Four Different Ways to Host Large Language Models on Amazon SageMaker

Pick the option that makes the most sense for your use-case

[Ram Vegiraju](#)
Published in

AWS in Plain English

.
9 min read
.
Aug 24
142
1

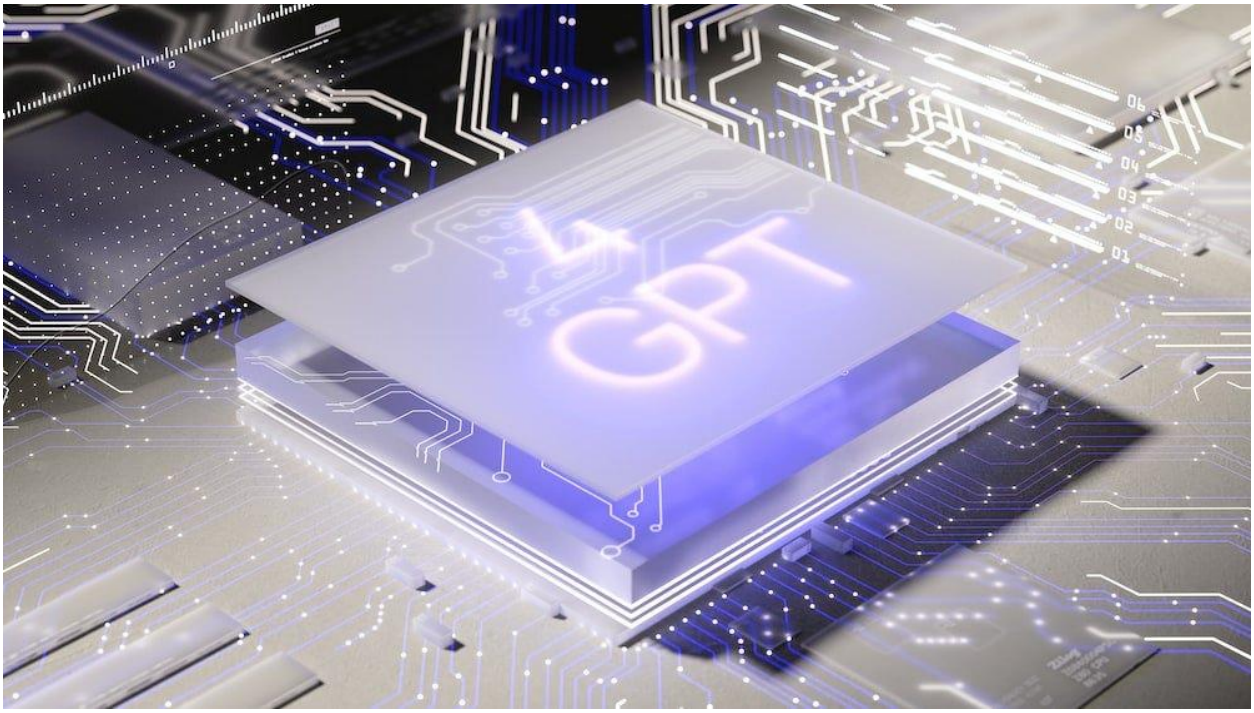


Image from Unsplash

Amazon SageMaker is a platform that can be utilized for advanced Machine Learning Model Hosting. As Generative AI continues to expand at a rapid rate, so do the challenges of hosting these large models. Many factors make hosting these models particularly difficult ranging from model size to being able to properly partition these models across GPUs.

Within Amazon SageMaker there are a variety of different options and integrations that help simplify the process of hosting LLMs. There are two main approaches that we will evaluate here:

- API driven approach that simplifies model deployment where you don't need to focus on lower level model serving optimization. You can simply provide the large language model you want and the type of hardware to back this model.
- Different Model Servers that SageMaker natively integrates with via its available Deep Learning Containers (DLCs). Here

there's more work to be done on the user's end, but greater control in your hands with the way your model is served.

In this article we'll explore some of the existing options and define each one's pros and cons and ideal use-cases.

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.

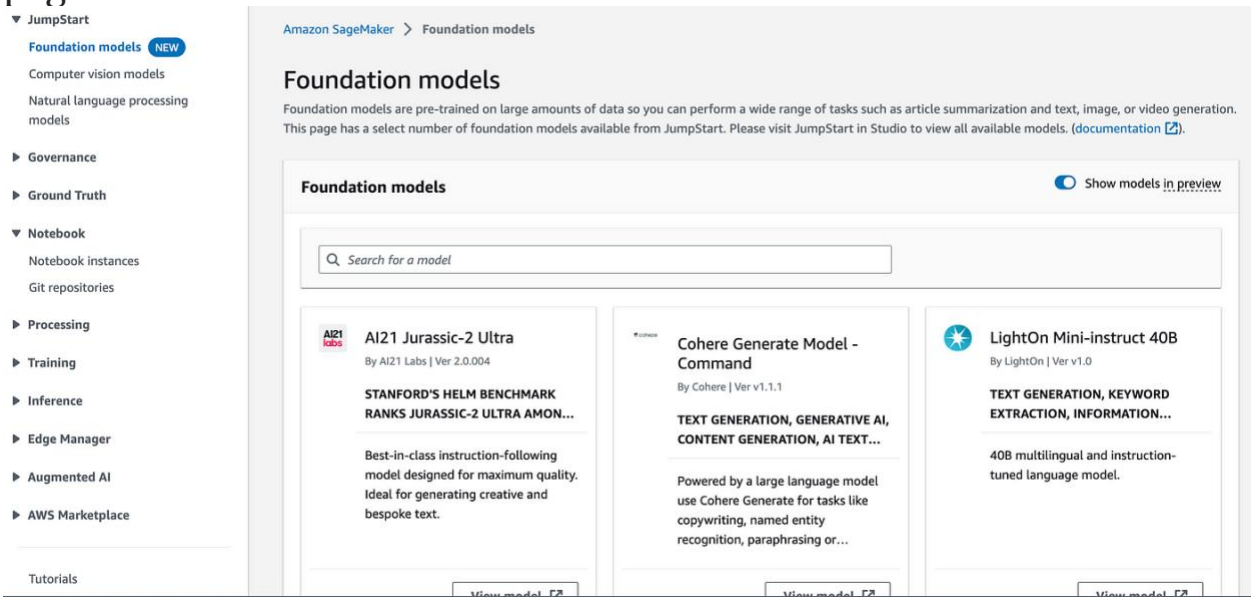
DISCLAIMER: I am a Machine Learning Architect at AWS and my opinions are my own.

Table of Contents

1. SageMaker JumpStart

SageMaker JumpStart is a feature that provides pre-trained, open source models for a variety of different domains and use-cases. With these models, lower level work such as interacting with containers and model serving is abstracted out. These models are available via the Boto3 Python SDK or the SageMaker Python SDK for quick deploy.

Specifically with the Generative AI implosion, there's a set of Foundational Models that are available from different language model providers such as Stability AI, Cohere, and more. You can find all available models within the SageMaker Console at the following page.



JumpStart Foundation Models (Screenshot by Author)

When do you use JumpStart? JumpStart is a great option if you're already satisfied with a pre-trained model's performance. To put it simply, here are some of the pros and reasons to utilize JumpStart:

- Low code approach, this is completely API driven so there's not any container or low level work that you need to do to for model hosting.
- Variety of existing Foundational Models from top language providers, this list is constantly expanding as well as the space grows.
- Option to fine-tune for certain models, this can further help with model performance.

Some reasons to consider an option outside JumpStart are the following (we will also explore how we can address these in the next section).

- If you want a large level of customization, JumpStart can be difficult. The model server is abstracted out for you, so it's hard to tell what is tunable. A way around this is to understand the container/server behind the Foundation Model by taking a look at the CloudWatch logs to see what model server has started.
- Inference scripts to control pre/post processing is not available for all foundation models, this is once again dependent on the model server that is being used under the hood.

For an example of utilizing one of the existing Foundation Models with SageMaker JumpStart, check out this article working with Cohere's Multilingual Model. Now that we've addressed some of the pros/cons with this approach, let's take a look at the alternative model server driven approach.

2. Model Server Driven Approach

I keep mentioning the word model server, what is a model server? Model Servers at a very basic premise are "inference as a service". We need an easy way to expose our models via an API, but these model servers take care of the grunt work behind the scenes. These model servers load and unload our model artifacts and provide the runtime environment for your ML models that you are hosting. These model servers can also be tuned depending on what they expose to the user.

In the case of **Large Language Models** there are **three specific model servers** that integrate with Amazon SageMaker which we will explore. SageMaker Inference ties into these model servers via the Deep Learning Containers (DLCs) that are provided by AWS. These DLCs integrate with the model servers and expose tunable parameters via environment variables that you can inject. Each model server has its own set of parameters that you can toggle to test inference performance.

When do we want to use a model server driven approach rather than the API driven approach we spoke of above?

- If you are not satisfied with the inference performance (latency and throughput) of a JumpStart solution. There are different model servers you can explore and different configurations within each model server that you can tune.
- This feeds into the aforementioned point, but if you want to have a larger amount of control in how your model is partitioned across your hardware this is more viable through a model server approach. Certain JumpStart Foundational Models do expose this via environment variables, but not the level of control which you will have via a Model Server driven approach.

2A. Deep Java Library (DJL Serving)

DJL Serving is a high performance universal model serving solution that can be utilized to host large language models. DJL Serving has been specifically tailored for large language models by supporting a variety of different model partitioning frameworks that are listed below:

- HuggingFace Accelerate
- Nvidia FasterTransformer
- Microsoft DeepSpeed

These different frameworks are available as backends for DJL Serving and you can tune specific variables for each framework in a serving.properties file that you provide to the model server. An example of variables you can play with in DJL includes batch size or worker count. Generally for a DJL Serving configuration you need three main artifacts:

- **serving.properties**: Defines the backend as well as any tunable parameters you want to inject, you can find a full list at the following link. This serving.properties file will also point towards your model data located in S3.
- **model.py (optional)**: If you would like to control pre/post processing of model inference, you can have custom code here to handle that.
- **requirements.txt (optional)**: By default PyTorch is installed, but you can also add any extra dependencies in this file.

With DJL Serving there are two routes for loading your model. If the model exists within the HuggingFace Model Hub you can point towards it in your serving.properties file.

```
#serving.properties file
engine=Python
```

```
option.model_id=facebook/bart-large
option.task=feature-extraction
```

Alternatively, you can also load your own model artifacts into S3 and point towards that path in your serving.properties file. Model download is also fastened to S3 by utilizing S5CMD.

```
#serving.properties file
engine = FasterTransformer
option.tensor_parallel_degree = 4
#sample s3 url with model data
option.s3url = "s3://sagemaker-example-files-prod-{region}/models/hf-large-model-djl-ds/flant5-xxl/
```

Reference the following articles and examples to see end to end deployment with both aforementioned approaches utilizing DJL Serving.

- Deploy Utilizing HuggingFace Model Hub
- Deploy Utilizing S3 Model Path

When should we utilize DJL Serving specifically?

- If you have pre-trained model artifacts or a HuggingFace Model Hub that you need in S3, DJL optimizations with

S5CMD makes it very easy to load this model in an efficient timely manner.

- If you are not happy with JumpStart or an API driven approach's inference performance, you can try to tune DJL Serving by experimenting with the different model partitioning backends it provides.
- If you are experienced or already integrated with Accelerate, DeepSpeed, or any of the available Model Partitioning Frameworks then DJL Serving becomes a very powerful option for you to tune in conjunction with SageMaker.

2B. Text Generation Inference (TGI)

Another model server that we can explore is Text Generation Inference (TGI) by HuggingFace. TGI is a Rust, Python, gRPC model server created by HuggingFace that can be used to host specific large language models. TGI natively supports a large set of optimizations for LLMs that can be found here.

Most importantly TGI natively supports specific model architectures including Flan-T5, Starcoder, and Falcon to name a few. These model architectures have been specifically optimized for in the TGI Model Server.

When do we use TGI specifically? Why TGI over DJL Serving when both can answer similar problems or host the same

models? In the case that there is a model such as Flan that TGI supports natively it is worth trying this option first for a few reasons:

- TGI is easy to use with SageMaker, you can simply provide the model ID and the TGI environment variables you want to tune and deploy directly with the SageMaker Python SDK. Unlike DJL Serving, there's no need for a serving.properties file or any engine selection for model partitioning. For example as seen below a Llama model deployment, takes under 20 lines of Python code.

```
# Define Model and Endpoint configuration parameter
config = {
  'HF_MODEL_ID': "decapoda-research/llama-7b-hf", # model_id from hf.co/models
  'SM_NUM_GPUS': json.dumps(number_of_gpu), # Number of GPU used per replica
  'MAX_INPUT_LENGTH': json.dumps(1024),  # Max length of input text
  'MAX_TOTAL_TOKENS': json.dumps(2048),  # Max length of the generation (including input text)
}

# create HuggingFaceModel with the image uri
llm_model = HuggingFaceModel(
  role=role,
  image_uri=llm_image,
  env=config
)

# real-time inference deployment
llm = llm_model.deploy(
  initial_instance_count=1,
```

```
    instance_type=instance_type,
    container_startup_health_check_timeout=health_check_timeout,
)
```

- In the case that the model architecture is supported by TGI, this is already an architecture that has been optimized for deployment so it's worth trying before DJL. That being said this is not always an apples to apples comparison and it is worth benchmarking both options to see what provides the best inference performance.

The one gotcha to consider with TGI is that **at the moment a custom inference script is not supported**. This means that any custom pre/post processing code you may have is not supported at the moment, in that case it is best to utilize DJL Serving or Triton which we will explore in the next section. For an end to end guideon TGI deployment please reference this article.

2C. Triton Inference Server

Nvidia Triton Inference Server is a universal model server solution that supports a variety of different frameworks such as the following:

- TensorRT
- Custom Python Backend
- TensorFlow

- PyTorch/TorchScript
- ONNX

To work with Triton you can define which backend you are utilizing along with backend specific environment variables in a config.pbtxt file as follows:

```
name: "triton_sample_deployment"
backend: "python" #different frameworks supported
max_batch_size: 8 #triton specific env variables
```

Why use Triton for LLM Hosting? While DJL Serving and TGI have very large language model specific optimizations, Triton has a vast array of dynamic options that can supercharge your LLM applications while also providing solid inference performance. A few of these features include:

- **CPU and MME GPU Support**: Triton models can be executed on CPUs or GPUs for performance. Currently with SageMaker Multi-Model Endpoints, Triton can be used as the model server for GPU based endpoints. A code sample is in the following Stable Diffusion MME GPU example where you can host multiple variants of a Stable Diffusion model behind a single endpoint utilizing Triton.

- **Model Pipelines/Ensembles**: Triton model ensembles combine multiple model executions, you can have a model with your preprocessing logic, a model with your inference, and a model with post processing. You can essentially stitch together multiple models which will process an inference request sequentially as it comes in.
- **Dynamic batching**: For models that support batching, Triton has multiple built-in scheduling and batching algorithms that combine individual inference requests together to improve inference throughput. These scheduling and batching decisions are transparent to the client requesting inference.
- **Concurrent model execution**: Multiple instances of the same model can run simultaneously on the same GPU or on multiple GPUs.

Triton can be a little harder to setup due to input/output data configuration in the config.pbtxt file, but it's wide array of options gives you a large amount of flexibility if you have a unique LLM hosting use-case.

3. Additional Resources & Conclusion

For more LLM Hosting examples related to SageMaker Inference, check out the official workshop here. SageMaker Real-Time Inference is a powerful tool that integrates with a variety of options. As LLMs

continue to soar in popularity it's essential to consider all options available for hosting your model to achieve the best possible inference performance. Each option has its own pros and cons and I hope this article was a useful primer for people working with LLM Hosting on SageMaker.