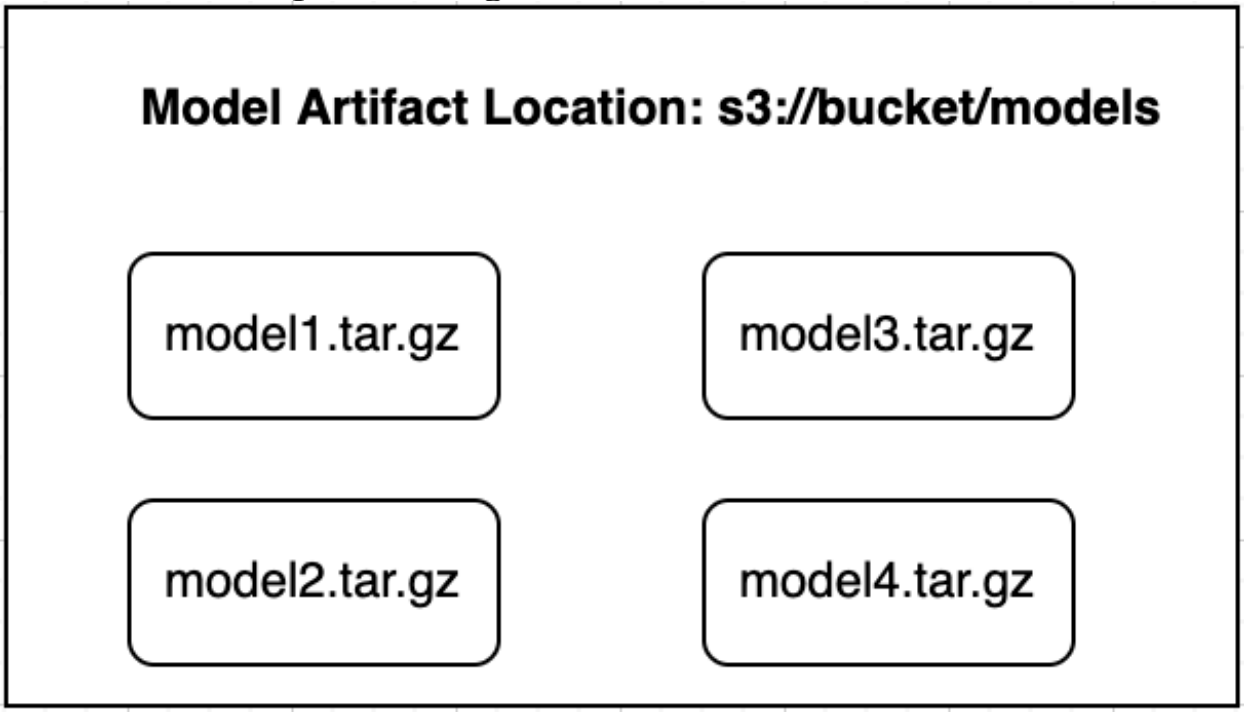


Image by Author

You can capture these different models in a model tarball (model.tar.gz). This model tarball will essentially contain all your model metadata in the format that the model serving solution expects. In this case we are using Triton as our model server so our model.tar.gz will look like the following:

```
- model.tar.gz/
  - linear_regression_model
    - 1
      - model.pt
      - model.py (optional, not included here)
    - config.pbtxt
```

For this example, we will make 200 copies of our model tarball to showcase how we can host multiple models on a singular endpoint. For real-world use-cases these tarballs will differ depending on the models you are pushing behind your endpoint. These tarballs are all captured in a common S3 path for SageMaker to understand:




MME Bucketing (Image by Author)

How are models behind MME managed? SageMaker MME will receive a request and dynamically load and cache the specific model that you have invoked. In the case that you are expecting high traffic for your endpoint it's also essential to either have multiple initial instances behind the endpoint or configure AutoScaling. For example, if a singular model is receiving a large number of invocations, this model will be loaded onto another instance to be able to serve the additional traffic. To further understand load testing SageMaker MME, please reference this guide.

## Local Setup & Testing

For this example, we will be working in a SageMaker Classic Notebook Instance with a conda_python3 kernel and a g4dn.4xlarge instance. We use a GPU based instance to locally test Triton before deploying to SageMaker Real-Time Inference.

In this example, we work with the popular BERT model. We want to first create our local model artifact, so we use PyTorch to trace and then save the serialized model artifact.

```
import torch
from transformers import BertModel, BertTokenizer
device = "cuda" if torch.cuda.is_available() else "cpu"

# Load bert model and tokenizer
model_name = 'bert-base-uncased'
model = BertModel.from_pretrained(model_name, torchscript = True)
tokenizer = BertTokenizer.from_pretrained(model_name)

# Sample Input
text = "I am super happy right now to be trying out BERT."

# Tokenize sample text
inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)

# jit trace model
traced_model = torch.jit.trace(model, (inputs["input_ids"], inputs["attention_mask"]))
```

```
# Save traced model
torch.jit.save(traced_model, "model.pt")
```

We can confirm our saved model infers properly by loading it and running a sample inference with the tokenized text.

```
# sample inference with loaded model
loaded_model = torch.jit.load("model.pt")
res = loaded_model(input_ids=inputs["input_ids"], attention_mask=inputs["attention_mask"])
res
```

We can now focus on setting up Triton to host this specific model. Why is it important to test Triton locally before implementing with SageMaker? We want to capture any issues with our setup before creating a SageMaker endpoint. Creating a SageMaker endpoint can take a few minutes and until you see the failure in the logs you won't have an idea of what's wrong with your setup even if it is as small as a scripting error or improper structuring of your model tarball. By locally testing Triton first we can quickly iterate on our configuration and model files to capture any errors.

For Triton we first need a config.pbtxt file. This captures our input and output dimensions as well as other Triton Server properties you want to tune. In this case we can grab the input and output shapes from the transformers library describing the BERT architecture.

```
from transformers import BertConfig
bert_config = BertConfig.from_pretrained(model_name)
max_sequence_length = bert_config.max_position_embeddings
output_shape = bert_config.hidden_size
print(f"Maximum Input Sequence Length: {max_sequence_length}")
print(f"Output Shape: {output_shape}")
```

We can use these values to then create our config.pbtxt file.

```
name: "bert_model"
platform: "pytorch_libtorch"

input [
 {
   name: "input_ids"
   data_type: TYPE_INT32
   dims: [1, 512]
 },
 {
   name: "attention_mask"
   data_type: TYPE_INT32
   dims: [1, 512]
 }
]

output [
 {
   name: "OUTPUT"
```

```
    data_type: TYPE_FP32
    dims: [512, 768]
  }
]
```

We then start our Triton Inference Server with the following Docker command pointing towards our model repository.

```
docker run --gpus all --rm -p 8000:8000 -p 8001:8001 -p 8002:8002 -v
/home/ec2-user/SageMaker:/models nvcr.io/nvidia/tritonserver:23.08-py3
tritonserver --model-repository=/models --exit-on-error=false --log-verbose=1
```

Once the container has been started you can make sample requests to ensure that we are able to successfully conduct inference with our existing model artifacts.

```
import requests
import json

# Specify the model name and version
model_name = "bert_model" #specified in config.pbtxt
model_version = "1"

# Set the inference URL based on the Triton server's address
url = f"http://localhost:8000/v2/models/{model_name}/versions/{model_version}/infer"

# sample invoke
```

```
output = requests.post(url, data=json.dumps(payload))
res = output.json()
```

Once this is working successfully, we can focus on our SageMaker MME deployment.

## SageMaker MME GPU Deployment

Now that we have our model artifacts in a format that our model server understands, we can encapsulate this into a model.tar.gz that is expected for SageMaker.

```
!tar -cvzf model.tar.gz bert_model/
```

We also create 200 copies of this model in a common S3 path to back our MME.

```
%%time
# we make a 200 copies of the tarball, this will take about ~6 minutes to finish (can vary depending on model size)
for i in range(200):
    with open("model.tar.gz", "rb") as f:
        s3_client.upload_fileobj(f, bucket, "{}/model-{}.tar.gz".format(s3_model_prefix,i))
```

Along with our model artifacts location, we also need to specify the managed Triton container that we are utilizing for SageMaker deployment.

```
triton_image_uri = "{account_id}.dkr.ecr.{region}.{base}/sagemaker-tritonserver:23.07-py3".format(
    account_id=account_id_map[region], region=region, base=base
)

print(f"Triton Inference server DLC image: {triton_image_uri}")
```

The next few steps are the usual SageMaker Endpoint creation flow:

- SageMaker Model: Points towards our model data and container.
- SageMaker Endpoint Configuration: Specifies our instance type and count.
- SageMaker Endpoint: REST Endpoint to invoke.

In our EndpointConfiguration object we specify a GPU based instance: g4dn.4xlarge in this instance.

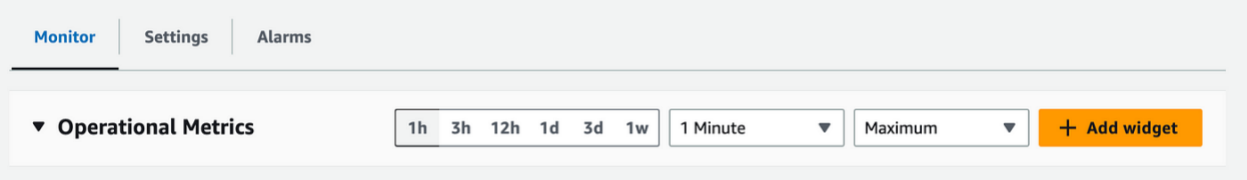
```
endpoint_config_response = client.create_endpoint_config(
    EndpointConfigName=endpoint_config_name,
    ProductionVariants=[
      {
        "VariantName": "tritontraffic",
```

```
            "ModelName": model_name,
            "InstanceType": "ml.g4dn.4xlarge",
            "InitialInstanceCount": 1,
            "InitialVariantWeight": 1
        },
    ],
)

endpoint_name = "triton-mme-gpu-ep" + time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime())

create_endpoint_response = client.create_endpoint(
    EndpointName=endpoint_name, EndpointConfigName=endpoint_config_name
)

print("Endpoint Arn: " + create_endpoint_response["EndpointArn"])
```

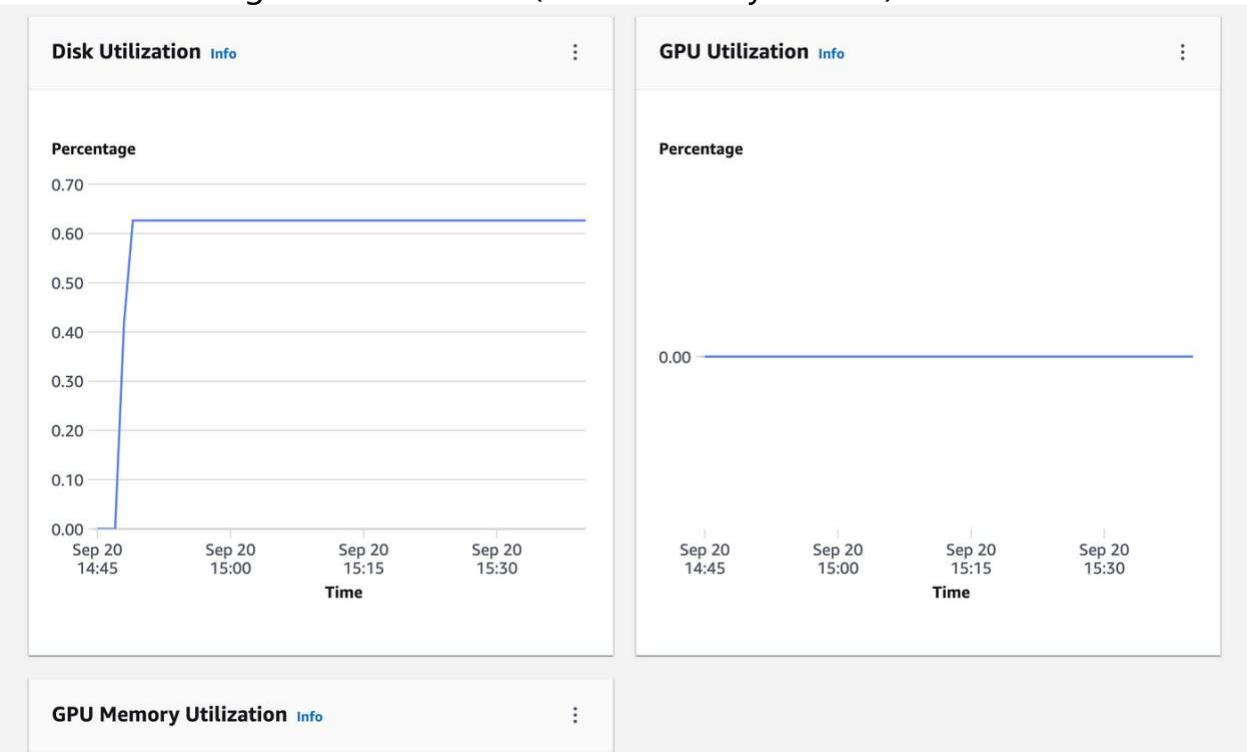
The endpoint may take a few minutes to create, but after it has you should be able to run sample inference. In the TargetModel header you can specify any model from 1–200 as we made that the delimiter for our different model.tar.gz artifacts.

```
response = runtime_client.invoke_endpoint(
    EndpointName=endpoint_name, ContentType="application/octet-stream",
    Body=json.dumps(payload), TargetModel='model-199.tar.gz'
)
print(json.loads(response["Body"].read().decode("utf8")))
```

As you run inference you can also monitor hardware and invocation metrics via CloudWatch. Specifically as this is a GPU based endpoint, you can monitor GPU Utilization via API or the SageMaker console.




Monitor Tab SageMaker Console (Screenshot by Author)




Hardware GPU Metrics (Screenshot by Author)

To understand all other MME CloudWatch metrics please reference the following documentation.

## Additional Resources & Conclusion

**SageMaker-Deployment/RealTime/Multi-Model-Endpoint/Triton-MME-GPU/mme-gpu-bert.ipynb at master ·…**

Compilation of examples of SageMaker inference options and other features. …

github.com

The entire code for the example can found at the link above. MME was already a very powerful feature, but when paired with GPU based hardware it can allow us to host larger models in the NLP and CV space. Triton is also a dynamic serving option that allows for multiple different frameworks and diverse hardware support to super charge our MME applications. For more SageMaker Inference examples please refer to the following link. If you are interested in understanding Triton better please refer to my starter guide with PyTorch models.

As always thank you for reading and feel free to leave any feedback.

*If you enjoyed this article feel free to connect with me on LinkedIn and subscribe to my Medium Newsletter. If you're new to Medium, sign up using my Membership Referral.*