From Ollama to vLLM

October 21, 2026 / Gavin Jackson

ai llm local-ai ollama vllm nvidia dgx-spark deepseek networking

My local AI experiments started with the RTX PRO 4000 Blackwell cards. I wanted something useful on a developer workstation: help with code, explanations of unfamiliar systems, and an assistant that could work inside a network where sending everything to a cloud API was not an option.

In my earlier Blackwell article, the recurring constraint was **24GB of VRAM**. Ollama made running the models straightforward. The interesting work was choosing a model and quantisation that fitted, leaving room for context, and connecting it to tools such as Hermes and Continue.

That setup taught me a pattern: get the model working directly, check the quality, then add the agent or IDE integration. When something broke, work back through those layers.

I took essentially the same approach to a **DGX Spark with 128GB of unified memory**, still using Ollama. Suddenly I could explore models roughly five times the size of my earlier workstation baseline. Then I connected **two Sparks with a single QSFP cable**, moved into vLLM, and started running **DeepSeek-V4-Flash-Vision-Exp**, a roughly 300-billion-parameter model.

And it is actually starting to look good.

That is my assessment of the responses so far. I still want repeatable measurements of speed, coding quality and tool use. But the progression from fitting a useful assistant onto a workstation card to serving something this large across two small computers has been quite a journey.

One Spark: the same pattern, much more room

The first Spark was a fairly comfortable transition. Pick a model, choose a sensible quant, load it into Ollama, and point the clients at it.

The big change was the memory budget. The Spark hardware guide lists **128GB of LPDDR5x unified system memory**, shared coherently by the CPU and GPU. Compared with the Blackwell card's 24GB, that is about **5.3 times the physical capacity**.

It gave me room to move from the smaller models I had been testing into the 70-120B range I discussed in my earlier Spark post. The exact increase in model size depends on quantisation and context; the fivefold comparison describes the scale of the opportunity, rather than a speed improvement.

The operating system uses that memory too. So do the inference engine, temporary buffers and conversation state. A 128GB Spark does not give a model 128GB exclusively for its weights.

Still, there was a pleasant period where I could reuse what I already knew and just try much larger models.

Then, naturally, I wanted another Spark.

Two Sparks: follow the bouncing ball

My starting points were NVIDIA's Connect Multiple DGX Sparks playbook and the intermediate vLLM cluster guide.

I followed the bouncing ball. Connect the machines, establish the cluster network, prepare the inference environment, and work towards a model endpoint.

The physical setup is surprisingly modest: two Sparks and one direct QSFP cable. NVIDIA documents this as a supported two-device layout, with a switch becoming an option as more devices join.

In my network checks, the dedicated link came up at **200,000Mb/s**, with the two ends reachable at `10.0.0.1` and `10.0.0.2`. That established basic connectivity. It did not establish that inference was using the best transport or that the model was correctly distributed.

That distinction becomes much more relevant once the network is part of generating every answer.

The other conceptual shift was moving beyond the single-machine Ollama setup I had been using. For this experiment, I needed a serving engine that could explicitly distribute one model across two machines. That brought me into **vLLM**.

What is vLLM actually doing?

vLLM is an open-source inference and serving engine. It loads a trained model, executes it on supported hardware, manages requests and memory, and provides an API that applications can use.

It does not add knowledge to the model. If the responses improve because I can now run a more capable model, that capability comes from the model. vLLM makes the deployment practical.

There are several different pieces to that practicality.

Splitting the model across the GPUs

With **tensor parallelism**, supported weight tensors are divided across GPUs. Both Sparks participate in the calculation, exchanging intermediate results as inference progresses. The model's entire weight set does not have to fit on either Spark individually.

With **pipeline parallelism**, groups of layers run in different stages. Work passes between those stages. This changes the communication pattern, but introduces pipeline scheduling and utilisation trade-offs.

These are established model-parallel techniques, described in NVIDIA's parallelism overview and supported by vLLM's distributed serving implementation. A recipe chooses a layout that its model implementation supports; adding another computer does not make every possible split valid.

For a two-Spark tensor-parallel deployment, each GPU holds a share of the model, along with whatever state and buffers that implementation needs. Some data can be replicated, and the split need not consume precisely equal memory.

The useful consequence is that a model too large for one Spark can become possible across the pair.

Coordinating the workers

There is a separate question: how do the processes find each other and start working together?

Some vLLM deployments use **Ray** to manage distributed workers. Current versions also support native multi-node multiprocessing. NVIDIA's current GLM recipe uses the native route, so a separate Ray cluster is not a universal requirement.

The relevant native-launch settings appear in vLLM's command reference:

Setting	Role
`--tensor-parallel-size 2`	Divide supported tensors across two GPU ranks.
`--nnodes 2`	Participate in a two-node deployment.
`--node-rank 0` / `1`	Give each node a distinct identity.
`--master-addr` / `--master-port`	Tell the processes where to rendezvous.
`--headless`	Run the worker without its own public API server.

These are pieces of a complete recipe. The checkpoint, container, kernels and network settings still need to match.

Worker coordination also has a different job from moving inference data. GPU collectives commonly use **NCCL**, while particular Spark recipes add specialised RoCE communication paths. Ray being healthy, or both processes starting successfully, does not prove that those transfers are efficient.

Using conversation memory efficiently

Generating a response involves more than reading the weights. The engine also retains attention state for tokens already processed: the **KV cache**.

The original vLLM PagedAttention explanation describes managing that cache in blocks, allowing logical token sequences to use non-contiguous physical memory. This reduces the waste associated with reserving large contiguous areas for conversations whose eventual lengths are unknown.

The broad benefit is better use of the available memory, especially when serving several requests. Modern models have different attention and recurrent-state designs, so the actual cache layout and kernels depend on the architecture.

For my purposes, the practical lesson survives all those details: **context has a memory cost even after the model has loaded**.

Scheduling work and reusing it

vLLM also supports continuous batching: active requests can share execution batches as requests arrive and finish. Chunked prefill divides long prompt processing into smaller pieces, helping the scheduler balance that work with ongoing generation. These are among the engine's documented serving features.

Prefix caching can reuse computed state when requests begin with the same token prefix. An agent repeatedly sending the same instructions or repository background is an obvious candidate. It mainly saves prompt-processing work; it does not eliminate the cost of generating the answer.

Some recipes also use speculative decoding, proposing several tokens and checking them with the target model. A proposal mechanism can use a separate draft model or the model's own multi-token prediction components. Whether it helps depends on acceptance rates and execution overhead.

This helps explain why a working launch command can be much longer than `ollama run`. There are more decisions exposed to the person operating the service.

How does a 300B model fit?

The model I am running is DeepSeek-V4-Flash-Vision-Exp. Its Hugging Face page lists approximately **305B parameters**. This is the experimental multimodal release, which adds visual modules and continued training to the Flash architecture.

Start with the weight arithmetic. Using **305 billion parameters** as a rough sizing figure, the idealised payload is:

Weight precision	Bytes per parameter	Weight payload
BF16 / FP16	2	610GB
8-bit	1	305GB
4-bit	0.5	152.5GB

These are decimal GB calculations, before quantisation scales, higher-precision tensors, caches, workspaces and system overhead. Real checkpoints mix precisions and have their own storage layouts.

Two Sparks provide **256GB of installed memory in aggregate**. A suitable four-bit checkpoint therefore has a plausible path to fitting. Its actual shards and runtime allocations must each fit within an individual Spark's budget.

The network does not create a coherent 256GB memory pool spanning the two computers. The serving engine distributes the model and exchanges the data needed to execute it.

Mixture-of-Experts models add another distinction: **total parameters versus parameters active for each token**. Routing selects a subset of experts, reducing computation relative to activating the entire model. For resident inference, those other experts still need storage.

The Vision-Exp release makes this especially interesting for the work I do. Alongside text and code, visual understanding opens up experiments with screenshots, diagrams and documentation. Those are capabilities I want to evaluate; my encouraging first impressions do not establish how well it handles each of them. Image processing also adds to the runtime memory budget.

The combination is what makes this interesting: a large pool of model weights, sparse computation, quantisation, and an engine that can divide the work between machines.

The cable is part of the computer now

A 200Gb/s link has a theoretical line rate of **25GB/s**, before protocol overhead. NVIDIA lists the Spark's local memory bandwidth at **273GB/s**. Those figures describe different paths, but they illustrate why moving data between boxes has to be treated carefully.

Every exchange has bandwidth and latency costs. For repeated small collectives, latency can matter as much as the headline link rate.

The checks I want around a deployment now go beyond a successful ping:

1. Verify the negotiated link speed and interface error counters.
2. Identify the ConnectX interfaces and RoCE devices that the recipe expects.
3. Confirm the inference transport is using the intended high-speed path.
4. Run communication tests before using a large model to diagnose the network.

NVIDIA's NCCL playbook provides communication tests for Spark topologies. The vLLM transport guidance also explains how NCCL logs can reveal a TCP socket fallback. Recipe-specific RoCE kernels need their own checks too.

One physical QSFP connection can expose multiple logical interfaces. That makes guessing from a device name an unreliable way to decide which interface should be used or disabled. I want the documented mapping in front of me before changing it.

What I want to run next

My next substantial test is NVIDIA's GLM-5.3-Flash recipe for two 128GB Sparks.

There are good alternatives worth comparing alongside it. My shortlist is based on documented deployment paths and the jobs I want these machines to do, rather than an assertion that one model wins every benchmark.

Candidate	Why it is on my list	Deployment evidence
GLM-5.3-Flash FP4-Local	My first choice for the next large coding and agent test. The upstream model has 320B total parameters and 18B active.	NVIDIA provides the exact two-128GB-Spark recipe linked above.
Qwen3.8-Flash-Next FP4-Local	A coding, tool-use and multimodal comparison; also useful for testing whether a smaller deployment is sufficient.	The NVIDIA checkpoint card documents Spark use, and NVIDIA's cluster guide links a two-Spark recipe.
Nemotron 3 Super 120B-A12B NVFP4	A more manageable agent and reasoning baseline, with 120B total parameters and 12B active.	The official model instructions include a single-Spark vLLM configuration. Two machines could run independent instances.
DeepSeek-V4-Flash-Vision-Exp	My existing roughly 305B multimodal point of comparison: promising responses, with proper task evaluation still to do.	Running across my two Sparks. Reproducing that deployment also requires the particular quantised weights and runtime settings.

For Qwen in particular, the model card lists a 125B main model plus additional n-gram embeddings and MTP parameters, and a checkpoint around 96GiB on disk. A headline parameter count alone is a poor guide to the deployed memory footprint.

My order would be GLM first, then Qwen and Nemotron on the same tasks. If a model running on one Spark produces comparable results, that leaves the second machine available for another user or workload.

Why the precise GLM recipe matters

The download I want is `nvidia/GLM-5.3-Flash-FP4-Local`. NVIDIA's checkpoint card describes quantisation-aware distillation and mixed tensor precisions. It also explicitly requires a matching community vLLM container while support for its format is being integrated upstream.

That is a stronger requirement than having four-bit weights and a recent generic container. The loader and kernels must understand that particular checkpoint.

The selected recipe currently pins `eugr/spark-vllm-b12x:nightly-20261015`, uses tensor parallelism across two nodes, FP8 KV cache, and native MTP speculative decoding. Its configured request limit is **500,000 tokens** with up to four sequences; those settings are a reference configuration, not evidence that every workload at those limits will be comfortable.

NVIDIA reports **33.94 tokens/sec decode throughput** for its SPEED-Bench qualitative coding workload, measured on **16 October 2026** on two 128GB Sparks. That result belongs to the published recipe and workload; it is a useful reference for my next experiment.

I would start with a shorter context and a single request, then increase the workload while watching memory and latency.

Getting back to a normal chat interface

After all that machinery, the application-facing part becomes familiar again. vLLM provides an OpenAI-compatible serving API, including model discovery and chat completions.

For example, from the head machine, once the selected recipe is running:

```
curl http://localhost:8000/v1/models

curl http://localhost:8000/v1/chat/completions \
 -H 'Content-Type: application/json' \
 -d '{
 "model": "nvidia/GLM-5.3-Flash-FP4-Local",
 "messages": [
 {"role": "user", "content": "Explain tensor parallelism using two DGX Sparks."}
 ],
 "max_tokens": 512,
 "stream": true
 }'
```

The model name must match what `/v1/models` reports. A recipe can publish an alias instead of its Hugging Face repository name.

A chat UI or agent that supports a custom OpenAI-compatible endpoint can then use the head node's `/v1` base URL. For coding agents, the model-specific chat template, reasoning parser and tool-call parser are part of making that integration work properly.

I want to repeat the same testing pattern that worked with Ollama: verify the direct API first, then connect the clients.

Spinning Up vLLM for Browser Users

Everything so far ends at an OpenAI-compatible API on the head node. Turning that into something an end user can reach from a web browser takes two pieces: the server has to listen on an address the LAN can reach, and a browser chat interface has to point at that address.

On a single machine, the serving side is one command:

```
vllm serve nvidia/GLM-5.3-Flash-FP4-Local \
 --host 0.0.0.0 \
 --port 8000
```

In the two-Spark recipes, the serving step is part of the recipe itself, and only the head node publishes the API. The second node runs `--headless`, so there is nothing to connect to on it. Users always point at the head node's regular LAN address, not the `10.0.0.x` interconnect, which is reserved for inference traffic between the Sparks.

For the browser side, [Open WebUI]() is the quickest self-hosted chat interface I have found. It talks to OpenAI-compatible endpoints and runs as a single container:

```
docker run -d --name open-webui --network host \
 -e OPENAI_API_BASE_URL=http://<head-node>:8000/v1 \
 -e OPENAI_API_KEY=unused \
 -v open-webui:/app/backend/data \
 ghcr.io/open-webui/open-webui:main
```

Browse to `http://<open-webui-host>:8080` and register; the first account created becomes the administrator. The model picker is populated from `/v1/models`, so the name shown matches whatever the recipe publishes, which may be an alias rather than the Hugging Face repository name.

Setting	***Value***
vLLM API base URL	`http://<head-node>:8000/v1`
API key	*vLLM accepts any key unless `--api-key` is set*
Open WebUI default port	`8080`

> *A few operational notes:*
>
> - *vLLM itself has no user accounts. Setting `--api-key` gives one shared secret, not per-user identity; per-user logins and separate chat histories come from Open WebUI. For anything beyond a home LAN, I would put the endpoint behind a reverse proxy that handles authentication properly.*
> - *If the served model needs a specific reasoning parser or tool-call parser, those options belong on the serving command. Without them, streamed thinking blocks and tool calls will not render the way the UI expects.*
> - *LibreChat and LobeChat follow the same pattern: set the base URL and key, pick the model, chat.*

Once the 200Gb/s switch arrives

A switch gives us a more convenient way to connect additional Sparks. The useful specification is **200Gb/s Ethernet per Spark-facing connection**, with compatible cables or optics. QSFP describes a connector family; the connector shape alone does not establish protocol or link speed.

NVIDIA has a multiple-Sparks-through-a-switch playbook. Moving to that topology changes addressing, interface selection and communication paths, so the direct-cable setup needs to be revalidated.

The memory arithmetic becomes appealing:

128GB Sparks	Installed memory in aggregate	What I would explore
Two	256GB	The current 300B-class experiment and the GLM comparison.
Three	384GB	A supported three-way model split, or a two-node model plus a separate service.
Four	512GB	Larger quantised models, two independent two-node deployments, or several single-node services.

Three-way tensor parallelism is not guaranteed to be supported by a given architecture. Four nodes are not guaranteed to be twice as fast as two. Those are measurements to make after establishing a supported configuration.

As a capacity estimate, **671B parameters at an idealised four bits need about 335.5GB for weights alone**. Four Sparks make that class of experiment worth investigating, subject to a suitable checkpoint, supported partitioning and memory headroom. A 1.6T model would need around 800GB even at that idealised precision, beyond four Sparks' installed capacity.

These calculations identify experiments to investigate. They do not establish a working launch recipe.

More machines give me several useful choices

One option is to spread a larger model across all the nodes. Another is to run replicas of a model that already fits, increasing the number of requests we can serve independently. A third is to give different services their own machines: coding, chat, embeddings or reranking.

For MoE models, expert parallelism offers another layout by placing experts on different GPUs. Its routing and communication requirements make it another configuration to benchmark, rather than an automatic improvement.

My first switch-connected milestone would be to reproduce the current two-node behaviour through the switch. Then add a node, validate communications, and compare supported layouts. Under load, I would also check the switch and NIC congestion counters using NVIDIA's NCCL networking guidance, including the relevant RoCE configuration for the actual switch.

For the models themselves, I want to record:

- **Time to first token**, including longer prompts.
- **Generation speed for one user**, and total throughput with several users.
- **Memory use and stable context limits** on every node.
- **Task success** on a fixed set of coding, Ansible, Linux troubleshooting and tool-use exercises.
- **Exact model revision, quantisation, container and launch settings**, so the comparison can be repeated.

That last part is becoming essential. A model name is no longer enough to describe the system that produced an answer.

A small cluster with a useful job

The Blackwell workstations established that local AI could be useful inside an ordinary developer workflow. One Spark let me take the same Ollama patterns into much larger models. Two Sparks have pushed me into distributed inference and made DeepSeek-V4-Flash-Vision-Exp a practical experiment.

There is more to operate now: workers, network transport, memory budgets, quantised checkpoints and kernels that need to agree with one another. Understanding those pieces is becoming as interesting as trying the next model.

GLM-5.3-Flash is next on my list. After that, the switch should give us room to find out whether the best use of more Sparks is a larger model, more simultaneous users, or a few specialised services.

The encouraging part is that the responses are already making me want to use the system. Now I want the measurements to catch up with that impression.

Sources and further reading

Technical recipes and model details checked on **21 October 2026**. Links in the relevant sections identify the source of specifications and reported results.

- My earlier RTX PRO 4000 Blackwell article
- My earlier DGX Spark article
- NVIDIA: connect multiple DGX Sparks
- NVIDIA: intermediate vLLM cluster guide
- DeepSeek-V4-Flash-Vision-Exp model card
- NVIDIA: GLM-5.3-Flash on two 128GB Sparks
- vLLM: parallelism and scaling
- NVIDIA: Spark NCCL tests
- NVIDIA: multiple Sparks through a switch

Downloaded from https://www.gavinj.net/post/from-ollama-to-vllm
Generated October 23, 2026. Copyright Gavin Jackson. All rights reserved.