# Ragflow

*Release 1*

**Jansen Jan**

**Nov 04, 2025**

# CONTENTS:

CHAPTER

ONE

# 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.

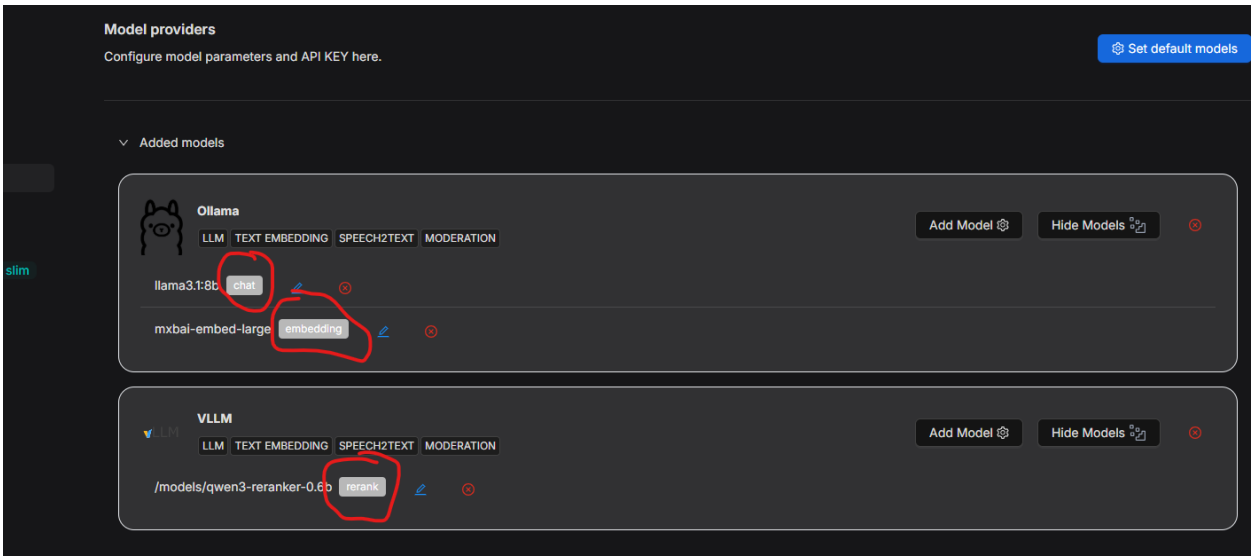


Fig. 1: **Figure 1**: The reranker evaluates query-chunk pairs to produce fine-grained relevance scores.

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.

## 1.1 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

TWO

# 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.

## 2.1 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.

## 2.2 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
        }
      ]
    }
  ]
}
```

## 2.3 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

THREE

# 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

—

## 3.1 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
```

—

## 3.2 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.

—

## 3.3 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
```

- Uses **accessible mirror** (`hf-mirror.com`).
- Saves model locally for volume mounting.

—

## 3.4 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.

—

## 3.5 Start the Service

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

## 3.6 Verify Availability

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

Expected output confirms the model is loaded and ready.

—

## 3.7 Integration with RAGFlow

Update RAGFlow config:

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

—

## 3.8 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

FOUR

# 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*)

—

## 4.1 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*).

—

## 4.2 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.

—

## 4.3 Architecture Diagram

—

## 4.4 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",
      ...
    }
  ]
}
```

—

## 4.5 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.

—

## 4.6 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* |

—

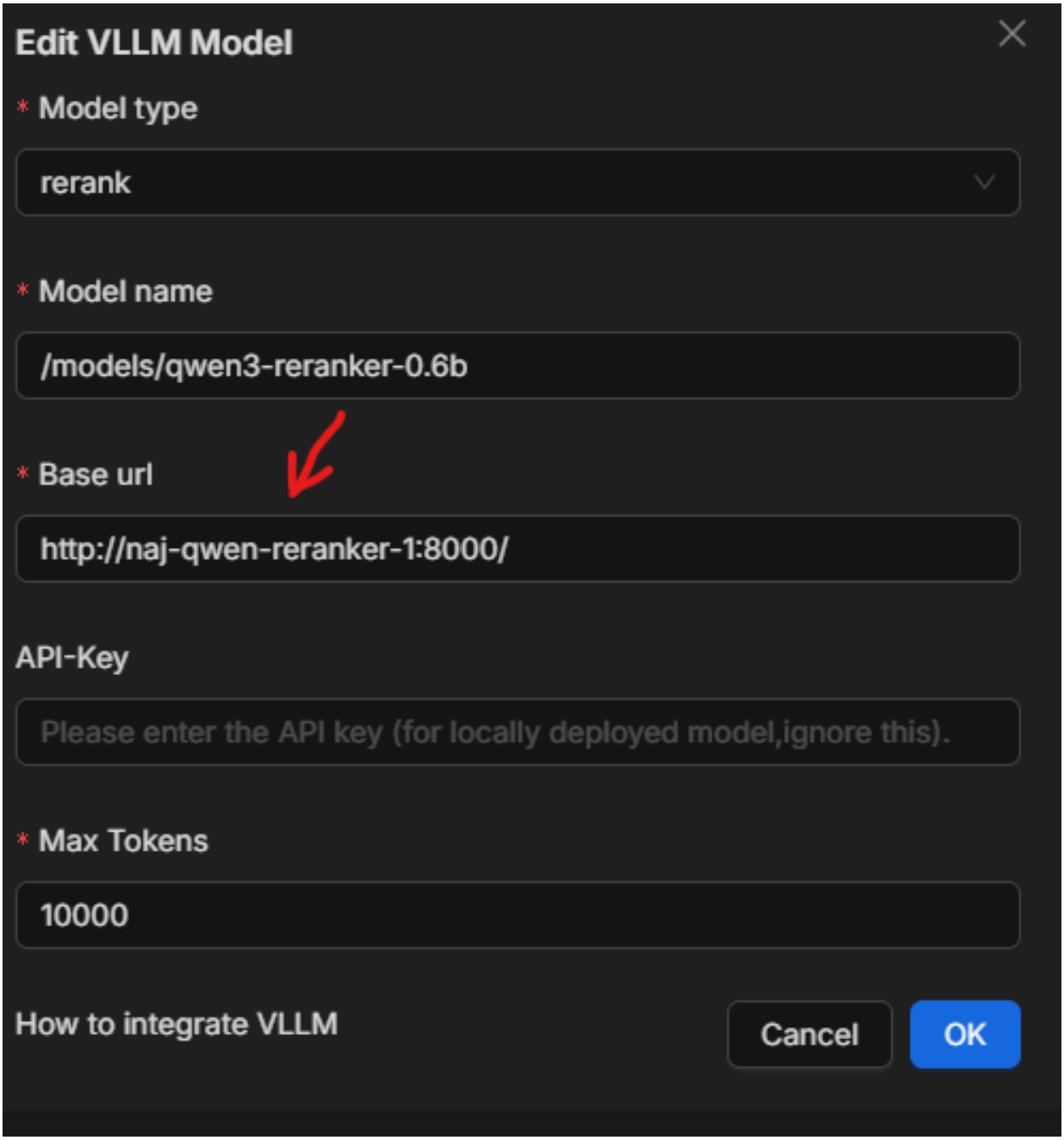


Fig. 1: **Figure 1**: RAGFlow containers communicate with vLLM reranker via internal Docker network *docker-ragflow*. External access (optional) via port *8123*.

## 4.7 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

FIVE

# 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

—

## 5.1 API Base URL

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

## 5.2 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.

—

## 5.3 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
```

—

## 5.4 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" }
}
```

—

## 5.5 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**

—

## 5.6 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 |

—

## 5.7 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

—

## 5.8 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

SIX

# INDICES AND TABLES

- genindex
- modindex
- search