

Meta releases Muse Spark Open, 816-billion-parameter multimodal model

Source copy kept by wheresthe.ai for the story "Meta releases Muse Spark Open, 816-billion-parameter multimodal model" (<https://wheresthe.ai/d/us-meta-releases-muse-spark-open-816-billion-parameter-multi-2026-10/>).

Source: <https://huggingface.co/blog/meta-muse-spark-open>

Text saved 18 Oct 2026 (the page as published).

SHA-256 of the saved text: ed415ccbc320c02d02aee3e88a2dcba059cc9986aa37814df02a4b920c1603db

[Back to Articles](#) a]:hidden">

Welcome Muse Spark Open: open-weight frontier multimodal mixture-of-experts model from Meta

Published October 15, 2026

[Update on GitHub](#) [Upvote 52](#)

+46

[Pedro Cuenca](#) pcuenq [Follow](#)

[mervemerve](#) mervemerve [Follow](#)

[ben burtenshaw](#) burtenshaw [Follow](#)

[Guarin Flück](#) guarin [Follow](#)

[Quentin Gallouédec](#) qgallouedec [Follow](#)

[Alvaro Bartolome](#) alvarobartt [Follow](#)

[Serveurperso](#) Serveurperso [Follow](#)

[Jun Kim Jundot](#) [Follow](#)

:last-child]:mb-0">

Muse Spark Open is an open-weight model from Meta. The mixture of experts model brings text, images, audio, and video into a model with approximately 816 billion parameters, around 42 billion active per token, and a context window of up to one million tokens. It generates text, giving developers a common interface for asking questions about different kinds of input.

Following Muse Glimmer, Muse Spark Open introduces a mixture-of-experts architecture at a much larger scale, and the most capable US open-weights model to date.

Meta trained Muse Spark Open for agentic coding: reading a codebase, writing and debugging code, and calling tools. It also takes images, audio, and video as input, so the same model can read a screenshot, listen to a voice note, or watch a screen recording while it works on your code. This post shows you how to run it, how to fine-tune it, and what we built with it. Find all the models here .

Overall Capabilities and Architecture

Meta describes Muse Spark Open as a coding-optimized model trained on real developer workflows such as multi-file refactors and long debugging sessions. Muse Spark Open is an agentic focused model that plans, acts, observes, and self-corrects over long tasks, including browsing the web with images.

Feature Muse Spark Open

Model size ~816B total parameters; ~42B active per token

Expert routing 256 experts per MoE layer; 8 selected per token

Inputs Text, images, audio, and video

Output Text

Context window Up to 1M tokens

Checkpoint formats BF16 and NVFP4

The checkpoint repositories are Muse-Spark-Open-816B-A42B and Muse-Spark-Open-816B-A42B-NVFP4.

Muse Spark Open is the full-size model of the Muse series of models. Muse Glimmer, which Meta released in August, is a small dense model from the same series, ideal for running locally. Both models use the same tokenizer and vocabulary. The chat template follows the Muse Glimmer template, with additions for the new modalities. In Muse Spark Open the input embedding and the output layer are separate matrices of about 1.66B parameters each.

Mixture of Experts

The language model has 62 layers and a hidden dimension of 8192. Each layer has 128 query heads and 16 key-value heads (grouped-query attention). Each head has 64 dimensions. The attention block has an output gate and an output normalization layer. The model also applies normalization to the queries and the keys (QK-norm). Each block combines its input and its output through a learned residual gate, not through a plain sum.

Muse Glimmer uses one dense feed-forward network in each layer. Muse Spark Open replaces this dense network with a Mixture-of-Experts (MoE) block. Each MoE block has 256 experts, and a router selects 8 experts for each token. Each expert is a feed-forward network that projects from 8192 to 4096 dimensions and back to 8192, then applies its own normalization to its output. The experts hold ~97% of the total model parameters.

Only 8 experts run per token, so the model computes each token with about 42B parameters. All 256 experts stay in memory, so deployment needs the memory of an 816B-parameter model, but runs at the much faster speed of a 42B model. The BF16 checkpoint is about 1.5 TB in safetensors format. The NVFP4 checkpoint stores the parameters in 4-bit precision with negligible loss in accuracy, specifically for NVIDIA Blackwell GPUs. In transformers, the BF16 checkpoint runs across 4 nodes of 8 H200 GPUs with tensor parallelism (TP=32) and the NVFP4 runs on a single B200 node. If you want to deploy Muse Spark Open, check out the llama.cpp , vLLM, and SGLang recipes below.

Perception Encoder

A perception encoder converts images and video frames into tokens for the language model, just like Muse Glimmer. The encoder has 1.8B parameters, 50 layers, a hidden dimension of 1536, and 16 attention heads. It splits each image into 14×14 patches and produces features with 4096 dimensions. Position information comes from learned absolute embeddings and 2D RoPE. The feed-forward network in each layer uses LayerNorm and GELU. Every fourth layer and the last layer use full attention, and the other layers use sparse attention.

Muse Glimmer 30B uses an encoder with the same architecture. The design follows the Perception Encoder work from Meta. Video input goes through the same encoder as images.

Audio Encoder

Muse Spark Open accepts audio as input through its audio encoder. An audio tokenizer converts the audio into tokens, and the chat template marks the audio position with <|audio|> tokens. The encoder emits one audio token every 80 ms, which yields 12.5 tokens per second. So let's say, you have an audio of 10 seconds, the number of input audio tokens would be 125. The encoder itself is a convolutional pyramid where each stage consists of a patch embedder of different patch sizes and 2 convolutional blocks of 1d kernels of size 7 with stride of 1. These stages are repeated for patch sizes of [16, 6, 5, 4]. At the end of the final stage, the temporal frames are fed to a final block of RMSNorm → linear projection → RMSNorm.

Long Context

The context window is 1M tokens at inference. The configuration sets max_position_embeddings to 1,048,576. This includes image, video, and audio tokens in the same window as text.

Grouped-query attention keeps the KV cache small for a model of this size. Each token adds $2 \times 16 \text{ heads} \times 64 \text{ dimensions} \times 62 \text{ layers} = 126,976$ values to the cache. In BF16, this is about 250 KB per token and about 266 GB for a full 1M-token context.

Inference Support

Transformers

The Muse Spark Open implementation provides `MuseSparkForConditionalGeneration` for text generation from multimodal inputs and `MuseSparkProcessor` for preparing conversations. The processor combines the tokenizer, image and video processors, audio feature extractor, and chat template.

Use a `DistributedConfig` to configure the parallelism settings. `tp_size` is the number of devices across which the dense layers are sharded, and `ep_size` is the number of devices across which the experts are distributed.

You can initialize the model and processor as follows.

```
import os

from transformers import AutoModelForMultimodalLM, AutoTokenizer, DistributedConfig

distributed_config = DistributedConfig( tp_size= int (os.environ[ "WORLD_SIZE" ]), ep_size= int (os.environ[
"WORLD_SIZE" ]), )

tokenizer = AutoTokenizer.from_pretrained( "meta-models/Muse-Spark-Open-816B-A42B " ) model =
AutoModelForMultimodalLM.from_pretrained( "meta-models/Muse-Spark-Open-816B-A42B" ,
distributed_config=distributed_config, )

messages = [ { "role" : "user" , "content" : "Explain in one sentence why the sky is usually blue." }, ] inputs =
tokenizer.apply_chat_template( messages, reasoning_effort= "medium" , # default: xhigh, options: low/medium/
high/xhigh add_generation_prompt= True , tokenize= True , return_dict= True , return_tensors= "pt" ,
).to(model.device).to(model.dtype)

input_len = inputs[ "input_ids" ].shape[- 1 ]

outputs = model.generate(**inputs, max_new_tokens= 2000 )
```

Text-only Inference

Prepare the conversation, then pass the messages to `processor.apply_chat_template` , enabling `add_generation_prompt=True` , `tokenize=True` , `return_dict=True` , and `return_tensors="pt"` . Pass the processor output to `model.generate` and decode the newly generated tokens, excluding the input prefix. The processor has a lever for the reasoning effort, mapping symbolic levels (low/medium/high/xhigh) to the integers the model itself accepts (16, 64, 256, 512). Note that turning off reasoning is not possible with Muse Spark Open, so keep the model reasoning at “low” if you want to consume less tokens to solve an easier task. You can also set a numeric effort by including a line like Reasoning strength: 1 in the system prompt.

```
messages = [ { "role" : "system" , "content" : "You should only answer with a number." }, { "role" : "user" , "content" :
"What is 17 * 23?" }, ]

inputs = processor.apply_chat_template( messages, add_generation_prompt= True , tokenize= True ,
return_dict= True , return_tensors= "pt" , reasoning_effort= "high" , # options:low/medium/high/xhigh
).to(model.device).to(model.dtype)

output = model.generate(**inputs, max_new_tokens= 2000 ) generated_tokens = output[ 0 ][inputs[ "input_ids"
].shape[ 1 ] :] print (processor.decode(generated_tokens, skip_special_tokens= False ))
```

Images and Text

The Muse Spark Open processor expands each image marker into the number of patch tokens required by the processed image, surrounded by image boundary tokens. This expansion keeps the text positions aligned with the visual features passed to the model. The processor also handles multiple images within a single prompt.

```
image_url = ( "https://huggingface.co/datasets/merve/vl-test-suite/" "resolve/main/pills.jpg" ) messages = [ { "role" :
"user" , "content" : [ { "type" : "image" , "image" : image_url }, { "type" : "text" , "text" : "Do any of the components
in this supplement interact?" }, ], }, ]

inputs = processor.apply_chat_template( messages, tokenize= True , add_generation_prompt= True ,
reasoning_effort= "medium" , return_dict= True , return_tensors= "pt" , ).to(model.device).to(model.dtype)

input_len = inputs[ "input_ids" ].shape[- 1 ]
```

```
outputs = model.generate(**inputs, max_new_tokens= 2000 ) response = processor.decode(outputs[ 0 ]
[input_len:], skip_special_tokens= False )
```

```
processor.parse_response(response)
```

Video Inference

Video prompts include frame timing as well as visual content. The processor handles videos with audios in them (which is an upgrade from Muse Glimmer thanks to the audio encoder). The processor downsamples video frames and audio to align frames together. By default, videos are sampled to 2 FPS, capped at 96 frames. You can modify this by setting the fps of the video processor:

```
processor.video_processor.fps = 4.0
```

As a general rule of thumb, videos are frames in temporal sequence, and the more you downsample, the more you will lose frames in between specific events, so you need to keep fps high if you want more accurate generations. However, there's a trade-off because you will require more memory and increase latency for prefill. The frames are interleaved with text input that states timestamps (e.g. "Time: 3.5s"). To enable passing in videos with audio, pass `load_audio_from_video=True` to `apply_chat_template`.

Preserve the video's metadata when preparing inputs with `return_metadata=True` so the processor can assign the correct framerate.

```
import torch
```

```
messages = [ { "role" : "user" , "content" : [ { "type" : "text" , "text" : ( "Describe what happens in this video. " "Use
timestamps and include relevant speech or sounds." ) , }, { "type" : "video" , "url" : "https://huggingface.co/datasets/
merve/vlm_test_images/resolve/main/ramen.mp4" } , ], } ]
```

```
inputs = processor.apply_chat_template( messages, add_generation_prompt= True , tokenize= True ,
return_dict= True , return_tensors= "pt" , load_audio_from_video= True , return_metadata= True ,
).to(model.device).to(model.dtype)
```

```
with torch.inference_mode(): output_ids = model.generate( **inputs, max_new_tokens= 256 , do_sample= False ,
)
```

```
generated_ids = output_ids[:, inputs[ "input_ids" ].shape[ 1 ] :] response = processor.batch_decode(
generated_ids, skip_special_tokens= True , )[ 0 ]
```

```
print (response)
```

Audio Inference

The Muse Spark Open audio tower consumes a raw waveform. The processor expects mono audio at 24 kHz and pads the waveform to a multiple of 1,920 samples, corresponding to one audio embedding per 80 milliseconds. Audio supplied directly to the processor at another sample rate must first be resampled. You can do this by using the `load_audio` util of `transformers`. `load_audio` uses `torchcodec`, `librosa` or `torchaudio` backends, so make sure to have one of these installed locally before running inference.

If you want to simply use `apply_chat_template` with messages though, this is all handled for you under the hood:

```
import torch from transformers import AutoProcessor, AutoModelForMultimodalLM
```

```
model_id = "meta-models/Muse-Spark-Open-816B-A42B" processor = AutoProcessor.from_pretrained(model_id)
```

```
model = AutoModelForMultimodalLM.from_pretrained(model_id). eval ()
```

```
messages = [ { "role" : "user" , "content" : [ { "type" : "audio" , "url" : "https://huggingface.co/buckets/huggingface/
audio-samples/resolve/bbaudio_question_39.mp3" , }, { "type" : "text" , "text" : "Answer the question." , }, ], } ]
```

```
# Downloads the audio, decodes it as mono, resamples it to 24 kHz, pads it, creates input_values_mask, and
expands the audio placeholders. inputs = processor.apply_chat_template( messages, add_generation_prompt=
True , tokenize= True , return_dict= True , return_tensors= "pt" , load_audio_backend= "auto" ,
).to(model.device).to(model.dtype)
```

```
with torch.inference_mode(): output_ids = model.generate( **inputs, max_new_tokens= 128 , do_sample= False ,
```

)

```
# Decode only newly generated tokens prompt_length = inputs[ "input_ids" ].shape[ 1 ] generated_text =  
processor.batch_decode( output_ids[:, prompt_length:], skip_special_tokens= True , )  
  
print (generated_text[ 0 ])
```

The processor inserts the matching number of audio tokens between audio boundary markers. It also returns a mask identifying real samples, allowing the model to exclude padded audio embeddings.

Tool Calling

Muse Spark Open can do text-only and multimodal tool calling. You can make the model call tools as follows.

```
from transformers import AutoProcessor, AutoModelForMultimodalLM
```

```
MODEL_ID = "meta-models/Muse-Spark-Open-816B-A42B"
```

```
# Load model processor = AutoProcessor.from_pretrained(MODEL_ID) model =  
AutoModelForMultimodalLM.from_pretrained( MODEL_ID, dtype= "auto" , device_map= "auto" )
```

```
tools = [ { "type" : "function" , "function" : { "name" : "weather.get" , "description" : "Get the current weather for a  
city." , "parameters" : { "type" : "object" , "properties" : { "city" : { "type" : "string" } , }, "required" : [ "city" ] , }, } , }
```

```
messages = [ { "role" : "user" , "content" : [ { "type" : "image" , "image" : "https://huggingface.co/datasets/merve/vl-  
test-suite/resolve/main/SF.png" } , { "type" : "text" , "text" : "I'm going to the city in this picture. What clothes should  
I wear?" } , ] , }, ]
```

```
inputs = processor.apply_chat_template( messages, tools=tools, tokenize= True , return_dict= True ,  
return_tensors= "pt" , add_generation_prompt= True , reasoning_strength= "high"  
) .to(model.device) .to(model.dtype)
```

```
input_len = inputs[ "input_ids" ].shape[- 1 ] outputs = model.generate(**inputs) response =  
processor.parse_response( completion, prefix=inputs[ "input_ids" ][ 0 ] , tools=tools , ) print (response)
```

Muse Spark Open uses a channel-based chat template (Harmony, first introduced by gpt-oss). Note that you don't have to know about the internals, but it's useful to check the channels the model routes to, in case you get low quality outputs. In the input message below:

```
<|begin_of_text|> <|start|>system<|message|> You are a helpful AI assistant. Knowledge cutoff: 2026-01-04.  
Current date: ... .. <|eot|> <|start|>user<|message|>... <|eot|> <|start|>assistant
```

```
<|start|>assistant is intentionally left open, it's the model that chooses recipient with to= . The model has different  
recipients, e.g. self asks the model to reason,
```

```
to=self<|message|>the model reasons here<|eom|> # reasoning <|start|>assistant to=user<|message|>final  
answer<|eot|> # final answer
```

self is for reasoning, commentary are comments before and between tool calls visible to the end user, user is the final response. Tool-call routing is parsed as follows (assuming you have passed tools to apply_chat_template :

```
<|start|>assistant to=weather.get<|message|> <|atem:function_calls|> <|atem:invoke name="weather.get"|>  
<|atem:parameter name="city"|>San Francisco</atem:parameter> </atem:invoke> </atem:function_calls> <|eot|>
```

Llama.cpp

Muse Spark 2-bit running on Llama Server

Muse Spark Open is integrated in llama.cpp and comes in the prebuilt binaries. You can install this with following command:

```
curl -LsSf https://llama.app/install.sh | sh
```

You can start the server as follows with quant suffix attached to model ID:

```
llama serve -hf ggml-org/Muse-Spark-Open-816B-A42B-GGUF:Q2_K
```

The repository containing quant is here . Once the server has started, you can head to localhost:9931 to chat

with the built-in WebUI. From the web UI you can add audio and images for inference and adjust the reasoning lever. Note that turning off reasoning is not possible with Muse Spark Open, so keep the model reasoning at “low” if you want to consume less tokens to solve an easier task. We tested quants on Mac M3 Ultra (512GB). An imatrix-calibrated 2-bit quant runs at 18 tokens/s, with plenty of room free for the KV cache (304 GB used by the model).

You can also query the server as follows. Feel free to check out the API reference for multimodality, tool calls and more.

```
## Text-only inference curl http://localhost:8080/v1/chat/completions \ -H "Content-Type: application/json" \ -d '{
"messages": [ {"role": "system", "content": "You are a helpful assistant."}, {"role": "user", "content": "Write a limerick
about python exceptions"} ] }'
```

```
## Image-text inference curl -s http://127.0.0.1:8080/v1/chat/completions \ -H "Content-Type: application/json" -H
"Authorization: Bearer <your-key>" \ -d '{"model": "muse-spark-open", "messages": [{"role": "user", "content": [
{"type": "text", "text": "What is in this image?"}, {"type": "image_url", "image_url": {"url": "data:image/png;base64,..."}
]}], "temperature": 0}'
```

```
## Audio-text inference curl -s http://127.0.0.1:8080/v1/chat/completions \ -H "Content-Type: application/json" -H
"Authorization: Bearer <your-key>" \ -d '{"model": "muse-spark-open", "messages": [{"role": "user", "content": [
{"type": "text", "text": "Transcribe this audio."}, {"type": "input_audio", "input_audio": {"data": "data:audio/
wav;base64,..."} ]}], "temperature": 0}'
```

llama.cpp team ships different quantized versions. We have tested Q2_K, Q4_K_S and Q_8 on Apple Silicon chips. You can run each quant by adding the quant id as suffix, llama serve -hf ggml-org/Muse-Spark-Open-816B-A42B:Q2_K .

SGLang

The Muse Spark Open integration for SGLang supports text, image, audio, and video inputs. The NVFP4 reference configuration runs across four GB300 GPUs.

From the SGLang integration environment, download the checkpoint from the Hugging Face Hub and start the server:

```
export MODEL_ID=meta-models/Muse-Spark-Open-816B-A42B-NVFP4 export CKPT="$(hf download
"$MODEL_ID")"
```

```
SGLANG_FLASHINFER_NVFP4_PER_TOKEN_ACTIVATION=1 \ uv run sglang_meta serve \ --model-path
"$CKPT" \ --served-model-name "$MODEL_ID" \ --quantization modelopt_fp4 \ --moe-runner-backend
flashinfer_trtllm \ --tp-size 4 \ --ep-size 4 \ --port 30000
```

The checkpoint includes the chat template, which we pass explicitly. The launcher configures the Muse reasoning and tool-call parsers.

Send a request through the OpenAI-compatible API:

```
curl http://localhost:30000/v1/chat/completions \ -H "Content-Type: application/json" \ -d '{ "model": "meta-models/
Muse-Spark-Open-816B-A42B-NVFP4", "messages": [ { "role": "user", "content": "Explain how a mixture-of-experts
model works." } ], "max_tokens": 2048 }'
```

The response separates the reasoning trace in reasoning_content from the final answer in content .

vLLM

The Muse Spark Open reference integration for vLLM supports text and image inputs with the NVFP4 checkpoint that require blackwell series GPUs.

After installing the reference plugin and its dependencies, download the checkpoint from the Hugging Face Hub and launch the server:

```
export MODEL_ID=meta-models/Muse-Spark-Open-816B-A42B-NVFP4 export CKPT="$(hf download
"$MODEL_ID")"
```

```
VLLM_ALLOW_LONG_MAX_MODEL_LEN=1 \ VLLM_PLUGINS=musespark1x_models.ptq_nvfp4 \ vllm serve
```

```
"$CKPT" \ --served-model-name "$MODEL_ID" \ --tokenizer-mode musespark1x \ --config-format auto \ --load-format internal_safetensor \ --tensor-parallel-size 4 \ --enable-expert-parallel \ --max-model-len 131072 \ --gpu-memory-utilization 0.92 \ --reasoning-parser musespark1x \ --tool-call-parser musespark1x \ --enable-auto-tool-choice \ --port 8000
```

This configuration sets a 128K-token serving context. The plugin supplies the model implementation, checkpoint loader, and reasoning and tool-call parsers.

Send a request using the same Hub model ID:

```
curl http://localhost:8000/v1/chat/completions \ -H "Content-Type: application/json" \ -d '{ "model": "meta-models/Muse-Spark-Open-816B-A42B-NVFP4", "messages": [{ "role": "user", "content": "Explain how a mixture-of-experts model works." }], "max_tokens": 2048 }'
```

MLX

Muse Spark Open is available in oMLX and mlx-lm today. oQ quantized models optimized for oMLX are available in 3-bit and 4-bit, ready to use right away.

We tested the oQ3e quant on an M3 Ultra (512 GB). The model takes about 344 GB, leaving room for the KV cache, and runs at about 225 tok/s prefill and 21 tok/s generation with a 1024-token prompt and 128 generated tokens.

Once you download the model through the oMLX interface, you can chat with it in the built-in chat UI. From the chat UI you can add images and adjust the reasoning effort on the fly, from low to xhigh. The model's reasoning shows up as a collapsible thinking block, separate from the final answer.

Note that it's not possible to turn reasoning off, and the default (Auto) uses xhigh. Set it to "low" if you want to use fewer tokens on easier tasks.

The language model is also available in this PR of the official mlx-lm repository. You can run it on the same high-quality oQ3e quant like this:

```
mlx_lm.generate --model mlx-community/Muse-Spark-Open-816B-A42B-oQ3e \ --prompt "Explain convolutional networks like I am 5" \ --max-tokens 5000
```

This is a big model, even the 3-bit quantization requires more than 256 GB of RAM, as explained above. You can also run distributed inference across multiple computers, please refer to the documentation for details.

Deploying Muse Spark Open with Inference Endpoints

You can deploy Muse Spark Open on Inference Endpoints with a pre-validated SGLang recipe for the NVFP4 variant which runs on either 8 × NVIDIA RTX Pro 6000 98 GB with 131072 context length, or with 8 × NVIDIA H200 141GB if you want 1048576 context length instead.

To deploy, run one command with the Hugging Face CLI:

```
hf endpoints catalog deploy --repo meta-models/Muse-Spark-Open-816B-A42B-NVFP4 --name muse-spark
```

Or open endpoints.huggingface.co/new/meta-models/Muse-Spark-Open-816B-A42B-NVFP4 and select Create Endpoint.

The endpoint exposes an OpenAI-compatible API with both Chat Completions API at /v1/chat/completions and Responses API at /v1/responses .

```
curl "https://<your-endpoint>.endpoints.huggingface.cloud/v1/chat/completions" \ -H "Authorization: Bearer $HF_TOKEN" \ -H "Content-Type: application/json" \ -d '{ "model": "meta-models/Muse-Spark-Open-816B-A42B-NVFP4", "messages": [{ "role": "user", "content": [ { "type": "image_url", "image_url": { "url": "https://huggingface.co/datasets/merve/vl-test-suite/resolve/main/pills.jpg" } }, { "type": "text", "text": "Do any of the components in this supplement interact?" } ] }, { "role": "assistant", "content": "" } ], "chat_template_kwargs": { "enable_thinking": false } }'
```

Any OpenAI-compatible client works with the same URL and your Hugging Face token as the API key.

Fine-tuning with TRL

You can fine-tune the 816B checkpoint with the TRL SFTTrainer . This section shows how we fine-tuned it with

TRL on 8 nodes of 8 H100 GPUs. The recipe trains LoRA adapters on the attention layers. The experts stay frozen. The full script is in the TRL examples .

First, we need to split the model across 64 GPU devices. Given the large size of the model, we need to parallelize it across multiple axes. We shard the model with FSDP and split the experts across GPUs, using `DistributedConfig` , which tells transformers how to split the model when it loads:

```
from transformers.distributed import DistributedConfig
```

```
distributed_config = DistributedConfig(fsdp_size= 64 , ep_size= 8 )
```

In the snippet above, `fsdp_size=64` shards the model across all 64 GPUs with FSDP2, and `ep_size=8` splits the 256 experts of each layer into 8 groups. Transformers sends each token to the GPU that holds its expert, so each GPU trains on its own slice of the batch.

Now, we can configure the `SFTConfig` for our training run. We've performed a simple task here to adapt the model's chat template on an open source dataset, but you could apply it to your own use case.

```
import torch from datasets import load_dataset from peft import LoraConfig from transformers.distributed import DistributedConfig from trl import SFTConfig, SFTTrainer
```

```
train_dataset = load_dataset( "allenai/tulu-3-sft-mixture" , split= "train[:500000]" )
```

```
training_args = SFTConfig( output_dir= "muse-spark-lora-sft" , model_init_kwargs={ "dtype" : torch.bfloat16, "distributed_config" : DistributedConfig(fsdp_size= 64 , ep_size= 8 )}, per_device_train_batch_size= 1 , max_steps= 500 , max_length= 2048 , packing= True , save_only_model= True , logging_steps= 1 , report_to= "trackio" , )
```

```
trainer = SFTTrainer( model= "meta-models/Muse-Spark-Open-816B-A42B" , args=training_args, train_dataset=train_dataset, peft_config=LoraConfig(r= 16 , lora_alpha= 32 , target_modules=[ "q_proj" , "k_proj" , "v_proj" , "o_proj" ]), ) trainer.train()
```

The core parts here are `packing=True` which joins examples into 2048-token sequences without padding. A block-diagonal mask keeps the examples separate. `save_only_model=True` saves the adapter and no optimizer state, because the trainer cannot resume a model that was sharded at load time. `LoraConfig` targets the four attention projections and the 256 experts of each layer stay frozen.

Then we can run this command across each node:

```
torchrun --nnodes 8 --nproc_per_node 8 --rdzv_backend c10d --rdzv_endpoint " $MASTER_IP :29700" sft_muse_spark_open.py
```

We trained for 500 steps, which took about 2 hours on 64 H100s, at about 8,000 tokens per second. Mean token accuracy rose from 72% to 84% and the loss fell from 1.26 to 0.57, which signals a good fine-tuning run.

Demos

Muse Spark Open is a coding and agentic model. We wanted to see that for ourselves, so we tried a mixture of multi-modal and agentic tasks that use the model's image and audio inputs. Everything below is something you can run or fork.

3D Asset Generator

We combined 2-bit quantized Muse Spark open (served on llama.cpp) with the Pi coding agent to generate and iteratively improve 3D assets in Three.js . You can find the prompt here , as the model is quantized it requires a bit more steering. We get 18 TPS on llama.cpp on Mac Ultra.

We initially describe the asset, and Pi uses the model to generate Three.js code for the asset, renders the result in a browser, and captures screenshots from multiple angles. It then sends those images back to the same model on llama server for visual feedback. Pi uses that feedback to edit the code and repeats the process.

Browser Interactions with OpenEnv and Hugging Face Sandbox

We tested Spark on computer use tasks and were amazed by the performance, so we have built a lean demo that also serves to teach you how a computer use agent works. You can also further customize this for your needs.

This is both an educational example and something you can run in a few lines. You can find it here .

Muse Spark NVFP4 Browser Agent with OpenEnv and HF Sandboxes

Hugging Face Sandboxes are sandboxes built on Hugging Face Jobs for easy code execution. The demo consists of three components:

Muse Spark Open served with Inference Endpoints,

OpenEnv wrapping web driver actions with helium and selenium (click, type, scroll etc) defining browser environment's reset, step, observation, termination interface,

Chromium web browser

where the remote sandbox contains the environment and the browser as illustrated below.

You can easily try this with a few steps, described in the example. Start Inference Endpoint with Muse Spark Open (NVFP4 works well too!), add your endpoint URL as environmental variable, publish your OpenEnv server in Spaces and also provide that, then you can run the agent loop. Muse Spark Open sees the previous actions, system prompt, url and the screenshot of the browser to take an action, and nothing else (no DOM, no HTML code, nothing) so it's a truly multimodal browser demo.

Call your Browser

Finally, we wanted to push Spark's agentic multimodality to the limits, so we set up this space that combines the model's audio, image, and agentic properties into a browser agent. We give the model a general description of a thing we need and it interacts with the integrated browser to find options on Amazon.

Muse Spark Open takes the voice input and reasons over the type of product that meets the descriptions, searches for it on Amazon, and selects the most appropriate based on criteria.

Wrapping Up

Muse Spark Open is a full-size open-weight model from the Muse series. Start with the BF16 or NVFP4 checkpoint, or try one of the great llama.cpp quants . If you build something with Muse Spark Open, share it, and tell us in the repository's discussions.

Models mentioned in this article 4

Datasets mentioned in this article 1

Papers mentioned in this article 1

Collections mentioned in this article 1

More Articles from our Blog

mlx open-source community

Jun Kim, oMLX creator and maintainer, joins Hugging Face to support the MLX community

+1

98 September 22, 2026

llm moe long-context

DeepSeek-V4: a million-token context that agents can actually use

62 April 24, 2026

Community i wish if i can run huge models like this on a phone or 4vram </

```
p>\n","updatedAt":"2026-10-16T11:41:36.813Z","author":{"_id":"6a5971b7e134a9f01848f5dd","avatarUrl":"/avatars/216151cb669fecf6ced896eef5b5e039.svg","fullName":"ahmed alghra","name":"TDLord07","type":"user","isPro":false,"isHf":false,"isHfAdmin":false,"isMod":false,"isUserFollowing":false,"language":"en","probability":0.7960788607597351},"editors":["TDLord07"],"editorAvatarUrls":["/avatars/216151cb669fecf6ced896eef5b5e039.svg"],"reactions":[],"isReport":false}},"status":"open","isReport":false,"pinned":false,"locked":false,"collection":"community_blogs"},"cor
```

```
[{"pcuenq","merv","burtenshaw","guarin","qgallouedec","alvarobartt","Serveurperso","Jundot"},{"primaryEmailConfirmed":true,"withThread":true,"cardDisplay":false,"repoDiscussionsLocked":false,"hideComments":true}]>
```

TDLord07 2 days ago

i wish if i can run huge models like this on a phone or 4vram

Reply

Edit Preview

Upload images, audio, and videos by dragging in the text input, pasting, or clicking here . Tap or paste here to upload images

Comment · Sign up or log in to comment

Upvote 52

+40

Models mentioned in this article 4

Datasets mentioned in this article 1

Papers mentioned in this article 1

Collections mentioned in this article 1