

what are the pros/cons of running on Ubuntu 22.04 system with two NVIDIA 3090 GPUs either (a) ollama or (b) vLLM or (c) Both (if possible)

◆ Show thinking ▾

Running Large Language Models (LLMs) on a system with two NVIDIA 3090 GPUs on Ubuntu 22.04 provides a powerful local inference setup. Choosing between Ollama, vLLM, or a combination depends heavily on your specific use case, priorities, and technical expertise. Here's a detailed breakdown:

System Configuration Notes

- **Ubuntu 22.04:** A stable and widely supported Linux
 - **"Just Works" Philosophy:** Ollama aims to simplify the process of downloading, running, and managing LLMs. Installation is typically straightforward, and models can be run with single commands (e.g., `ollama run llama3`).
 - **Unified Interface:** Provides a consistent API and CLI for interacting with various models.
 - **Model Management:** Easy to discover, pull, and manage different open-source models (Llama2, Llama3, Gemma, etc.) from its library.
 - **Modelfiles:** Allows for easy customization and fine-tuning of models using a simple `Modelfile` syntax.
- 2. **Broad Hardware Compatibility:** While optimized for GPUs, it also runs well on CPUs, offering flexibility. Your 3090s will be fully utilized.
- 3. **Local and Private:** All inference happens locally, ensuring data privacy and security without relying on cloud services.
- 4. **Community & Ecosystem:** Growing community support and integration with frontends like Open WebUI, making it easy to create a user-friendly interface.
- 5. **Multi-GPU Support (Implicit/Basic):** Ollama primarily uses workload.
- 6. **Structured Outputs and Tool Use:** Recent advancements allow for constrained JSON schema outputs and tool integration, enhancing its utility for RAG and agentic workflows.

Cons:

1. **Performance (Relative to vLLM):**
 - **Throughput:** While Ollama has made significant performance improvements, it generally offers lower throughput and higher latency compared to vLLM, especially for high-volume concurrent requests. Its memory management is less optimized than vLLM's PagedAttention.
 - **Concurrency:** Ollama is designed more for single-user or low-concurrency scenarios. While it has an API and can handle multiple requests, high concurrent loads can lead to increased RAM requirements (each request has its own context) and longer waiting times. Running multiple models concurrently can also quickly overwhelm a single GPU.
6. **Structured Outputs and Tool Use:** Recent advancements allow for constrained JSON schema outputs and tool

integration, enhancing its utility for RAG and agentic workflows.

Cons:

1. **Performance (Relative to vLLM):**
 - **Throughput:** While Ollama has made significant performance improvements, it generally offers lower throughput and higher latency compared to vLLM, especially for high-volume concurrent requests. Its memory management is less optimized than vLLM's PagedAttention.
 - **Concurrency:** Ollama is designed more for single-user or low-concurrency scenarios. While it has an API and can handle multiple requests, high concurrent loads can lead to increased RAM requirements (each request has its own context) and longer waiting times. Running multiple models concurrently can also quickly overwhelm a single GPU.
2. **Less Granular Control:** Prioritizes simplicity over deep optimization. You have less fine-grained control over GPU memory allocation, batching strategies, and advanced inference techniques compared to vLLM.
3. **Memory Management:** While it can utilize multiple GPUs, its memory management for the Key-Value (KV) cache is not as sophisticated as vLLM's PagedAttention, which can lead to less efficient GPU memory utilization, especially with long contexts or high concurrency.

(b) Running with vLLM

vLLM is an inference engine specifically designed for high-throughput and low-latency LLM serving in production environments.

Pros:

1. **Superior Performance (Throughput & Latency):**
 - **PagedAttention:** This is vLLM's key innovation. It's a highly optimized memory management algorithm for the KV cache that drastically reduces memory fragmentation and overhead. This allows vLLM to serve significantly more concurrent requests and handle longer sequences with greater efficiency. It can achieve orders of magnitude higher throughput than other frameworks.
 - **Continuous Batching:** vLLM dynamically batches incoming requests, even if they arrive at different times. This maximizes GPU utilization by keeping the GPUs busy and reduces latency by avoiding idle time.
 - **Optimized CUDA Kernels:** Includes highly optimized CUDA kernels for various operations, leveraging the NVIDIA 3090's architecture efficiently.
2. **Distributed Inference (Multi-GPU & Multi-Node):**
 - **Tensor Parallelism:** vLLM inherently supports tensor parallelism, allowing it to split large models across multiple GPUs (like your two 3090s) seamlessly. This means you can run models that would otherwise not fit into a single 24GB GPU, making full use of your 48GB VRAM.
 - **Pipeline Parallelism:** Also supports pipeline parallelism for even larger models or complex serving scenarios.
3. **Production-Ready:** Built for scale, high concurrency, and robustness, making it ideal for serving LLMs in production applications.

4. **OpenAI-Compatible API:** Provides an API that is largely compatible with OpenAI's API, making it easy to integrate with existing applications or frontends.
5. **Efficient Memory Utilization:** Due to PagedAttention, vLLM makes the most out of your available GPU memory, allowing you to run larger batch sizes or more concurrent requests.

Cons:

1. **Higher Learning Curve & Setup Complexity:**
 - **More Configuration:** Requires more setup and configuration knowledge compared to Ollama.
 - **Dependency Management:** While installation can be done via pip or Docker, setting up for distributed inference or specific optimizations might require a deeper understanding of its architecture and dependencies.
2. **GPU-Centric:** Primarily optimized for CUDA-enabled GPUs. While it has some CPU support, its core benefits are realized on powerful GPUs like your 3090s.
3. **Less "Batteries-Included" for Model Management:** Unlike Ollama, which has a built-in model hub and easy `ollama run` commands, vLLM typically requires you to manage model weights (e.g., from Hugging Face) and their configurations more explicitly.
4. **Memory Pre-allocation (Potential Disadvantage):** While PagedAttention optimizes memory usage *during* inference, it might pre-allocate significant GPU memory up to the maximum sequence length, which can be a consideration for extremely constrained environments (though unlikely to be an issue with two 3090s for most use cases).

(c) Running Both (If Possible)

It is absolutely possible to run both Ollama and vLLM on the same Ubuntu 22.04 system with two 3090 GPUs, though not necessarily on the exact same GPUs simultaneously for serving the same model or handling overlapping workloads.

How it could work:

1. **Port Separation:** Ollama typically runs on port `11434` and vLLM on `8000` (by default). They can coexist as separate services.
2. **GPU Allocation:**
 - **Dedicated GPUs:** You could dedicate one 3090 to Ollama and the other to vLLM. This would allow them to run simultaneously without resource contention, assuming the models they are serving fit within the respective GPU's VRAM.
 - **Shared GPUs (Carefully):** You could run one service, then stop it, and run the other. Running them truly simultaneously on the same GPUs would lead to significant resource contention (especially VRAM), potentially crashing one or both services due to out-of-memory errors or extremely poor performance.
 - **Model-specific routing:** If you're building a larger system, you could use an API gateway or load balancer to route requests to either Ollama or vLLM based on the model requested or the nature of the request (e.g., Ollama for smaller, interactive models, vLLM for high-throughput production models).
3. **Containerization (Docker):** This is highly recommended for running both. You can containerize Ollama and vLLM in separate Docker containers. This provides isolation and easier management. You can control which GPUs each

container has access to using Docker's `--gpus` flag (e.g., `--gpus device=0` for one GPU, `--gpus device=1` for the other).

Pros of Running Both:

1. **Flexibility and Hybrid Approach:** This is often the most pragmatic solution in a complex AI setup.
 - **Development & Prototyping:** Use Ollama for rapid prototyping, local development, and quick experimentation with new models due to its ease of use.
 - **Production Serving:** Deploy vLLM for production workloads that demand high throughput, low latency, and efficient GPU utilization.
 - **Model Tiering:** Serve smaller, less performance-critical models with Ollama and larger, high-demand models with vLLM.
2. **Leverage Strengths:** You get the best of both worlds: Ollama's user-friendliness for local interactive use and vLLM's raw performance for backend serving.
3. **Research and Benchmarking:** Allows for direct comparison and benchmarking of different models and serving strategies on your hardware.

Cons of Running Both Simultaneously (on shared GPUs):

1. **Resource Contention:** VRAM is a finite resource. If both try to load large models onto the same GPUs, you'll encounter out-of-memory issues or severe performance degradation. This is less of an issue if you dedicate one GPU to each, but still requires careful resource planning.
2. **Increased Complexity:** Managing two separate inference servers adds overhead in terms of configuration, monitoring, and debugging.
3. **Driver & CUDA Compatibility:** While Ubuntu 22.04 and NVIDIA 3090s are well-supported, ensuring the correct NVIDIA drivers and CUDA toolkit versions are compatible with both Ollama (or its underlying `llama.cpp` version) and vLLM can sometimes be a minor challenge. Using Docker can help mitigate some of these dependency issues.

Conclusion and Recommendation for Your Setup:

With two NVIDIA 3090 GPUs (48GB total VRAM) on Ubuntu 22.04, you have a very capable system.

- **For quick local experimentation, ease of use, and personal projects:** Ollama is an excellent choice. It will generally make good use of your GPUs, especially for models up to 70B parameters, distributing them if needed.
- **For production-grade serving, high concurrency, and maximum throughput:** vLLM is the clear winner. Its PagedAttention mechanism and continuous batching will significantly outperform Ollama in these scenarios, making the most of your dual 3090s. This is crucial for applications that need to serve many users simultaneously or process large batches of requests.
- **For a versatile workstation or development environment that can transition to serving:** A hybrid approach (c) is likely the most powerful.
 - Use Docker to containerize both.
 - For testing or light interactive use, perhaps let Ollama use all available GPUs.
 - For serving, launch vLLM with tensor parallelism across both GPUs.
 - If you need to run both at the exact same time and serve

ditterent models, carefully allocate one 3090 to each service within their respective Docker containers to avoid VRAM conflicts.

Given your powerful hardware, the ability of vLLM to efficiently leverage multiple GPUs with tensor parallelism for very large models (e.g., Llama 70B) makes it particularly appealing for pushing the limits of what you can run locally. Ollama, while great for usability, won't achieve the same level of performance optimization for high-concurrency scenarios. For serious AI development or serving, investing the time to set up vLLM correctly will yield significant performance dividends.

 Sources

↶ ↷ ⏪ ⏵ ⏴ ⏵

How much faster is vLLM than ollama in tokens/sec given a LLM that fits within a RTX 3090 24GB VRAM... ▾

◆ Show thinking ▾

The difference in tokens/second between vLLM and Ollama, especially when the LLM fits within a single RTX 3090's 24GB VRAM, can be significant, particularly under concurrent load. It's not a fixed multiplier, as it depends on several factors:

Key Factors Influencing Performance Differences:

1. **Concurrency/Batching:** This is the most crucial factor.
 - **vLLM (with PagedAttention & Continuous Batching):** vLLM excels at handling multiple simultaneous requests (high concurrency) and dynamic batching. Its PagedAttention algorithm efficiently manages the Key-Value (KV) cache, minimizing memory waste and allowing more requests to be processed in parallel. This is where vLLM's throughput truly shines.  
 - **Ollama (underlying llama.cpp):** Ollama is built on `llama.cpp`. While `llama.cpp` is highly optimized for single-stream performance and efficient CPU/GPU offloading, it doesn't have the same advanced memory management (like PagedAttention) or continuous batching capabilities as vLLM. For a single, isolated request, their speeds might be closer, but as you scale up concurrent users or requests, Ollama's performance will degrade much more rapidly due to KV cache fragmentation and less efficient GPU utilization. 

2. Model Size and Quantization:

- For a model that *just* fits within 24GB (e.g., a 70B parameter model with 4-bit quantization), vLLM's memory efficiency can still allow for larger batch sizes or longer context windows, leading to higher overall throughput.
- For smaller models (e.g., 7B or 13B parameters that fit easily), the difference might be less pronounced in a single-request scenario, but the advantages of vLLM's batching still hold under load.

3. Prompt and Generation Lengths:

- Longer prompts and generated responses benefit more from vLLM's PagedAttention, as it reduces KV cache memory fragmentation for sequences of varying lengths. 

4. Backend Optimizations:

- vLLM leverages highly optimized CUDA kernels, often including techniques like Flash Attention, which can provide significant speedups for the attention

mechanism on NVIDIA GPUs. ↵

- `llama.cpp` also has impressive CUDA optimizations, but vLLM's architecture is specifically tailored for serving and maximizing throughput. ↵

Estimated Performance Difference (Tokens/Sec):

Single Request / Low Concurrency:

For a single, isolated request, the difference might be **1.5x to 2x faster** for vLLM compared to Ollama (or directly using `llama.cpp` which Ollama wraps). Some benchmarks show `llama.cpp` to be around 89 tokens/sec and `vLLM` around 84 tokens/sec for `llama3.1:8b Q4` in single-request scenarios, implying that for specific configurations and models, the difference might be less significant or even slightly in favor of `llama.cpp` for single-request, if `vLLM` incurs more overhead for its advanced features when only serving one client. However, this is for a specific model and quantization. Other sources contradict this, showing `llama.cpp` at 89 tokens/sec and `llama.cpp` directly (without Ollama's wrapper overhead) at 161 tokens/sec for a quantized model, making it 1.8x faster than Ollama in some cases. It's important to note that Ollama itself might add some overhead compared to raw `llama.cpp`.

High Concurrency / Production Workloads:

This is where vLLM truly differentiates itself. When running multiple concurrent requests or large batch sizes, vLLM can be **several times to over an order of magnitude faster** in terms of *aggregate tokens/second* (total tokens processed per second across all requests).

- **2x-5x+ faster:** For a moderate number of concurrent requests.
- **10x-24x higher throughput:** The vLLM paper itself (and subsequent benchmarks) claimed 14x-24x higher throughput than HuggingFace Transformers and 2.2x-3.5x higher than HuggingFace Text Generation Inference (TGI) for Llama-7B on an A10G. While these numbers are not directly comparable to Ollama on a 3090, they illustrate the *potential* for massive gains from vLLM's architectural optimizations under load.

Why the Discrepancy?

The primary reason for vLLM's superior performance in concurrent scenarios is **PagedAttention**. Traditional KV cache management (as used by `llama.cpp` and thus Ollama) pre-allocates memory for the maximum possible sequence length for each request. This leads to:

- **Memory Fragmentation:** Gaps in GPU memory that cannot be used.
- **Wasted Memory:** Even if a request only uses a short context, the full maximum length is reserved.

PagedAttention, inspired by virtual memory in operating systems, allocates KV cache in fixed-size "blocks" on demand. This allows:

- **Elimination of external fragmentation:** Any free block can be used.
- **Minimization of internal fragmentation:** Only the last block might have wasted space.
- **Memory Sharing:** Multiple requests can share the same prompt prefix KV cache blocks, which is highly efficient for applications like chatbots where many users might start with similar initial prompts.
- **Continuous Batching:** Requests are processed as soon as they arrive, maximizing GPU utilization rather than waiting for a full batch to accumulate. ↵

These benefits translate directly into a much higher number of active sequences (requests) that can fit into the GPU's memory and be processed simultaneously, drastically increasing the aggregate tokens/second output from the system.

In summary:

- If you're primarily running **one prompt at a time** for local, **interactive use**, the speed difference might be noticeable but not astronomical (perhaps **1.5x - 2x** or even closer depending on the specific model and prompt/generation length).
- If you intend to serve the LLM to **multiple users concurrently, perform batch inference, or handle long, complex interactions**, vLLM will demonstrate a **significantly higher tokens/second throughput**, potentially by a factor of **5x or more**, making it the unequivocal choice for such use cases on your RTX 3090.

 Sources

I currently have the most recent ollama version configured and running on Ubuntu 22.04 with two...

◆ Show thinking ▾

You're in a great position with two RTX 3090s and Ubuntu 22.04 for a powerful local LLM setup. Running both Ollama and vLLM in Docker containers is an excellent strategy for flexibility, leveraging each tool's strengths while maintaining a clean host system.

Here's a step-by-step guide to reconfigure your system to use both Ollama and vLLM within Docker containers, including how to manage GPU allocation:

Prerequisites:

1. **NVIDIA Drivers:** Ensure you have the latest stable NVIDIA drivers installed on your Ubuntu 22.04 system. You mentioned Ollama is already running, so this is likely already set up. You can check with `nvidia-smi`.
2. **Docker:** Install Docker Engine and Docker Compose (if not already installed).
 - **Install Docker Engine:**

```
Bash

# Add Docker's official GPG key:
sudo apt update
sudo apt install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) sign
  "$(. /etc/os-release && echo "$VERSION_CODENAME"
  sudo tee /etc/apt/sources.list.d/docker.list >
  sudo apt update
  sudo apt install docker-ce docker-ce-cli contain
```

- **Manage Docker as a non-root user (recommended):**

```
Bash

sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker # You might need to log out and ba
```

3. **NVIDIA Container Toolkit:** This is essential for Docker containers to access your NVIDIA GPUs.

```
Bash

distribution=$( . /etc/os-release;echo $ID$VERSION_ID
&& curl -fsSL https://nvidia.github.io/libnvidia-
&& curl -s -L https://nvidia.github.io/libnvidia-
sed 's#deb https://#deb [signed-by=/usr/share/key-
sudo tee /etc/apt/sources.list.d/nvidia-container-
sudo apt update
sudo apt install -y nvidia-container-toolkit
sudo systemctl restart docker
```

Verification: Run `docker run --rm --gpus all nvidia/cuda:12.4.0-base-ubuntu22.04 nvidia-smi` to ensure Docker can see your GPUs. You should see `nvidia-smi` output.

Step-by-Step Reconfiguration:

We'll use `docker-compose.yaml` to define and run both services. This makes managing them much easier.

1. Stop and Remove Existing Ollama Installation (if not already in Docker)

If your current Ollama installation is *not* in Docker, it's best to remove it to avoid conflicts and free up resources.

```
Bash

# If you installed with the official script:
sudo systemctl stop ollama
sudo systemctl disable ollama
sudo rm /etc/systemd/system/ollama.service
sudo rm /usr/local/bin/ollama
sudo rm -rf ~/.ollama # This removes models and data! B
```

If you had it running in a `docker run` command, stop and remove that container:

```
Bash

docker stop ollama_container_name # Replace ollama_
docker rm ollama_container_name
```

2. Create Project Directory

Create a directory for your Docker Compose setup:

```
Bash

mkdir ~/llm_docker_setup
cd ~/llm_docker_setup
```

3. Prepare `docker-compose.yaml`

This `docker-compose.yaml` will define two services: `ollama` and `vllm`. You have two GPUs, typically indexed `0` and `1`. You can choose to:

- **Dedicate one GPU to each:** (`device=0` for Ollama, `device=1` for vLLM). This ensures no VRAM contention if you run them simultaneously, but limits each service to 24GB.
- **Allow both to use all GPUs:** (`devices: ["all"]` for both). This means they will contend for resources if running simultaneously. For vLLM, `tensor-parallel-size 2` will use both GPUs efficiently. For Ollama, it will use both if the model is large enough or configured, but might not be as efficient as vLLM for parallelism.
- **Run one at a time, but enable multi-GPU for each when active.** This is often the most practical, using the `--gpus all` strategy, and simply starting the one you need at the moment.

Let's go with the **hybrid approach** where you define both and can choose to start them independently or carefully manage resources if running concurrently. We'll set up `vllm` to use

`tensor-parallel-size 2` by default for your 3090s. For Ollama, we'll let it use all available GPUs and rely on its internal `llama.cpp` multi-GPU logic.

Create a file named `docker-compose.yaml` in your

`~/llm_docker_setup` directory:

```
YAML
version: '3.8'

services:
  ollama:
    container_name: ollama_server
    image: ollama/ollama:latest
    ports:
      - "11434:11434" # Ollama API port
    volumes:
      - ollama_models:/root/.ollama # Persistent volume
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all # Ollama will detect and use a
              capabilities: [gpu]
    restart: unless-stopped
    environment:
      # Optional: You can explicitly set GPU count, but
      # - OLLAMA_GPU_COUNT=2
      # Optional: Limit VRAM usage for Ollama if needed
      # - OLLAMA_GPU_MEMORY_LIMIT=20GB

  vllm:
    container_name: vllm_server
    image: vllm/vllm-openai:latest # Official vLLM image
    ports:
      - "8000:8000" # vLLM API port (OpenAI compatible)
    volumes:
      # Mount a host directory for Hugging Face models
      - ~/llm_models_cache:/root/.cache/huggingface
      # If you have models stored elsewhere, add more mounts
      # - /path/to/your/local_models:/app/models
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all # vLLM will use all available
              capabilities: [gpu]
    restart: unless-stopped
    command: >
      python3 -m vllm.entrypoints.api_server
      --host 0.0.0.0
      --port 8000
      --model mistralai/Mistral-7B-Instruct-v0.2 # Required
      --tensor-parallel-size 2 # Crucial for using both
      --gpu-memory-utilization 0.9 # Adjust based on your
      # --max-model-len 4096 # Adjust max context length
      # --dtype bfloat16 # Use bfloat16 for better performance
      # --trust-remote-code # Required for some models

  volumes:
    ollama_models: # Docker volume for Ollama's persistent
```

Important Considerations for the `docker-compose.yaml`:

- `vllm_models_cache` **volume:** Change `~/llm_models_cache` to a path on your host system where you want Hugging Face models to be downloaded and cached. This avoids re-downloading models if you restart the container or rebuild the image.
- `--model mistralai/Mistral-7B-Instruct-v0.2`: You **MUST change this to the Hugging Face ID of the model you want vLLM to serve.** For a 70B model like Llama-2-70b or Llama-3-70b, ensure you specify the correct path or Hugging Face ID. For example, `meta-llama/Llama-2-70b-chat-hf` (requires Hugging Face token for gated models) or a specific quantized version if you plan to use one. You might need to download the model weights to

`~/vllm_models_cache` beforehand for larger models or if you don't want vLLM to download them at startup.

- `--tensor-parallel-size 2`: This tells vLLM to distribute the model across both your RTX 3090 GPUs. This is critical for larger models that might not fit on a single 24GB card (e.g., a 70B model in 4-bit precision typically needs ~40GB VRAM).
- `--gpu-memory-utilization 0.9`: This controls how much of the GPU memory vLLM will try to use for its KV cache and model weights. 0.9 (90%) is a good starting point. Adjust based on your specific models and if you experience OOM errors.
- `OLLAMA_GPU_COUNT / OLLAMA_GPU_MEMORY_LIMIT`: For Ollama, `count: all` usually works well, allowing it to leverage both GPUs if needed for larger models (via `llama.cpp`'s offloading). You can uncomment and set these environment variables if you want more explicit control, e.g., `OLLAMA_GPU_COUNT=1` if you want it to only use one GPU, leaving the other for vLLM, or `OLLAMA_GPU_MEMORY_LIMIT=20GB` to reserve some VRAM.

4. Download a Model for Ollama (Optional, but recommended for testing)

You can pull a model after Ollama starts, or if you want to ensure it's ready, you can manually pull a model (e.g., Llama3 8B) into your Ollama volume.

To pull a model:

```
Bash
# Start just the Ollama service
docker compose up -d ollama

# Wait a moment for it to start. Then, execute into the
docker exec -it ollama_server ollama pull llama3 # Or a

# Once pulled, you can stop ollama if you want to run v
# docker compose stop ollama
```

The `ollama_models` named volume will persist your downloaded models across container restarts.

5. Start the Services

Now you can start your services.

Option A: Start both (careful with VRAM if using very large models on both simultaneously)

```
Bash
docker compose up -d
```

This will start both `ollama_server` and `vllm_server` in detached mode.

Option B: Start only one service at a time (recommended for focused testing or resource management)

To start only Ollama:

```
Bash
docker compose up -d ollama
```

To start only vLLM:

```
Bash
docker compose up -d vllm
```

6. Verify and Test

Check container status:

```
Bash
```

```
docker ps
```

You should see `ollama_server` and `vllm_server` running.

Check logs:

```
Bash
```

```
docker logs ollama_server  
docker logs vllm_server
```

Look for messages indicating successful startup and GPU detection. For vLLM, you should see messages about

```
Initialized model and Tensor parallel size: 2.
```

Test Ollama:

You can interact with Ollama via its CLI from your host or by exec'ing into the container:

```
Bash
```

```
# From host, if you mapped port 11434  
curl http://localhost:11434/api/generate -d '{  
    "model": "llama3",  
    "prompt": "Why is the sky blue?",  
    "stream": false  
}'  
  
# Or exec into the container (if you pulled a model pre  
docker exec -it ollama_server ollama run llama3 "Explai
```

Test vLLM:

vLLM provides an OpenAI-compatible API.

```
Bash
```

```
curl http://localhost:8000/v1/chat/completions \  
-H "Content-Type: application/json" \  
-d '{  
    "model": "mistralai/Mistral-7B-Instruct-v0.2", # Or  
    "messages": [  
        {"role": "user", "content": "Hello, how are you?"}  
    ],  
    "max_tokens": 100,  
    "temperature": 0.7  
}'
```

7. Stop Services

To stop all services defined in `docker-compose.yaml`:

```
Bash
```

```
docker compose down
```

To stop a specific service:

```
Bash
```

```
docker compose stop ollama  
docker compose stop vllm
```

Advanced Considerations:

- **Model Storage:** The current setup uses Docker volumes for Ollama and a host bind mount for vLLM. For larger models, ensure you have sufficient disk space. Consider a shared network drive if you have a NAS.
- **Quantization:** For very large models (e.g., Llama 70B), you will likely need to use quantized versions (e.g., Q4, Q8) to fit them into your 48GB VRAM. vLLM supports various quantization methods (AWQ, GPTQ, SqueezeLLM, FP8). You would typically specify the quantized model in the `--model` argument or provide the path to its weights.
- **Environment Variables:** For specific models or performance tuning, you might need to add more environment variables or command-line arguments to the `vllm` service in your

`docker-compose.yaml` (e.g., `CUDA_VISIBLE_DEVICES` to
more explicit GPU selection, though `count: all` is usually
sufficient for vLLM's tensor parallelism).

- **Open WebUI/Other Frontends:** You can easily integrate tools like Open WebUI (which runs as another Docker container) by pointing it to your Ollama or vLLM API endpoints. Just add another service to your `docker-compose.yaml` for Open WebUI and configure its `OLLAMA_BASE_URL` or `VLLM_BASE_URL` (or similar for other UIs).
- **Troubleshooting:** If a container fails to start, check `docker logs <container_name>`. Common issues include insufficient GPU memory, incorrect model paths, or driver/toolkit issues.

This setup provides a robust and flexible environment for leveraging the power of your dual RTX 3090s with both Ollama and vLLM.

 Sources

↶ ↷ C ⏪ ⏪ :

Ask Gemini

+  Video  Deep Research  Canvas



Gemini can make mistakes, so double-check it

