Ragflow

Release 1

Jansen Jan

Nov 18, 2025

CONTENTS:

1 About RAGFlow: Named Among GitHub's Fastest-Growing Open Source Projects **1**
- 1.1 The Rise of Retrieval-Augmented Generation in Production . . . 1
- 1.2 Why RAGFlow Resonates in the AI Era . . . 1
- 1.3 A Project in Active Development . . . 2
- 1.4 Conclusion . . . 2

2 RAGFlow System Architecture **3**
- 2.1 Overview . . . 3
- 2.2 Core Infrastructure Containers . . . 3
- 2.3 RAGFlow Application Containers . . . 3
- 2.4 High-Level Architecture Diagram . . . 4
- 2.5 Data & Execution Flow . . . 4

3 Why Infiniflow RAGFlow Uses a Reranker, an Embedding Model, and a Chat Model **5**
- 3.1 Synergy of the Three Models . . . 6

4 Why vLLM is Used to Serve the Reranker Model **7**
- 4.1 Key Reasons for Using vLLM to Serve the Reranker . . . 7
- 4.2 Serving the Reranker Locally with vLLM . . . 8
- 4.3 RAGFlow Integration . . . 9

5 Serving vLLM Reranker Using Docker (CPU-Only) **11**
- 5.1 Docker Compose Configuration (CPU Mode) . . . 11
- 5.2 Key Components Explained . . . 12
- 5.3 Why the Model Must Be Pre-Downloaded Locally . . . 12
- 5.4 Why CPU-Only (No GPU)? . . . 13
- 5.5 Start the Service . . . 13
- 5.6 Verify Availability . . . 13
- 5.7 Integration with RAGFlow . . . 13
- 5.8 Benefits of This CPU + Docker Setup . . . 13

6 Integrating vLLM with RAGFlow via Docker Network **15**
- 6.1 Why Network Configuration is Required . . . 15
- 6.2 Step-by-Step: Configure Docker Network . . . 15
- 6.3 Architecture Diagram . . . 17
- 6.4 Verification . . . 17
- 6.5 Benefits of This Setup . . . 17
- 6.6 Troubleshooting Tips . . . 17
- 6.7 Summary . . . 19

7 Batch Processing and Metadata Management in Infiniflow RAGFlow **21**

7.1 API Base URL . . . 21
7.2 Authentication . . . 21
7.3 Step 1: Retrieve Dataset and Document IDs . . . 22
7.4 Step 2: Add Metadata to a Document (via PUT) . . . 22
7.5 Use Case: Batch Metadata Enrichment . . . 23
7.6 Other Batch-Capable Endpoints . . . 24
7.7 Best Practices . . . 24
7.8 See also: . . . 24
7.9 Summary . . . 24

8 How the Knowledge Graph in Infiniflow/RAGFlow Works 25
8.1 Overview . . . 25
8.2 Construction Process . . . 25
8.3 Query and Retrieval Process . . . 26
8.4 Key Features and Limitations . . . 26

9 Running Llama 3.1 with llama.cpp 27
9.1 1. Model Format: GGUF . . . 27
9.2 2. Compile llama.cpp for Intel i7 (CPU-only) . . . 28
9.3 3. Run the Model with Web Interface . . . 28
9.4 4. Compile llama.cpp with NVIDIA GPU Support (CUDA) . . . 29
9.5 Summary . . . 30

10 Running Multiple Models on llama.cpp Using Docker 31
10.1 Example `docker-compose.yml` . . . 31
10.2 Key Configuration Notes . . . 32
10.3 Usage . . . 32

11 Deploying LLMs in Hybrid Cloud: Why llama.cpp Wins for Us 33
11.1 1. Current Setup: Ollama in Testing . . . 33
11.2 2. Production Requirements: Hybrid Cloud & Multi-User Access . . . 34
11.3 3. Evaluation: vLLM vs llama.cpp . . . 34
11.4 4. Why llama.cpp Is Our Production Choice . . . 34
11.5 5. Migration Path: From Ollama → llama.cpp . . . 35
11.6 Summary . . . 35

12 How InfiniFlow RAGFlow Uses gVisor 37
12.1 gVisor Integration . . . 37
12.2 How RAGFlow Uses gVisor Sandboxes . . . 37
12.3 Security Advantages in Practice . . . 38
12.4 Enabling/Disabling the Sandbox . . . 38
12.5 Summary . . . 38

13 RAGFlow GPU vs CPU: Full Explanation (2025 Edition) 39
13.1 Why Does RAGFlow Still Need a GPU Even When Using Ollama? . . . 39
13.2 Complete RAGFlow Pipeline (with GPU usage marked) . . . 39
13.3 What DeepDoc Actually Does (and Why GPU Makes It 5–20× Faster) . . . 40
13.4 Real-World Performance Numbers . . . 40
13.5 When Do You Actually Need ragflow-gpu? . . . 40
13.6 Recommended Setup in 2025 (Best of Both Worlds) . . . 40
13.7 Monitoring & Verification . . . 41
13.8 Conclusion . . . 41

14 Indices and tables 43

CHAPTER

ONE

ABOUT RAGFLOW: NAMED AMONG GITHUB'S FASTEST-GROWING OPEN SOURCE PROJECTS

October 28, 2025 · 3 min read

The release of **GitHub's 2025 Octoverse report** marks a pivotal moment for the open source ecosystem—and for projects like **RAGFlow**, which has emerged as **one of the fastest-growing open source projects by contributors this year**.

With a **remarkable 2,596% year-over-year growth in contributor engagement**, RAGFlow isn't just gaining traction—**it's defining the next wave of AI-powered development**.

—

1.1 The Rise of Retrieval-Augmented Generation in Production

As the Octoverse report highlights, **AI is no longer experimental—it's foundational**.

- **4.3 million+ AI-related repositories** on GitHub
- **1.1 million+ public repos** import LLM SDKs — a **178% YoY increase**

In this context, **RAGFlow's rapid adoption signals a clear shift**: developers are moving **beyond prototyping** and into **production-grade AI workflows**.

RAGFlow—an **end-to-end retrieval-augmented generation engine with built-in agent capabilities**—is perfectly positioned to meet this demand. It enables developers to build **scalable, context-aware AI applications** that are both **powerful and practical**.

> As the report notes: > *"AI infrastructure is emerging as a major magnet" for open source contributions.* > — **RAGFlow sits squarely at the intersection of AI infrastructure and real-world usability.**

—

1.2 Why RAGFlow Resonates in the AI Era

Several trends highlighted in the Octoverse report **align closely** with RAGFlow's design and mission:

1. **From Notebooks to Production** - Jupyter Notebooks: **+75% YoY** - Python codebases: **surging** - **RAGFlow supports this transition** with a **structured, reproducible framework** for deploying RAG systems in production.
2. **Agentic Workflows Are Going Mainstream** - GitHub Copilot coding agent launch - Rise of AI-assisted development - **RAGFlow's built-in agent capabilities** automate **retrieval, reasoning, and response generation**—key components of modern AI apps.

3. **Security and Scalability Are Top of Mind - 172% YoY increase** in Broken Access Control vulnerabilities - **RAGFlow's enterprise-ready deployment** helps teams address these challenges **secure-by-design**

—

1.3 A Project in Active Development

RAGFlow's evolution mirrors a **deliberate journey**—from solving foundational RAG challenges to **shaping the next generation of enterprise AI infrastructure**.

Phase 1: Solving Core RAG Limitations RAGFlow first made its mark by **systematically addressing core RAG limitations** through integrated technological innovation:

- **Deep document understanding** for parsing complex formats (PDFs, tables, forms)
- **Hybrid retrieval** blending multiple search strategies (vector, keyword, graph)
- **Built-in advanced tools**: **GraphRAG**, **RAPTOR**, and more
- Result: **dramatically enhanced retrieval accuracy and reasoning performance**

Phase 2: The Superior Context Engine for Enterprise Agents Now, building on this robust technical foundation, **RAGFlow is steering toward a bolder vision**:

> **To become the superior context engine for enterprise-grade Agents.**

- Evolving from a **specialized RAG engine** into a **unified, resilient context layer**
- Positioning itself as the **essential data foundation for LLMs in the enterprise**
- Enabling **Agents of any kind** to access **rich, precise, and secure context**
- Ensuring **reliable and effective operation across all tasks**

—

1.4 Conclusion

RAGFlow's **explosive growth** in the 2025 Octoverse is not a coincidence.

It reflects a **global developer movement** toward **production-ready, agentic, secure AI systems**—and RAGFlow is **leading the charge**.

From **deep document parsing** to **scalable agent workflows**, RAGFlow delivers the **infrastructure** and **usability** that modern AI demands.

The future of enterprise AI is context-aware, agent-driven, and open source—and RAGFlow is building it.

CHAPTER

TWO

RAGFLOW SYSTEM ARCHITECTURE

2.1 Overview

RAGFlow is a **fully containerized**, micro-service-based RAG (Retrieval-Augmented Generation) engine. When you run the official `docker-compose.yml` (or `docker-compose-gpu.yml`), the following Docker containers are launched automatically:

2.2 Core Infrastructure Containers

Container	Responsibility
mysql	Persistent storage for users, knowledge bases, datasets, conversation history, API keys, etc.
redis	Cache layer + task queue (Celery) for asynchronous jobs (document parsing, chunking, embedding)
elasticsearch	Hybrid search engine: stores text chunks, keyword indices, and (optionally) dense vectors
minio	S3-compatible object storage for original uploaded files (PDFs, Word, images, etc.)

2.3 RAGFlow Application Containers

Container	Responsibility
ragflow-server (a.k.a. ragflow-api)	Main FastAPI backend. Exposes all REST endpoints (/api/v1/...)
ragflow-web	Frontend (React + Ant Design) – the chat & knowledge-base management UI
celery-worker (often named ragflow-worker)	Background workers that execute long-running tasks (DeepDoc parsing, embedding generation, indexing)
nginx (optional in some setups)	Reverse proxy / static file server (used in production deployments)

2.4 High-Level Architecture Diagram

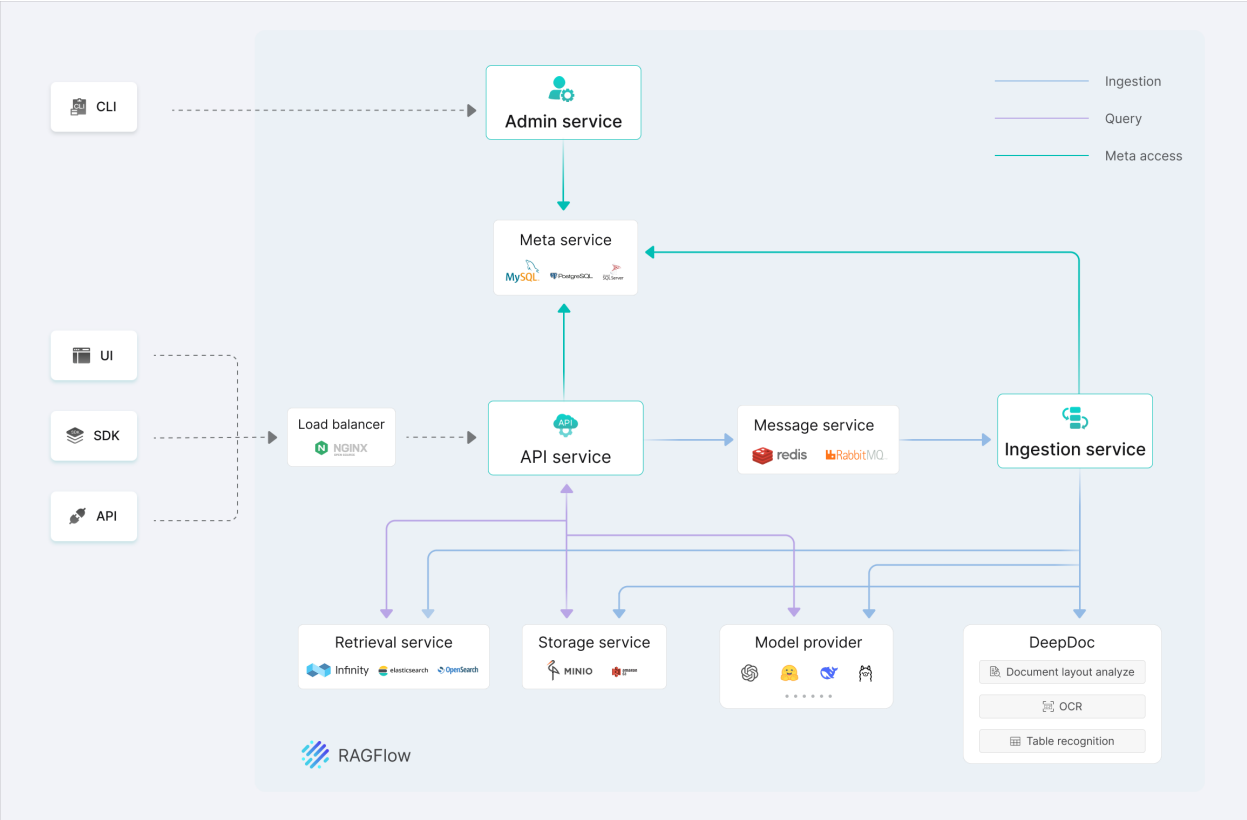

Fig. 1: RAGFlow complete system architecture (as of v0.19+, 2025)

2.5 Data & Execution Flow

1. **User uploads a document** → stored in MinIO → metadata saved in MySQL → parsing task queued in Redis.
2. **Celery worker** picks the task → runs DeepDoc (layout detection, OCR, table extraction) → splits into chunks.
3. **Chunks** are sent to the configured embedding model (Ollama, vLLM, OpenAI, etc.) → vectors returned.

CHAPTER

THREE

WHY INFINIFLOW RAGFLOW USES A RERANKER, AN EMBEDDING MODEL, AND A CHAT MODEL

Infiniflow RAGFlow is a Retrieval-Augmented Generation (RAG) framework designed to build high-quality, traceable question-answering systems over complex data sources. To achieve accurate and contextually relevant responses, RAGFlow employs three distinct models that work in concert:

1. **Embedding Model** - **Purpose**: Converts both the user query and the chunks of retrieved documents into dense vector representations in the same semantic space. - **Role in Pipeline**: Enables semantic similarity search during the retrieval phase. By computing cosine similarity (or other distance metrics) between the query embedding and document chunk embeddings, RAGFlow retrieves the most semantically relevant passages from a large corpus—far beyond keyword matching.

2. **Reranker** - **Purpose**: Refines the initial retrieval results by re-scoring the top-k candidate chunks using a cross-encoder architecture. - **Role in Pipeline**: While the embedding model provides efficient approximate retrieval, the reranker applies a more computationally intensive but accurate relevance scoring. This step significantly improves precision by pushing the most contextually appropriate chunks to the top, reducing noise before generation.

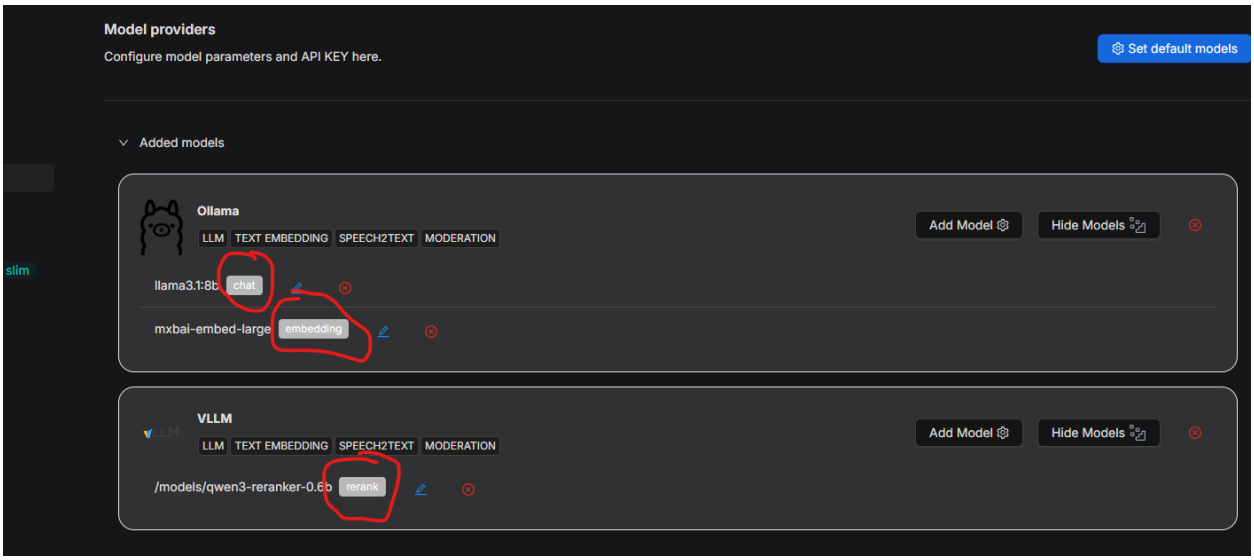

Fig. 1: **Figure 1**: The reranker evaluates query-chunk pairs to produce fine-grained relevance scores.

3. **Chat Model (LLM)** - **Purpose**: Generates the final natural language response grounded in the refined retrieved context. - **Role in Pipeline**: Takes the top reranked chunks as context and synthesizes a coherent, accurate, and fluent answer. The chat model (typically a large language model fine-tuned for instruction following) ensures the output is not only factually aligned with the source material but also conversational and user-friendly.

3.1 Synergy of the Three Models

- **Embedding Model** → Broad, fast, semantic retrieval
- **Reranker** → Precise, fine-grained reordering
- **Chat Model** → Coherent, grounded generation

This modular design allows RAGFlow to balance **speed**, **accuracy**, and **interpretability**, making it suitable for enterprise-grade RAG applications where both performance and trustworthiness are critical.

CHAPTER

FOUR

WHY VLLM IS USED TO SERVE THE RERANKER MODEL

vLLM is a high-throughput, memory-efficient inference engine specifically designed for serving large language models (LLMs). In **Infiniflow RAGFlow**, the **reranker model**—responsible for fine-grained relevance scoring of retrieved document chunks—is served using **vLLM** to ensure low-latency, scalable, and production-ready performance.

4.1 Key Reasons for Using vLLM to Serve the Reranker

1. **PagedAttention for Memory Efficiency** - vLLM uses **PagedAttention**, a novel attention mechanism that manages KV cache in non-contiguous memory pages. - This dramatically reduces memory fragmentation and enables **higher batch sizes** and **longer sequence lengths** (up to 8192 tokens in this case), critical for processing query-chunk pairs during reranking.
2. **High Throughput & Low Latency** - Supports **continuous batching**, allowing dynamic batch formation as requests arrive. - Eliminates head-of-line blocking and maximizes GPU utilization—ideal for real-time reranking in interactive RAG pipelines.
3. **OpenAI-Compatible API** - Exposes a clean, standardized REST API compatible with OpenAI's format. - Enables seamless integration with RAGFlow's orchestration layer without custom inference code.
4. **Support for Cross-Encoder Rerankers** - Models like **Qwen3-Reranker-0.6B** are cross-encoders that take `[query, passage]` pairs as input. - vLLM efficiently handles the bidirectional attention required, delivering relevance scores via `logits[0]` (typically for binary classification: relevant/irrelevant).
5. **Ollama Does Not Support Reranker Models (Yet)** - **Ollama** is excellent for local LLM inference and chat models, but **currently lacks native support for reranker (cross-encoder) models**. - Rerankers require structured input formatting and logit extraction that Ollama's current API and model loading system do not accommodate. - vLLM, in contrast, supports any Hugging Face transformer model—including rerankers—with full access to outputs and fine-grained control.
6. **Scalability Advantage Over Ollama** - When scaling to **multiple concurrent users** or **high-throughput workloads**, vLLM is significantly more robust than Ollama. - vLLM supports **distributed serving**, **tensor parallelism**, **GPU clustering**, and **dynamic batching at scale**. - Ollama is primarily designed for **single-user, local development**, and does not scale efficiently in production environments.

4.2 Serving the Reranker Locally with vLLM

You can run the reranker model locally using vLLM with the following command:

```
vllm serve /models/qwen3-reranker-0.6b \
    --port 8123 \
    --max-model-len 8192 \
    --dtype auto \
    --trust-remote-code
```

Once running, the model is accessible via the OpenAI-compatible endpoint:

GET `http://localhost:8123/v1/models`

Example Response:


```
{
  "object": "list",
  "data": [
    {
      "id": "/models/qwen3-reranker-0.6b",
      "object": "model",
      "created": 1762258164,
      "owned_by": "vllm",
      "root": "/models/qwen3-reranker-0.6b",
      "parent": null,
      "max_model_len": 8192,
      "permission": [
        {
          "id": "modelperm-1a0d5938e30b4eeebb53d9e5c7d9599e",
          "object": "model_permission",
          "created": 1762258164,
          "allow_create_engine": false,
          "allow_sampling": true,
          "allow_logprobs": true,
          "allow_search_indices": false,
          "allow_view": true,
          "allow_fine_tuning": false,
          "organization": "*",
          "group": null,
          "is_blocking": false
        }
      ]
    }
  ]
}
```

4.3 RAGFlow Integration

RAGFlow configures the reranker endpoint in its settings:

```
reranker:
  provider: vllm
  api_base: http://localhost:8123/v1
  model: /models/qwen3-reranker-0.6b
```

During inference, RAGFlow sends batched `[query, passage]` pairs to the vLLM server, receives relevance scores, and reorders chunks before passing them to the chat model.

Result: Fast, accurate, and scalable reranking powered by optimized LLM inference—**where Ollama cannot currently follow, and where vLLM excels in both development and production.**

CHAPTER

FIVE

SERVING VLLM RERANKER USING DOCKER (CPU-ONLY)

To ensure **reproducibility**, **portability**, and **isolation**, **vLLM** can be deployed using **Docker**. This is especially useful in environments with restricted internet access (e.g., corporate networks behind proxies or firewalls), where **Hugging Face Hub** may be blocked or rate-limited.

In this setup, **vLLM runs on CPU only** because:

- **Laptop has no GPU**
- **Home server has an old NVIDIA GPU** (not supported by vLLM's CUDA requirements)

Thus, we use the **official CPU-optimized vLLM image** built from: https://github.com/vllm-project/vllm/blob/main/docker/Dockerfile.cpu

—

5.1 Docker Compose Configuration (CPU Mode)

```
version: '3.8'
services:
  qwen-reranker:
    image: vllm-cpu:latest
    ports: ["8123:8000"]
    volumes:
      - /home/naj/qwen3-reranker-0.6b:/models/qwen3-reranker-0.6b:ro
    environment:
      VLLM_HF_OVERRIDES: |
        {
          "architectures": ["Qwen3ForSequenceClassification"],
          "classifier_from_token": ["no", "yes"],
          "is_original_qwen3_reranker": true
        }
    command: >
      /models/qwen3-reranker-0.6b
      --task score
      --dtype float32
      --port 8000
      --trust-remote-code
      --max-model-len 8192
    deploy:
      resources:
        limits:
```

(continues on next page)

(continued from previous page)

```
        cpus: '10'
        memory: 16G
    shm_size: 4g
    restart: unless-stopped
```

—

5.2 Key Components Explained

- ``**image: vllm-cpu:latest**`` - Official vLLM CPU image (no CUDA dependencies). - Built from: vllm-project/vllm/docker/Dockerfile.cpu - Uses PyTorch CPU backend with optimized inference kernels.
- ``**ports: ["8123:8000"]**`` - Host port **8123** → container port **8000** (vLLM default).
- ``**volumes**`` - Mounts **locally pre-downloaded model** in **read-only** mode.
- ``**VLLM_HF_OVERRIDES**`` - Required for **Qwen3-Reranker** due to custom classification head and token handling.
- ``**command**`` - `--task score`: Enables reranker scoring (outputs relevance logits). - `--dtype float32`: Mandatory on CPU (no half-precision support). - `--max-model-len 8192`: Supports long query+passage pairs.
- **Resource Limits** - `cpus: '10'` and `memory: 16G` prevent system overload. - `shm_size: 4g` ensures sufficient shared memory for batched inference.

—

5.3 Why the Model Must Be Pre-Downloaded Locally

The container **cannot download the model at runtime** due to:

1. **Corporate Proxy / Firewall** - Outbound traffic to `huggingface.co` is blocked or requires authentication.
2. **Hugging Face Hub Blocked** - Git LFS and model downloads fail in restricted networks.
3. **vLLM Auto-Download Fails Offline** - vLLM uses `transformers.AutoModel` → attempts online download if model not found.

Solution: Download via mirror

```
HF_ENDPOINT=https://hf-mirror.com huggingface-cli download Qwen/Qwen3-Reranker-0.6B --
↪local-dir ./qwen3-reranker-0.6b
```

- Remark : for some models you need a token HF_TOKEN=xxxxxxxx (you have to specify the model in the token definition!)
- Remark2 : use "sudo" if non-root!!!
- Uses **accessible mirror** (`hf-mirror.com`).
- Saves model locally for volume mounting.

—

5.4 Why CPU-Only (No GPU)?

- **Laptop**: Integrated graphics only (no discrete GPU).
- **Home Server**: NVIDIA GPU too old (e.g., pre-Ampere) → **not supported** by vLLM's CUDA 11.8+ / FlashAttention requirements.
- **vLLM CPU image** enables full functionality without GPU.

> **Performance Note**: CPU inference is slower (~1–3 sec per batch), but sufficient for **development**, **prototyping**, or **low-throughput** use cases.

—

5.5 Start the Service

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

5.6 Verify Availability

```
curl http://localhost:8123/v1/models
```

Expected output confirms the model is loaded and ready.

—

5.7 Integration with RAGFlow

Update RAGFlow config:

```
reranker:
  provider: vllm
  api_base: http://localhost:8123/v1
  model: /models/qwen3-reranker-0.6b
```

—

5.8 Benefits of This CPU + Docker Setup

- **Works on any machine** (laptop, old server, air-gapped systems)
- **No GPU required**
- **Offline-first** with pre-downloaded model
- **Consistent environment** via Docker
- **Secure**: read-only model, isolated container
- **Scalable later**: switch to GPU image when hardware upgrades

Ideal for local RAGFlow development and constrained production environments.

CHAPTER

SIX

INTEGRATING VLLM WITH RAGFLOW VIA DOCKER NETWORK

To enable **Infiniflow RAGFlow** (running in Docker) to communicate with a **vLLM reranker container**, both services must be on the **same Docker network**. By default, containers are isolated and cannot resolve each other by service name unless explicitly networked.

In this setup, we ensure seamless internal communication between:

- **RAGFlow** (web + backend containers)
- **vLLM reranker** (serving *Qwen3-Reranker-0.6B*)

—

6.1 Why Network Configuration is Required

- RAGFlow runs inside Docker (typically via *docker-compose*).
- vLLM reranker runs in a **separate container** (e.g., CPU-only).
- RAGFlow needs to call: *http://<vllm-service-name>:8000/v1* internally.
- Without shared network → *Connection refused* or DNS lookup failure.

Solution: Attach both services to a **custom Docker bridge network** (e.g., *docker-ragflow*).

—

6.2 Step-by-Step: Configure Docker Network

1. Create a Custom Network

```
docker network create docker-ragflow
```

—

2. Update *docker-compose.yaml* for vLLM Reranker

Ensure the vLLM service uses the network:

Listing 1: docker-compose.yaml (vLLM)

```
version: '3.8'
services:
  qwen-reranker:
```

(continues on next page)

(continued from previous page)

```
    image: vllm-cpu:latest
    container_name: ragflow-vllm-reranker
    ports: ["8123:8000"]
    volumes:
      - /home/naj/qwen3-reranker-0.6b:/models/qwen3-reranker-0.6b:ro
    environment:
      VLLM_HF_OVERRIDES: |
        {
          "architectures": ["Qwen3ForSequenceClassification"],
          "classifier_from_token": ["no", "yes"],
          "is_original_qwen3_reranker": true
        }
    command: >
      /models/qwen3-reranker-0.6b
      --task score
      --dtype float32
      --port 8000
      --trust-remote-code
      --max-model-len 8192
    deploy:
      resources:
        limits:
          cpus: '10'
          memory: 16G
    shm_size: 4g
    restart: unless-stopped
    networks:
      - docker-ragflow

networks:
  docker-ragflow:
    external: true
```

—

3. Connect RAGFlow Containers to the Same Network

If RAGFlow is already running via its own *docker-compose*, **attach** it:

```
docker network connect docker-ragflow ragflow-web
docker network connect docker-ragflow ragflow-server
```

> Replace *ragflow-web*, *ragflow-server* with actual container names (check with *docker ps*).

—

4. Configure RAGFlow to Use Internal vLLM Endpoint

In RAGFlow settings (UI or config file), set:

```
reranker:
  provider: vllm
  api_base: http://ragflow-vllm-reranker:8000/v1
  model: /models/qwen3-reranker-0.6b
```

Key: Use **container name** (*ragflow-vllm-reranker*) — Docker DNS resolves it automatically within the network.

—

6.3 Architecture Diagram

—

6.4 Verification

1. **From RAGFlow container**, test connectivity:

```
docker exec -it ragflow-server curl http://ragflow-vllm-reranker:8000/v1/models
```

2. **Expected output**:

```
{
  "object": "list",
  "data": [
    {
      "id": "/models/qwen3-reranker-0.6b",
      ...
    }
  ]
}
```

—

6.5 Benefits of This Setup

- **Zero external exposure** (optional): vLLM accessible **only** within *docker-ragflow* network.
- **Secure & fast** internal communication.
- **Scalable**: Add more rerankers, LLMs, or vector DBs on same network.
- **Portable**: Works across dev, staging, production with same config.

—

6.6 Troubleshooting Tips

Issue | Solution |

|------|————-| | *Connection refused* | Check network: *docker network inspect docker-ragflow* | | *Unknown host* | Use **container name**, not *localhost* | | Port conflict | Ensure no other service uses *8000* inside network | | Model not loading | Verify volume mount and *trust-remote-code* |

—

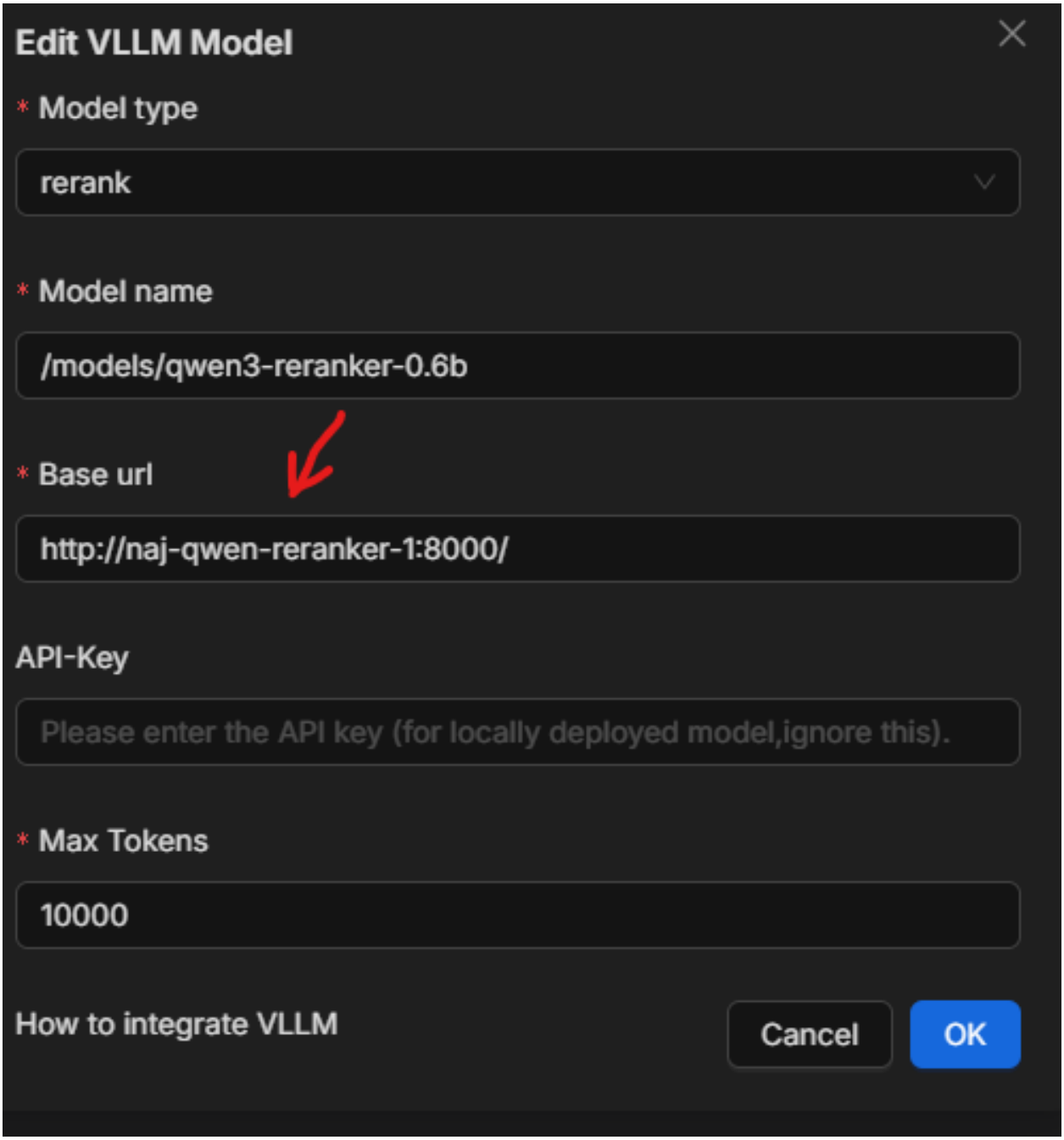

Fig. 1: **Figure 1**: RAGFlow containers communicate with vLLM reranker via internal Docker network *docker-ragflow*. External access (optional) via port *8123*.

6.7 Summary

To use **vLLM inside RAGFlow Docker environment**:

1. Create network: *docker network create docker-ragflow*
2. Connect both RAGFlow and vLLM containers
3. Use **container name** in *api_base*
4. Enjoy **fast, secure, internal reranking**

> **No need for public IPs, reverse proxies, or complex routing** — Docker handles it all.

CHAPTER

SEVEN

BATCH PROCESSING AND METADATA MANAGEMENT IN INFINIFLOW RAGFLOW

Infiniflow RAGFlow provides a **RESTful API** (*/api/v1*) that enables **programmatic control** over datasets and documents, making it ideal for **batch processing large volumes of documents**, **automated ingestion pipelines**, and **metadata enrichment**.

This is essential in enterprise settings where thousands of PDFs, reports, or web pages need to be:

- Ingested in bulk
- Tagged with structured metadata (author, date, source, category, etc.)
- Updated post-ingestion
- Queried or filtered later via the RAG system

—

7.1 API Base URL

```
http://<RAGFLOW_HOST>/api/v1
```

7.2 Authentication

All requests require a **Bearer token**:

```
Authorization: Bearer ragflow-<your-token>
```

> **Tip**: Obtain token via login or API key management in the RAGFlow UI.

—

7.3 Step 1: Retrieve Dataset and Document IDs

Before updating, you **must know** the target:

- **Dataset ID** (e.g., *f388c05e9df711f0a0fe0242ac170003*)
- **Document ID** (e.g., *4920227c9eb711f0bff40242ac170003*)

List all datasets:

```
curl -H "Authorization: Bearer ragflow-..." \
     http://192.168.0.213/api/v1/datasets
```

List documents in a dataset:

```
curl -H "Authorization: Bearer ragflow-..." \
     http://192.168.0.213/api/v1/datasets/<dataset_id>/documents
```

—

7.4 Step 2: Add Metadata to a Document (via PUT)

Use the **PUT** endpoint to **update metadata** of an existing document:

```
curl --request PUT \
     --url http://192.168.0.213/api/v1/datasets/f388c05e9df711f0a0fe0242ac170003/
↪documents/4920227c9eb711f0bff40242ac170003 \
     --header 'Content-Type: multipart/form-data' \
     --header 'Authorization: Bearer ragflow-QxNWIzMGNlOWRmMzExZjBhZjljMDI0Mm' \
     --data '{
       "meta_fields": {
         "author": "Example Author",
         "publish_date": "2025-01-01",
         "category": "AI Business Report",
         "url": "https://example.com/report.pdf"
       }
     }'
```

Request Breakdown:

- **Method**: *PUT*
- **Path**: */api/v1/datasets/<dataset_id>/documents/<document_id>*
- **Content-Type**: *multipart/form-data* (required even for JSON payload)
- **Body**: JSON string with *"meta_fields"* object

Response (on success):

```
{
  "code": 0,
  "message": "Success",
  "data": { "document_id": "4920227c9eb711f0bff40242ac170003" }
}
```

—

7.5 Use Case: Batch Metadata Enrichment

You can **automate metadata tagging** for **1000s of documents** using a script:

```
import requests
import json

BASE_URL = "http://192.168.0.213/api/v1"
TOKEN = "ragflow-QxNWIzMGNlOWRmMzExZjBhZjljMDI0Mm"
HEADERS = {
    "Authorization": f"Bearer {TOKEN}",
    "Content-Type": "multipart/form-data"
}

# Example: Load CSV with doc_id, author, date, url...
import pandas as pd
df = pd.read_csv("documents_metadata.csv")

for _, row in df.iterrows():
    dataset_id = row['dataset_id']
    doc_id = row['document_id']
    payload = {
        "meta_fields": {
            "author": row['author'],
            "publish_date": row['publish_date'],
            "category": row['category'],
            "url": row['source_url']
        }
    }

    files = {'': ('', json.dumps(payload), 'application/json')}
    resp = requests.put(
        f"{BASE_URL}/datasets/{dataset_id}/documents/{doc_id}",
        headers=HEADERS,
        files=files
    )
    print(doc_id, resp.json().get("message"))
```

Benefits:

- Enrich RAG context with **structured, queryable metadata**
- Enable **filtering** in UI or API (e.g., "Show reports from 2025 by Author X")
- Improve **traceability** and **auditability**

—

7.6 Other Batch-Capable Endpoints

Endpoint | Purpose |

|--------|———| | *POST /api/v1/datasets* | Create new dataset | | *POST /api/v1/datasets/{id}/documents* | Upload new documents (with metadata) | | *DELETE /api/v1/datasets/{id}/documents/{doc_id}* | Remove document | | *GET /api/v1/datasets/{id}/documents* | List + filter by metadata |

—

7.7 Best Practices

1. **Always use IDs** — never rely on filenames
2. **Batch in chunks** (e.g., 100 docs/sec) to avoid rate limits
3. **Validate metadata schema** in RAGFlow settings first
4. **Log responses** for retry logic
5. **Use dataset-level permissions** for access control

—

7.8 See also:

https://github.com/infiniflow/ragflow/blob/main/example/http/dataset_example.sh

7.9 Summary

RAGFlow's **API-first design** enables:

- **Scalable batch ingestion**
- **Rich metadata attachment**
- **Full automation** of document lifecycle

> **Perfect for ETL pipelines, CMS integration, or enterprise knowledge base automation.**

With this API, you can manage **tens of thousands of documents** with full metadata — all programmatically.

CHAPTER

EIGHT

HOW THE KNOWLEDGE GRAPH IN INFINIFLOW/RAGFLOW WORKS

RAGFlow is an open-source Retrieval-Augmented Generation (RAG) engine developed by Infiniflow, designed to enhance LLM-based question-answering by integrating deep document understanding with structured data processing. The knowledge graph (KG) component plays a pivotal role in handling complex queries, particularly multi-hop question-answering, by extracting and organizing entities and relationships from documents into a graph structure. This enables more accurate and interconnected retrieval beyond simple vector-based searches.

8.1 Overview

The KG is constructed as an intermediate step in RAGFlow's data pipeline, bridging raw document extraction and final indexing. It transforms unstructured text into a relational graph, allowing for entity-based reasoning and traversal during retrieval. Key benefits include:

- **Multi-Hop Query Support**: Facilitates queries requiring inference across multiple documents or concepts (e.g., "What caused the event that affected company X?").
- **Dynamic Updates**: From version 0.16.0 onward, the KG is built across an entire knowledge base (multiple files) and automatically updates when new documents are uploaded and parsed.
- **Integration with RAG 2.0**: Part of preprocessing stages like document clustering and domain-specific embedding, ensuring retrieval results are contextually rich and grounded.

The KG is stored as chunks in RAGFlow's document engine (Elasticsearch by default or Infinity for advanced vector/graph capabilities), making it queryable alongside embeddings.

8.2 Construction Process

The KG construction occurs after initial document parsing but before indexing. Here's the step-by-step workflow:

1. **Document Ingestion and Extraction**: - Users upload files (e.g., PDF, Word, Excel, TXT) to a knowledge base. - RAGFlow's Deep Document Understanding (DDU) module parses the content, extracting structured elements like text blocks, tables, and layouts using OCR and layout models.
2. **Entity and Relation Extraction**: - Using NLP models (integrated via configurable LLMs or embedding services), RAGFlow identifies entities (e.g., people, organizations, events) and relations (e.g., "causes", "affiliated with") from extracted chunks. - This is model-driven: Preprocessing applies entity recognition and relation extraction to raw text, often leveraging domain-specific prompts for accuracy.
3. **Graph Building**: - Entities become nodes, and relations form directed/undirected edges. - The graph is unified across the entire dataset (not per-file since v0.16.0), enabling cross-document connections. - Acceleration features (introduced in later releases) optimize extraction speed, such as batch processing or efficient model inference.

4. **Storage**: - Graph chunks (nodes, edges, metadata) are serialized and stored in the document engine. - No separate graph database is required; it's embedded within the vector/full-text index for hybrid queries.

Note: Construction can be toggled per knowledge base and is optional, but recommended for complex domains like finance or healthcare.

8.3 Query and Retrieval Process

During inference, the KG enhances retrieval in the following manner:

1. **Query Parsing**: - Incoming user queries are analyzed to detect multi-hop intent (e.g., via LLM routing).
2. **Hybrid Retrieval**: - Combine vector similarity search (for semantic relevance) with graph traversal:
    - Start from query entities as seed nodes.
    - Traverse edges to fetch connected nodes (e.g., 1-2 hops).
    - Rank results by relevance scores, incorporating graph proximity.
    - Infinity engine (optional) supports efficient graph-range filtering alongside vectors.
3. **Augmentation and Generation**: - Retrieved graph-derived contexts (e.g., subgraphs or paths) are fused with text chunks. - Fed to the LLM for grounded generation, with citations traceable to source documents.

This process addresses limitations of pure vector RAG, such as hallucination in interconnected scenarios, by providing explicit relational paths.

8.4 Key Features and Limitations

- **Features**: - **Scalability**: Handles enterprise-scale knowledge bases with dynamic rebuilding. - **Customizability**: Configurable extraction models and hop limits. - **Agent Integration**: Supports agentic workflows for iterative graph exploration. - **Performance**: Accelerated extraction in v0.21+ releases.
- **Limitations**: - Relies on quality of upstream extraction; noisy documents may yield incomplete graphs. - Graph depth is configurable but can increase latency for deep traversals. - Arm64 Linux support is limited when using Infinity.

For implementation details, refer to the official guide at https://github.com/infiniflow/ragflow/blob/main/docs/guides/dataset/construct_knowledge_graph.md. To experiment, deploy RAGFlow via Docker and enable KG in your knowledge base settings.

CHAPTER

NINE

RUNNING LLAMA 3.1 WITH LLAMA.CPP

Table of Contents

- *1. Model Format: GGUF*
- *2. Compile llama.cpp for Intel i7 (CPU-only)*
- *3. Run the Model with Web Interface*
  - *Features*
- *4. Compile llama.cpp with NVIDIA GPU Support (CUDA)*
  - *Prerequisites*
  - *Build with CUDA*
  - *Run with GPU offloading*
- *Summary*

9.1 1. Model Format: GGUF

`llama.cpp` uses the **GGUF** (GPT-Generated Unified Format) model format.

You can download a pre-quantized **Llama 3.1 8B** model in GGUF format directly from Hugging Face:

```
https://huggingface.co/QuantFactory/Meta-Llama-3-8B-GGUF
```

Example file: `Meta-Llama-3-8B.Q4_K_S.gguf` (~4.7 GB, 4-bit quantization, excellent quality/size tradeoff).

Note: The `Q4_K_S` variant uses ~4.7 GB RAM and runs efficiently on Intel i7 CPUs.

9.2 2. Compile llama.cpp for Intel i7 (CPU-only)

```
# Step 1: Clone the repository
git clone https://github.com/ggerganov/llama.cpp
cd llama.cpp

# Step 2: Build for CPU (Intel i7, AVX2 enabled by default)
make clean
make -j$(nproc) LLAMA_CPU=1

# Optional: Force AVX2 (most i7 CPUs support it)
make clean
make -j$(nproc) LLAMA_CPU=1 LLAMA_AVX2=1
```

The binaries will be in the root directory:

- `./llama-cli` → interactive CLI
- `./server` → web server (OpenAI-compatible API + full web UI)

Warning: **Do not use vLLM** on older Xeon v2 CPUs — they **lack AVX-512**, which vLLM requires. **llama.cpp is a better choice** — it runs efficiently with just AVX2 or even SSE.

9.3 3. Run the Model with Web Interface

Place the downloaded GGUF file in a `models/` folder:

```
mkdir -p models
# Copy or symlink the model
ln -s /path/to/Meta-Llama-3-8B.Q4_K_S.gguf models/
```

Start the server on port **8087** using **12 threads**:

```
./server \
  --model models/Meta-Llama-3-8B.Q4_K_S.gguf \
  --port 8087 \
  --threads 12 \
  --host 0.0.0.0
```

In a corporate network : avoid contacting the proxy !!

```
curl -X POST http://127.0.0.1:8087/v1/chat/completions \
  --noproxy 127.0.0.1,localhost \
  -H "Content-Type: application/json" \
  -d '{
    "model": "Meta-Llama-3-8B",
    "messages": [
      {"role": "system", "content": "You are a helpful assistant."},
      {"role": "user", "content": "Hello! How are you?"}
```

(continues on next page)

(continued from previous page)

```
    ]
  }'
```

9.3.1 Features

- **Full web UI** at: http://localhost:8087
- **OpenAI-compatible API** at: http://localhost:8087/v1
- List models:

```
curl http://localhost:8087/v1/models
```

Response:


```
{
  "object": "list",
  "data": [
    {
      "id": "models/Meta-Llama-3-8B.Q4_K_S.gguf",
      "object": "model",
      "created": 1762783003,
      "owned_by": "llamacpp",
      "meta": {
        "vocab_type": 2,
        "n_vocab": 128256,
        "n_ctx_train": 8192,
        "n_embd": 4096,
        "n_params": 8030261248,
        "size": 4684832768
      }
    }
  ]
}
```


9.4 4. Compile llama.cpp with NVIDIA GPU Support (CUDA)

If you have an **NVIDIA GPU** (e.g., RTX 3060, 4070, A100, etc.), enable **CUDA acceleration**:

9.4.1 Prerequisites

- NVIDIA driver (\geq{} 525)
- CUDA Toolkit (\geq{} 11.8, preferably 12.x)
- `nvcc` in `$PATH`

9.4.2 Build with CUDA

```
# Clean previous build
make clean

# Build with full CUDA support
make -j$(nproc) \
  LLAMA_CUDA=1 \
  LLAMA_CUDA_DMMV=1 \
  LLAMA_CUDA_F16=1

# Optional: Specify compute capability (e.g., for RTX 40xx)
# make LLAMA_CUDA=1 CUDA_ARCH="-gencode arch=compute_89,code=sm_89"
```

9.4.3 Run with GPU offloading

```
./server \
  --model models/Meta-Llama-3-8B.Q4_K_S.gguf \
  --port 8087 \
  --threads 8 \
  --n-gpu-layers 999 \   # offload ALL layers to GPU
  --host 0.0.0.0
```

Tip: Use `nvidia-smi` to monitor VRAM usage. For 8B Q4 (~4.7 GB), even a **6 GB GPU** can run it fully offloaded.

9.5 Summary

Feature	Command / Note
Model Format	**GGUF**
Download	https://huggingface.co/QuantFactory/...GGUF
CPU Build (i7)	`make LLAMA_CPU=1`
GPU Build (CUDA)	`make LLAMA_CUDA=1`
Run Server	`./server --model ... --port 8087`
Web UI	http://localhost:8087
API	http://localhost:8087/v1

llama.cpp = lightweight, CPU/GPU flexible, no AVX-512 needed → ideal replacement for vLLM on older hardware.

—

CHAPTER

TEN

RUNNING MULTIPLE MODELS ON LLAMA.CPP USING DOCKER

This guide demonstrates how to run multiple language models simultaneously using `llama.cpp` in Docker via `docker-compose`. The example below defines two services: a lightweight reranker model (Qwen3 0.6B) and a general-purpose chat model (Llama 3.1).

10.1 Example `docker-compose.yml`

```yaml
services:
  qwen-reranker:
    image: ghcr.io/ggerganov/llama.cpp:server-cpu
    ports:
      - "8123:8080"
    volumes:
      - /home/naj/qwen3-reranker-0.6b:/models/qwen3-reranker-0.6b:ro
    environment:
      - MODEL=/models/qwen3-reranker-0.6b
    command: >
      --model /models/qwen3-reranker-0.6b/model.gguf
      --port 8080
      --host 0.0.0.0
      --n-gpu-layers 0
      --ctx-size 8192
      --threads 6
      --temp 0.0
      --rpc
    deploy:
      resources:
        limits:
          cpus: '6'
          memory: 10G
    shm_size: 4g
    restart: unless-stopped

  llama3.1-chat:
    image: ghcr.io/ggerganov/llama.cpp:server-cpu
    ports:
      - "8124:8080"
    volumes:
      - /home/naj/llama3.1:/models/llama3.1:ro
```

(continues on next page)

(continued from previous page)

```
    environment:
      - MODEL=/models/llama3.1
    command: >
      --model /models/llama3.1/model.gguf
      --port 8080
      --host 0.0.0.0
      --n-gpu-layers 0
      --ctx-size 8192
      --threads 10
      --temp 0.7
      --rpc
    deploy:
      resources:
        limits:
          cpus: '10'
          memory: 20G
    shm_size: 8g
    restart: unless-stopped
```

10.2 Key Configuration Notes

- **Images**: Both services use the official CPU-optimized `llama.cpp` server image.
- **Ports**: - Reranker exposed on `8123` → internal `8080` - Chat model exposed on `8124` → internal `8080`
- **Volumes**: Model directories are mounted read-only (`:ro`) from the host.
- **Environment**: `MODEL` variable simplifies path references in commands.
- **Command Flags**: - `--n-gpu-layers 0`: Forces CPU-only inference. - `--ctx-size 8192`: Sets context length. - `--temp`: Controls randomness (0.0 for deterministic reranking, 0.7 for chat). - `--rpc`: Enables RPC interface for external control.
- **Resource Limits**: CPU and memory capped via `deploy.resources.limits`.
- **Shared Memory (shm_size)**: Increased to support larger contexts and batching.
- **Restart Policy**: `unless-stopped` ensures containers restart on failure or reboot.

10.3 Usage

Start the services:

```
docker compose up -d
```

Access the models:

- Reranker: `http://localhost:8123`
- Chat: `http://localhost:8124`

Send requests using the OpenAI-compatible API or `llama.cpp` client tools.

```rst .. _deployment-considerations:
```

CHAPTER

ELEVEN

DEPLOYING LLMS IN HYBRID CLOUD: WHY LLAMA.CPP WINS FOR US

Table of Contents

- *1. Current Setup: Ollama in Testing*
- *2. Production Requirements: Hybrid Cloud & Multi-User Access*
- *3. Evaluation: vLLM vs llama.cpp*
- *4. Why llama.cpp Is Our Production Choice*
  - *Example: Production-Ready Server*
- *5. Migration Path: From Ollama → llama.cpp*
- *Summary*

11.1 1. Current Setup: Ollama in Testing

We have been using **Ollama** in a **test environment** with excellent results:

- **Easy to use** — *ollama run llama3.1* just works
- **Docker support** is first-class:

```
FROM ollama/ollama
COPY Modelfile /root/.ollama/
RUN ollama create my-llama3.1 -f Modelfile
```

- Models are pulled, versioned, and cached automatically
- Web UI and OpenAI-compatible API available out of the box

Verdict: Perfect for **prototyping**, **local dev**, and **small-scale testing**.

11.2 2. Production Requirements: Hybrid Cloud & Multi-User Access

When moving to **production in a hybrid cloud**, new constraints emerge:

We need a **lightweight, portable, hardware-agnostic** inference engine.

11.3 3. Evaluation: vLLM vs llama.cpp

Criteria	vLLM	llama.cpp
Hardware Requirements	Requires **AVX-512** (fails on Xeon v2, older i7)	Runs on **SSE2+**, AVX2 optional
GPU Support	Excellent (PagedAttention, high throughput)	CUDA, Metal, Vulkan — full offloading
CPU Performance	Poor without AVX-512	**Best-in-class** quantized inference
Binary Size	~200 MB + Python deps	**< 10 MB** (statically linked)
Deployment	Python server, complex deps	Single binary, *scp* and run
Multi-user / API	Built-in OpenAI API	*server* binary with full OpenAI compat + web UI
Quantization Support	FP16/BF16 only	Q4_K, Q5_K, Q8_0, etc. — **4–8 GB models fit in RAM**

Key Finding:

> **We cannot use vLLM** on our legacy Xeon v2 fleet due to missing **AVX-512**. > **llama.cpp runs efficiently** on the **same hardware** with **Q4_K_M** models.

11.4 4. Why llama.cpp Is Our Production Choice

Decision

llama.cpp is selected for **hybrid cloud LLM deployment** because:

- **Runs everywhere**: Old CPUs, new GPUs, laptops, edge
- **Single static binary**: No Python, no CUDA runtime hell
- **GGUF format**: Share models with Ollama, local files, S3
- **Built-in server**: OpenAI API + full web UI
- **Thread & context control**: *–threads*, *–ctx-size*, *–n-gpu-layers*
- **Kubernetes-ready**: Tiny image, fast startup

11.4.1 Example: Production-Ready Server

```
./llama.cpp/server \
  --model /models/llama3.1-8b-instruct.Q4_K_M.gguf \
  --port 8080 \
  --host 0.0.0.0 \
  --threads 16 \
  --ctx-size 8192 \
  --n-gpu-layers 0    # CPU-only on older nodes
  --log-disable
```

Deploy via Docker:

```
FROM alpine:latest
COPY llama.cpp/server /usr/bin/
COPY models/*.gguf /models/
EXPOSE 8080
CMD ["server", "--model", "/models/llama3.1-8b-instruct.Q4_K_M.gguf", "--port", "8080"]
```

11.5 5. Migration Path: From Ollama → llama.cpp

```
# 1. Reuse Ollama's GGUF
cp ~/.ollama/models/blobs/sha256-* /production/models/

# 2. Deploy llama.cpp server
kubectl apply -f llama-cpp-deployment.yaml

# 3. Point clients to new endpoint
export OPENAI_API_BASE=http://llama-cpp-prod:8080/v1
```

Zero model reconversion. Zero downtime.

11.6 Summary

Use Case	Recommended Tool
Local dev / prototyping	**Ollama**
Hybrid cloud, old hardware, scale	**llama.cpp**
High-throughput GPU cluster	vLLM (if AVX-512 available)

> llama.cpp = the Swiss Army knife of LLM inference.

—

CHAPTER

TWELVE

HOW INFINIFLOW RAGFLOW USES GVISOR

InfiniFlow RAGFlow is an open-source RAG (Retrieval-Augmented Generation) engine that supports document understanding and LLM orchestration. To enhance security when executing untrusted or user-provided code (e.g., Python agents, dynamic data processing scripts, or custom tool functions), RAGFlow runs certain components inside isolated sandboxes.

12.1 gVisor Integration

RAGFlow leverages **gVisor** as its primary sandboxing technology in containerized (Docker/Kubernetes) deployments.

12.1.1 What is gVisor?

gVisor is an open-source project by Google that provides a user-space kernel (the Sentry component) and a lightweight virtual machine (runsc runtime). It intercepts and emulates Linux system calls instead of letting the container directly access the host kernel.

Key benefits of gVisor in a containerized environment:

- **Strong syscall isolation**: Only a limited, explicitly allowed set of syscalls reaches the host kernel. Unknown or dangerous syscalls are blocked or emulated in user space.
- **Reduced kernel attack surface**: Even if a process escapes the container's namespaces/cgroups/seccomp filters, it still operates through gVisor's restricted interface.
- **Compatibility with standard Docker**: gVisor integrates as an OCI-compatible runtime (`runsc`), so no changes to Dockerfile or Kubernetes pod spec syntax are required beyond specifying the runtime.

12.2 How RAGFlow Uses gVisor Sandboxes

When the sandbox feature is enabled (default in recent RAGFlow releases), the following happens:

1. **Agent/Tool Execution Sandbox** - Python-based agents and custom tools are executed inside a dedicated gVisor-powered container. - The sandbox container has extremely limited privileges:
    - No direct access to the host filesystem (only a small tmpfs and necessary code mounts as read-only).
    - Restricted network access (usually completely disabled or limited to internal RAGFlow services).
    - Dropped capabilities and a strict seccomp profile enforced by gVisor.
2. **Runtime Selection** - RAGFlow's Docker Compose and Helm charts include a `ragflow-sandbox` service that uses the `runsc` runtime:

```
runtime: runsc
runtime-config:
  # Optional gVisor flags
  platform: ptrace  # or kvm on supported hardware
```

3. **Communication** - The main RAGFlow application communicates with the sandbox via gRPC or stdin/stdout (depending on version). - Code and data are injected securely at container startup; no runtime file writes from inside the sandbox are allowed.

12.3 Security Advantages in Practice

12.4 Enabling/Disabling the Sandbox

In `docker-compose.yml` or Helm values:

```
services:
  ragflow:
    environment:
      - ENABLE_SANDBOX=true   # default true in v0.10+
```

To run without gVisor (less secure, for trusted environments only):

```
environment:
  - ENABLE_SANDBOX=false
```

12.5 Summary

By running untrusted code execution inside gVisor-powered containers, RAGFlow significantly reduces the risk of a compromised agent or malicious user-uploaded script affecting the host system or other services, while maintaining good performance and seamless integration with standard container orchestration tools.

CHAPTER

THIRTEEN

RAGFLOW GPU VS CPU: FULL EXPLANATION (2025 EDITION)

13.1 Why Does RAGFlow Still Need a GPU Even When Using Ollama?

You are absolutely right to ask this question.

Most people configure RAGFlow to use **external services** (Ollama, Xinference, vLLM, OpenAI, etc.) for:

- Embedding model
- Re-ranking model
- Inference LLM (the actual answer generator)

Ollama runs these models **entirely on your GPU** — RAGFlow only sends HTTP requests. So why does the official documentation and community still strongly recommend the **ragflow-gpu** image (or setting `DEVICE=gpu`)?

The answer is simple: **Deep Document Understanding (DeepDoc)** — the part that happens *before* any embedding or LLM call.

13.2 Complete RAGFlow Pipeline (with GPU usage marked)

Step	Component	Runs inside RAGFlow?	Uses GPU when?
1. Document Upload	DeepDoc parser	Yes (core of RAGFlow)	**YES** — heavily
2. Chunking	Text splitting	Yes	No (pure CPU)
3. Embedding	Sentence-Transformers, BGE…	External (Ollama, etc.)	GPU via Ollama/vLLM
4. Vector storage	Elasticsearch / InfiniFlow DB	Yes	No
5. Retrieval	Vector + keyword search	Yes	No
6. Re-ranking	Cross-encoder (optional)	External or local	GPU via external service
7. Answer generation	LLM (Llama 3, Qwen2, etc.)	External (Ollama, etc.)	GPU via Ollama/vLLM

→ The **only part that RAGFlow itself accelerates with GPU** is Step 1 — but it is by far the most compute-intensive for real-world documents.

13.3 What DeepDoc Actually Does (and Why GPU Makes It 5–20× Faster)

When you upload a PDF, scanned image, or complex report, DeepDoc performs these AI-heavy tasks:

1. **Layout Detection** Detects columns, headers, footers, reading order using CNN-based models (LayoutLM-style).
2. **Table Structure Recognition (TSR)** Identifies table boundaries, row/column spans, merged cells — extremely important for accurate retrieval.
3. **Formula & Math Recognition** Converts LaTeX/math images into readable text.
4. **Enhanced OCR** For scanned PDFs: runs deep-learning OCR models (not just Tesseract CPU).
5. **Visual Language Model Tasks** (charts, diagrams, screenshots) Optionally calls lightweight VLMs (Qwen2-VL, LLaVA, etc.) to describe images inside the document.

All of these run **inside RAGFlow's deepdoc module** using PyTorch + CUDA when `DEVICE=gpu` is enabled.

13.4 Real-World Performance Numbers

13.5 When Do You Actually Need ragflow-gpu?

YES – You need it if your documents contain any of the following: - Scanned pages (images instead of selectable text) - Complex tables or financial reports - Charts, graphs, screenshots - Mixed layouts (multi-column, sidebars, footnotes) - Handwritten notes or formulas

NO – You can stay on ragflow-cpu if: - All documents are clean, born-digital text (Word → PDF, Markdown, etc.) - You only do quick prototypes with a few simple files - You have no NVIDIA GPU available

13.6 Recommended Setup in 2025 (Best of Both Worlds)

```
# .env file
DEVICE=gpu                    # ← Enables DeepDoc GPU acceleration
SVR_HTTP_PORT=80

# Use Ollama (or vLLM) for embeddings + LLM
EMBEDDING_MODEL=ollama/bge-large
RERANK_MODEL=reranker  (this cannnot be served by ollama, vLLM or llama.cpp is needed)
LLM_MODEL=ollama/llama3.1:70b
```

Result: - DeepDoc parsing → blazing fast on your NVIDIA GPU (inside RAGFlow container) - Embeddings + LLM → also blazing fast on the same GPU (inside Ollama container) - No bottlenecks anywhere

13.7 Monitoring & Verification

During document upload:

```
watch -n 1 nvidia-smi
```

You will see: - RAGFlow container using 4–12 GB VRAM during parsing - Ollama container using VRAM only when embedding or generatinging

13.8 Conclusion

- **ragflow-cpu** → fine for clean text only
- **ragflow-gpu** → mandatory for real-world unstructured documents

Even if Ollama handles your LLM and embeddings perfectly, **without ragflow-gpu, the very first step (understanding the document) will be painfully slow or inaccurate**.

Use `DEVICE=gpu` — it's the difference between a toy and a true enterprise-grade RAG system.

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search