ChatGPT

NVIDIA NeMo Retriever in a RAG Pipeline: Embedding & Reranking Services

Retrieval-Augmented Generation (RAG) is a technique where an AI model (LLM) is augmented with external knowledge from a document database or knowledge base to improve accuracy and relevance [1]. NVIDIA's **NeMo Retriever** provides specialized microservices for **text embedding** and **text reranking** that can be plugged into RAG workflows to boost performance and accuracy [2] [3]. These microservices (called **NVIDIA Inference Microservices** or **NIMs**) are modular components running in Docker containers (with NVIDIA Triton inference server under the hood) and expose easy-to-use APIs (compatible with OpenAI's API schema) [4] [5]. In simple terms:

- **Embedding Service (Text Embedding NIM)** – converts text into high-dimensional numerical vectors ("embeddings") that capture semantic meaning [6]. This service is used to encode documents and queries for semantic search in the RAG pipeline.
- **Reranking Service (Text Reranking NIM)** – takes a query and a set of retrieved passages, and reorders (ranks) those passages based on relevance to the query [7] [8]. It's a fine-tuned model (e.g. based on Mistral or Llama) that evaluates how well each candidate passage answers the question, helping pick the best context for the LLM.

NVIDIA introduced these NeMo Retriever services in 2025 to enable higher-accuracy RAG with enterprise data, boasting improvements like ~50% fewer incorrect answers compared to baseline open-source pipelines [9]. Below, we explain where the embedding and reranker NIMs fit into the RAG workflow, provide a code example integrating them (using LangChain for simplicity), and list the prerequisites (accounts, licenses, etc.) needed to use these services.

How the NeMo Embedding and Reranking NIMs Fit into a RAG Workflow

A RAG pipeline has two core stages: **retrieval** (finding relevant information for a query) and **generation** (producing an answer using that information). NVIDIA's NeMo Retriever services enhance the retrieval stage:

- **Document Ingestion & Indexing**: Before queries are answered, all your enterprise data (documents, webpages, PDFs, etc.) must be ingested and indexed. This involves extracting text from files (for example, using NeMo's multimodal extractors for PDFs/images) and breaking the text into chunks. Each text chunk is then fed to the **Embedding NIM** to produce an embedding vector [6]. These vectors are stored in a **vector database** (e.g. FAISS, RedisVector, etc.), often accelerated by GPU libraries like NVIDIA cuVectorSearch (cuVS) for fast similarity search [6]. The Embedding NIM essentially acts as the "featurizer" that turns unstructured text into a numerical form suitable for similarity lookup.

- **Query Time Retrieval**: When a user asks a question (query), the workflow first uses the **Embedding NIM** again – this time to embed the **query** into the same vector space as the documents. The system then performs a **vector similarity search** in the vector database to find the top-N document chunks whose embeddings are most similar to the query embedding [10]. These initial results are candidate context passages that might contain relevant information.

- **Neural Reranking**: The top retrieved passages from the vector search are next passed to the **Reranking NIM**, along with the original query. The Reranker reads each passage in full (and can consider more subtle context than the vector similarity did) and assigns a relevance score or reorders the passages based on how well each passage answers the query [7]. This step improves accuracy by making sure the *most relevant* information is identified, especially important if the initial vector search isn't perfect or if results come from different sources with different scoring algorithms [11]. The Reranking NIM is a fine-tuned language model (for example, a 3.5B-parameter Mistral model [4]) that excels at QA relevance tasks. It returns the passages sorted by relevance (and typically you would take the top few passages after reranking for use in answer generation).

- **Answer Generation**: Finally, the top **reranked** passages are given as context to the **LLM** (the generative model) which produces the answer. NVIDIA also provides LLM services (LLM NIMs) that you can use for generation [10], or you can use any other large language model. The LLM's prompt might be constructed by concatenating the top passages along with the user's question, so that the model can "ground" its answer in those passages. The result is a contextually relevant answer that cites or uses information from your documents, reducing hallucination.

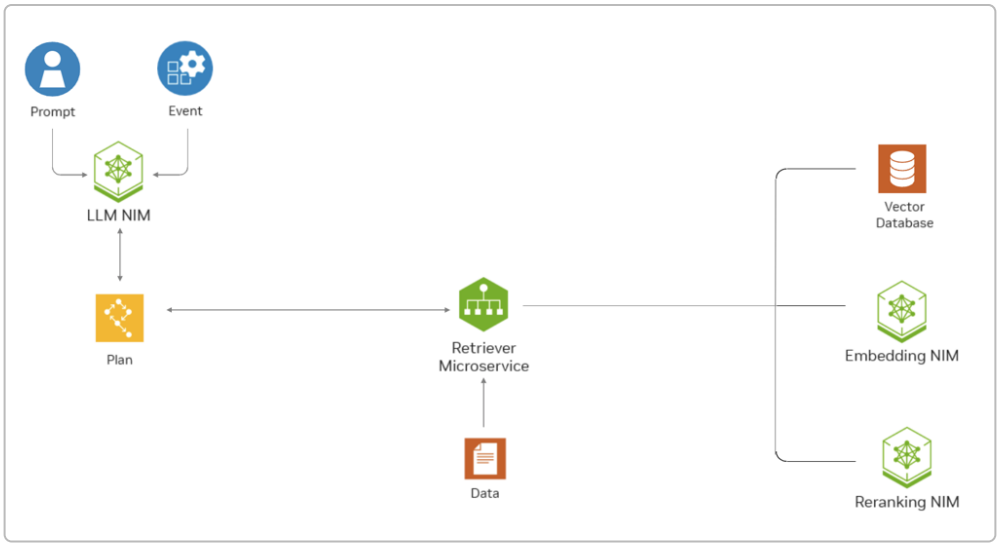

Diagram: The Retrieval-Augmented Generation pipeline with NVIDIA NeMo Retriever. In the Retrieval Pipeline (top), a user query is converted to an embedding (via the Embedding NIM) and used to fetch similar vectors from the vector database. The Reranking NIM then reorders these results by relevance before passing the top context to an LLM NIM for answer generation. (The lower Extraction Pipeline depicts ingestion of enterprise data – e.g. parsing PDFs into text using NeMo services – which is indexed by the Embedding NIM.) [6] [10]

Conceptual and Technical Details

NeMo Embedding Service (Text Embedding NIM): This service packages a state-of-the-art embedding model (such as *NV-EmbedQA-Mistral7B-v2* or a smaller Llama-based model) into a container for high-throughput inference [5]. It transforms text into a numerical vector (embedding) capturing the semantic meaning. In a RAG context, the embedding service is used at two points: (a) **Indexing** documents (each document or chunk is embedded and stored), and (b) **Query encoding** (transform the incoming question into the same vector space). The quality of these embeddings is critical — better embeddings mean that relevant documents will rank higher in the similarity search. NVIDIA's models are fine-tuned for **question-answer retrieval**, so they produce embeddings that are especially good at linking questions to answer-containing passages [12] [13]. For example, NeMo's *"EmbedQA"* models are optimized for QA tasks and support long contexts (up to 512 or 8192 tokens) for embedding lengthy passages [14] [15]. The embedding service runs on GPU (TensorRT-optimized) for speed and can handle batch requests for efficiency [16].

NeMo Reranker Service (Text Reranking NIM): This is a separate microservice that packages a *cross-attention reranking model* (for instance, a **Mistral 7B-based QA reranker trimmed to 3.5B parameters** [4] or a smaller Llama 1B model) in a container. Its job is to take a query and a list of text passages and output a ranking or scores indicating which passage is most relevant [7]. Internally, this model typically concatenates the query with each passage and uses a transformer neural network to judge relevance (much like a cross-encoder in IR literature). By fine-tuning on QA data, it can understand nuances and context better than the raw embedding similarity scores. In practice, using a reranker after vector search can significantly boost answer accuracy – NVIDIA reports up to **50% fewer incorrect answers** in pipelines that use the NeMo reranker versus ones that rely only on embedding similarity [9]. The Reranking NIM exposes an API (HTTP endpoint) where you send the query and a list of passages; it returns a list of passages sorted by relevance or accompanied by relevance scores [17] [18]. Typically you would select the top *k* passages (e.g. top 3 or 5) from this output to feed to the LLM. If your initial vector search spanned multiple data sources or modalities, the reranker also helps normalize the results by quality.

Microservice APIs and Integration: Both the embedding and reranking services are designed to be easy to integrate. They expose REST endpoints that mirror OpenAI's API style (making them compatible with existing tools and SDKs) [4] [19]. For example, the Text Embedding NIM provides a `POST /v1/embeddings` endpoint that accepts a JSON payload with your text input and model name, and returns embedding vectors. The Text Reranking NIM provides a `POST /v1/ranking` endpoint where you submit the query and candidate passages, and it returns a sorted list or scores [17] [20]. This means you can use standard HTTP calls or even OpenAI API-compatible client libraries to query the services. NVIDIA also offers integration libraries (for instance, the `langchain-nvidia-ai-endpoints` Python package) that wrap these calls for you in a LangChain-friendly interface. We'll demonstrate usage with LangChain next.

> *Technical Note:* Some NeMo embedding models (like the NV-EmbedQA series) expect an **input type** flag to distinguish between embedding a **passage** (document text) versus a **query**. They are often dual-mode models that slightly adjust embeddings depending on the use-case (document vs question) for optimal retrieval accuracy [21]. If you use these models, you must specify `input_type: "passage"` when embedding documents for indexing, and `input_type: "query"` when embedding a user question [22] [23]. (The service allows adding a `-passage` or `-query` suffix to the model name as a shortcut to indicate the mode [24] – for example, using model `"NV-Embed-QA-query"` versus `"NV-Embed-QA-`

`passage"`). Using the wrong mode can degrade retrieval performance, so it's an important detail to get right.

Tutorial: Python RAG Pipeline with NeMo Retriever Services

Let's walk through a simplified **Python example** of integrating NeMo's Embedding and Reranking services into a RAG pipeline. We will use the LangChain framework for convenience, which has built-in support for NVIDIA's NIM endpoints. This example assumes you have the NeMo services running (more on deployment in the prerequisites section) – for illustration, we'll assume an embedding service is accessible at `http://localhost:8080` and a reranker service at `http://localhost:8000`. We also assume you have an API key if using NVIDIA's hosted cloud, or none if running locally.

1. Initialize the NeMo Text Embedding service client. We use LangChain's `NeMoEmbeddings` class to connect to the embedding NIM. You need to specify the model and the endpoint URL (or an API key if using the hosted service). In this example we'll use a locally hosted model (Llama 1B QA embedder) running on `localhost:8080`:

```
from langchain_nvidia_ai_endpoints.embeddings import NeMoEmbeddings

# Connect to NeMo Retriever Embedding service (local deployment)
embedding_model = NeMoEmbeddings(
    model="nvidia/llama-3.2-nv-embedqa-1b-v2",
# embedding model ID to use (1B param Llama-based QA embedder)
    base_url="http://localhost:8080/v1",           # base URL of the embedding
NIM service (OpenAI-like API)
    batch_size=16                                  # batch size for embedding
calls (tune based on your GPU)
)
```

In this snippet, we instantiate an embedding model client that will send requests to the NeMo Embedding microservice [25]. The `model` parameter chooses which model to use (here we chose the *llama-3.2-nv-embedQA-1B-v2* model – ~1.2B parameters, suitable for demonstration). The `base_url` points to our service; by appending `/v1` we indicate the OpenAI-compatible REST API root. If you were using NVIDIA's **hosted API service** instead of a local container, you could omit the `base_url` and provide an API key, for example:

```
embedding_model = NeMoEmbeddings(model="NV-Embed-QA-003",
nvidia_api_key="YOUR_API_KEY")
```

When using the hosted service, the `NeMoEmbeddings` class knows the default cloud endpoint and just needs your NVIDIA API key for authentication [26] [27]. (The model name `"NV-Embed-QA-003"` is an alias used in the API Catalog – one can also use the full model IDs as listed in NVIDIA's docs.)

2. Embed and index documents in a vector store. With the embedding model ready, we need a vector database to store embeddings and perform similarity search. For simplicity, we'll use an in-memory FAISS index via LangChain. Suppose we have a list of text documents (`docs`) that we want to make searchable:

```
from langchain.vectorstores import FAISS

# Example document texts to index (in practice, load and chunk your actual data)
docs = [
    "NVIDIA H200 is the first GPU with 141 GB of HBM3e memory, delivering 4.8
TB/s of bandwidth.",
    "The NVIDIA Triton Inference Server supports multiple frameworks and
provides an HTTP/GRPC endpoint for AI model serving.",
    # ... (more documents)
]

# Create a FAISS vector store by embedding all documents via the NeMo embedding
service
vector_store = FAISS.from_texts(docs, embedding_model)
```

Under the hood, this will call the NeMo Embedding service for each document (batching requests in groups of 16 here) to get embeddings, and store those vectors in the FAISS index. Each embedding is a high-dimensional vector (e.g., 2048 dimensions for the Llama-1B model [28]). Once indexed, we can query this vector store with new questions.

3. Initialize the NeMo Text Reranking service client. Now we set up the reranker using LangChain's `NVIDIARerank` class. We'll connect to our local reranker service at `localhost:8000` and specify the reranker model to use (in this case, the 4B Mistral-based QA reranker):

```
from langchain_nvidia_ai_endpoints.reranking import NVIDIARerank

# Connect to NeMo Retriever Reranking service (local deployment)
reranker = NVIDIARerank(
    model="nvidia/nv-rerankqa-mistral-4b-v3",   # reranker model ID (3.5B param
Mistral model fine-tuned for QA)
    base_url="http://localhost:8000/v1"
# base URL for the reranking NIM (local service)
    # If using hosted API: provide nvidia_api_key instead of base_url
)
```

Like before, if we were using the NVIDIA hosted endpoint for reranking, we'd provide `nvidia_api_key="YOUR_API_KEY"` and omit `base_url` . By default, `NVIDIARerank` will use the cloud service when an API key is given [26] [29] . The `model` here is the full ID of the reranker model (NVIDIA provides models like `nv-rerankqa-mistral-4b-v3` and also a smaller `llama-3.2-nv-rerankqa-1b-v2` – you can choose based on your accuracy needs and GPU capacity).

4. Retrieve relevant documents for a query and apply reranking. Now we can simulate a user query and show how the two services work together. Suppose the user asks:

```
query = "What interfaces does NVIDIA Triton support?"
```

First, we use the **vector store** to get initial matches. The vector store will call the embedding service to embed the query, then perform similarity search among the stored document vectors:

```
# Step 1: Initial retrieval using vector similarity search
candidate_docs = vector_store.similarity_search(query, k=10)
print(f"Initial documents retrieved: {len(candidate_docs)}")
```

Let's say we retrieved up to 10 candidate chunks from our small database. These `candidate_docs` are LangChain `Document` objects containing the text and metadata. Next, we feed them to the **Reranker** to refine the results:

```
# Step 2: Neural reranking of the candidates
reranked_docs = reranker.compress_documents(documents=candidate_docs,
query=query)
print(f"Reranked top documents: {len(reranked_docs)}")
```

The `compress_documents` method will call the reranking NIM's API behind the scenes, sending the query and the list of passage texts [30] [31]. The service returns a reordered (and possibly trimmed) list of documents. By default, `NVIDIARerank` will return the **top 5** documents (`top_n=5` by default, configurable) [32] [33]. Each returned Document's metadata will include a `relevance_score` assigned by the reranker. For example, we can inspect the results:

```
for doc in reranked_docs:
    score = doc.metadata.get("relevance_score", None)
    snippet = doc.page_content[:100]  # first 100 characters of the doc
    print(f"Score: {score:.3f} | Passage: {snippet}...")
```

This might output something like:

```
Score: 16.625 | Passage: NVIDIA H200 Tensor Core GPU | Datasheet 1 – NVIDIA H200
Tensor Core GPU supercharges...
Score: 11.508 | Passage: NVIDIA H200 NVL is the ideal choice for customers with
space constraints...
Score: 8.258  | Passage: NVIDIA H200 Tensor Core GPU | Datasheet 2 – Memory
bandwidth is crucial for HPC applications...
...
```

(The scores above are just an example – higher means more relevant. You can see the passages have been sorted by relevance [34] [35].) In our Triton question example, the reranker would ensure any passage that actually mentions Triton's interfaces is ranked above irrelevant ones.

Now we have the **most relevant** chunks in `reranked_docs`. The final step in a full RAG pipeline would be to pass these top passages, along with the user's question, into an LLM to generate a final answer. For completeness, here's how you might do that with an NVIDIA LLM service (e.g., using `ChatNVIDIA` from the LangChain integration):

```
from langchain_nvidia_ai_endpoints import ChatNVIDIA

# Initialize an LLM (for example, a 70B Llama2 model hosted by NVIDIA API or a
local NIM)
llm = ChatNVIDIA(model="ai-llama2-70b", max_tokens=500,
nvidia_api_key="YOUR_API_KEY")

# Construct a prompt with the top reranked context
context_text = "\n".join([doc.page_content for doc in reranked_docs])
prompt = f"Context:\n{context_text}\n\nQuestion: {query}\nAnswer:"
response = llm.invoke(prompt)
print(response.content)
```

In this prompt, we supply the `Context` (the documents returned by the reranker) and the question; the LLM will ideally use the context to produce an informed answer. The NeMo LLM NIMs can similarly be self-hosted or used via the NVIDIA API Catalog. (Using LangChain's `ChatNVIDIA` will handle calling the LLM service API [36].)

Note: You don't *have* to use LangChain – you could call the NeMo NIM REST endpoints directly using `requests` or any HTTP client. For example, to embed text via REST you would `POST` to `http://localhost:8080/v1/embeddings` with a JSON payload like:

```
{
  "model": "nvidia/nv-embedqa-e5-v5",
  "input": ["Your text to embed"],
  "input_type": "passage"
}
```

and get back a JSON with the embeddings. For reranking, you'd call `POST http://localhost:8000/v1/ranking` with a payload containing the query and an array of passages (as shown in the cURL example) [17] [37]. The LangChain wrappers we used (`NeMoEmbeddings` and `NVIDIARerank`) simply abstract these API calls into convenient Python classes.

Prerequisites for Using NeMo Retriever Services

To access and run NVIDIA's NeMo Retriever embedding and reranking services, make sure you have the following prerequisites in place:

- **NVIDIA NGC Account and API Access:** You'll need an NVIDIA account (free) to either pull the container images or use the hosted API. NVIDIA provides a cloud **API Catalog** where you can test these models without hosting them yourself. To use it, create a free account on the NVIDIA API catalog and generate an API key [38]. This involves logging in to NGC (NVIDIA GPU Cloud), selecting a model from the catalog, and clicking "Get API Key" – the key will be used to authenticate your requests. Set the key as an environment variable (e.g. `NVIDIA_API_KEY`) so that client libraries can pick it up [39].

- **NVIDIA AI Enterprise License (for self-hosting):** While initial experimentation can be done on the hosted API for free, deploying the NeMo NIM services on your own infrastructure may require an NVIDIA AI Enterprise license. NVIDIA makes the NIM container images available through the NGC private registry as part of the AI Enterprise suite [40]. In practice, this means that after testing a model in the API Catalog, an enterprise customer can "export" the model (pull the Docker container) and run it on-premises or in their cloud, but only if they have the proper licensing. If your organization has NVIDIA AI Enterprise, you can obtain the containers (each model is a separate Docker image on NGC, e.g., `nvcr.io/nvidia/llama-3.2-nv-embedqa-1b-v2:latest`) and deploy them with Docker or Kubernetes. *(Individual developers can often get trial access or use the hosted endpoints – the licensing primarily matters for production deployment and support.)*

- **Hardware and Drivers:** To run the embedding or reranker microservices yourself, you need a machine with an **NVIDIA GPU** that has enough memory for the chosen model. Smaller models like the 1B parameter Llama embedding or reranker can run on a 16–24 GB GPU (e.g. NVIDIA RTX 4090 or A10G) [41] [42], while the larger 7B models may require 24 GB or more (and can benefit from 80 GB A100/H100 GPUs for max performance) [43] [44]. Ensure you have recent NVIDIA drivers and **Docker** installed (the NIM containers use CUDA and Triton; Docker 24+ is recommended [45]). NVIDIA also provides Helm charts for deploying on Kubernetes [46] if you prefer a cluster setup. In any case, you'll want CUDA-capable GPUs and ideally NVIDIA's container toolkit set up for GPU passthrough to Docker.

- **Deployment of NIM Services:** Once you have access to the container images (either via `docker pull` from NGC or an exported model from the API catalog), you will run the embedding and reranking services as separate processes (containers). Each service listens on a port for API requests (by default, many NIM containers use port 8000 for their APIs, but you can map to any host port). For example, you might run the embedding service container and publish it at `localhost:8080`, and the reranker at `localhost:8000`, as we assumed in the code. The documentation provides commands and Helm charts to deploy these – for a simple local Docker run you might do: `docker run -p 8080:8000 nvcr.io/nvidia/llama-3.2-nv-embedqa-1b-v2:latest` (along with necessary environment variables or volume mounts if any). Make sure each service is up and healthy (you can test by GET on `/v1/models` or similar to see if it responds).

- **LangChain and Client Libraries:** If you plan to integrate via LangChain (as we did above), install the `langchain` library and NVIDIA's integration package `langchain-nvidia-ai-endpoints`. This provides the `NeMoEmbeddings`, `NVIDIARerank`, and `ChatNVIDIA` classes we used. You can install these with pip [47]. Alternatively, you can use NVIDIA's own SDKs or simply direct HTTP calls. The LangChain route is convenient for Python development, but not required.

- **Model-Specific Considerations:** Be aware of which model variant you are using for each service. Different models have different maximum input lengths and performance profiles. For instance, *Llama-3.2-NV-EmbedQA-1B-v2* supports up to 8192 tokens context for embeddings [14], whereas *Mistral7B* based ones might support 512 tokens. The reranker models also have limits (e.g. one model supports up to 512 tokens passages, another up to 8192) [48]. If your documents are large, you should chunk them to fit the model's context size. Also remember to use the correct `input_type` or model suffix for query vs passage embedding as noted earlier (this is a common pitfall).

In summary, NVIDIA's NeMo Retriever embedding and reranking services can be seamlessly inserted into a RAG pipeline to improve retrieval quality. The embedding NIM creates high-quality vector representations of your data and queries [12], and the reranking NIM ensures the retrieved documents are truly relevant [8] before you generate answers. By deploying these microservices (or using NVIDIA's hosted endpoints) and integrating them with your application (via LangChain or direct API calls), you can build an enterprise-grade RAG system that is both **accurate** and **scalable**. Just ensure you have the necessary NVIDIA account setup, appropriate hardware or cloud access, and you follow the model usage guidelines. With those pieces in place, you're ready to build a RAG workflow that takes advantage of NVIDIA's optimizations for fast, accurate retrieval on your own data. [40] [38]

Sources: NVIDIA NeMo Retriever documentation [13] [10], NVIDIA developer blog and LangChain integration guides [49] [50], DataRobot and other integration notes. The code example above is adapted from NVIDIA's RAG toolkit and LangChain playbooks [50] [25].

---

[1] [38] [47] [49] Tips for Building a RAG Pipeline with NVIDIA AI LangChain AI Endpoints | NVIDIA Technical Blog
https://developer.nvidia.com/blog/tips-for-building-a-rag-pipeline-with-nvidia-ai-langchain-ai-endpoints/

[2] [3] [6] [8] [9] [10] [12] [13] NeMo Retriever | NVIDIA Developer
https://developer.nvidia.com/nemo-retriever

[4] [7] [11] [46] Overview of NeMo Retriever Text Reranking NIM — NeMo Retriever Text Reranking NIM
https://docs.nvidia.com/nim/nemo-retriever/text-reranking/latest/overview.html

[5] [19] Overview of NeMo Retriever Text Embedding NIM — NeMo Retriever Text Embedding NIM
https://docs.nvidia.com/nim/nemo-retriever/text-embedding/latest/overview.html

[14] [15] [28] [41] [42] [43] [44] Support Matrix for NeMo Retriever Text Embedding NIM — NeMo Retriever Text Embedding NIM
https://docs.nvidia.com/nim/nemo-retriever/text-embedding/latest/support-matrix.html

[16] [25] NVIDIA NeMo embeddings | LangChain
https://python.langchain.com/v0.1/docs/integrations/text_embedding/nemo/

17 18 20 30 31 37 Use NeMo Retriever Text Reranking NIM — NeMo Retriever Text Reranking NIM
https://docs.nvidia.com/nim/nemo-retriever/text-reranking/latest/using-reranking.html

21 22 23 24 API Reference for NeMo Retriever Text Embedding NIM — NeMo Retriever Text Embedding NIM
https://docs.nvidia.com/nim/nemo-retriever/text-embedding/latest/reference.html

26 27 29 32 33 NVIDIARerank — LangChain documentation
https://api.python.langchain.com/en/latest/nvidia_ai_endpoints/reranking/langchain_nvidia_ai_endpoints.reranking.NVIDIARerank.html

34 35 39 45 50 LangChain Playbook — NeMo Retriever Text Reranking NIM
https://docs.nvidia.com/nim/nemo-retriever/text-reranking/1.3.0/playbook.html

36 40 ChatNVIDIA | LangChain
https://python.langchain.com/docs/integrations/chat/nvidia_ai_endpoints/

48 Support Matrix for NeMo Retriever Text Reranking NIM — NeMo Retriever Text Reranking NIM
https://docs.nvidia.com/nim/nemo-retriever/text-reranking/latest/support-matrix.html