ChatGPT

Implementing NVIDIA NIM and RAG with NeMo

xyz company is exploring NVIDIA's **NIM (NVIDIA Inference Microservices)** and **RAG (Retrieval-Augmented Generation)** to build internal solutions for knowledge retrieval, document summarization, and code generation. This tutorial will explain these concepts in simple terms, show how NVIDIA's **NeMo** framework integrates with them, and walk through a Python-based example of a RAG pipeline. We'll use clear language, diagrams for clarity, and annotated code blocks to guide you step by step.

What is NVIDIA NIM (Inference Microservices)?

NVIDIA **NIM** is a set of **pre-built, accelerated inference microservices** that package AI models (like large language models, embedding models, etc.) into ready-to-deploy containers with standard APIs [1] [2]. In practical terms, a NIM is a Docker container running a model (for example, a GPT-based chatbot or a text embedding model) that you can deploy on any NVIDIA GPU-powered infrastructure (cloud or on-premises). Each NIM microservice exposes a web API (compatible with popular standards like OpenAI's API) so that developers can easily integrate it into applications without worrying about the underlying model optimization or GPU details [3]. Key benefits of NIM include scalability, support for many model architectures (e.g., GPT, Llama, etc.), and enterprise features like security and monitoring [4].

In simple terms: NIM lets you **self-host powerful AI models as microservices**. Instead of calling OpenAI's API in the cloud, you can run NVIDIA's optimized model containers on your own Azure servers or Kubernetes cluster, and use them through a convenient API. This is especially useful if you need to keep data in-house or want to optimize performance and cost by using your own GPUs. For example, NIM provides microservices for: - **LLMs (Large Language Models)** – for text generation or chat (sometimes called "Reasoning" NIMs). - **Text Embedding models** – to convert text into vector embeddings for semantic search. - **Text Reranking models** – to improve search results relevance. - **Others** like speech recognition, vision, etc., but our focus here is on text-based services.

NVIDIA NeMo (the open-source framework) underpins many of these microservices. NeMo is an end-to-end toolkit for building and deploying generative AI models [5]. In the context of NIM, NeMo provides the models and pipeline logic that run inside these microservices. Essentially, NIM is how NVIDIA packages and serves NeMo models in production. For instance, **NeMo Retriever** is a collection of NIM microservices specialized for retrieval tasks (more on this shortly) [6].

What is Retrieval-Augmented Generation (RAG)?

Retrieval-Augmented Generation (RAG) is a technique that **combines a large language model with a retrieval system** to improve the accuracy and relevance of the model's outputs [7]. Instead of relying *solely* on the LLM's built-in knowledge (which may be outdated or limited), a RAG system **retrieves relevant information** from an external knowledge source (documents, databases, code repositories, etc.) and provides that information to the LLM as additional context when generating an answer. This helps ground the LLM's response in factual, up-to-date data and reduces "hallucinations" (irrelevant or incorrect outputs).

How RAG works (in simple terms): Imagine you have a big corporate wiki or a set of PDF manuals. With RAG, when a user asks a question, you don't just ask the LLM directly. Instead, you first **search** your document collection for parts that might contain the answer: 1. **Indexing phase:** Beforehand, all documents are broken into chunks and turned into **vectors** (numerical representations) by an **embedding model**. These vectors are stored in a special database (a **vector index**) that allows similarity search. 2. **Retrieval phase:** When a query comes in, the same embedding model converts the query into a vector, and the system finds the closest matching document vectors (i.e. the most relevant text chunks) from the index. 3. **Generation phase:** The content of those retrieved text chunks is then given to the LLM *along with the user's query*. The LLM uses this extra context to generate its answer.

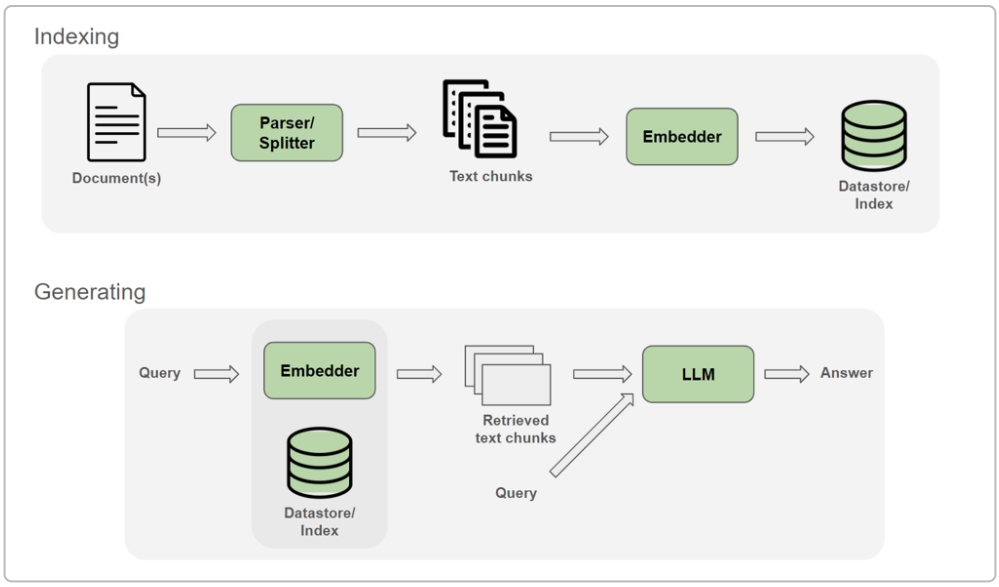

Illustration of a RAG pipeline: In the indexing stage (top), documents are split into text chunks, embedded into vectors, and stored in an index. In the query stage (bottom), a user's query is embedded and used to retrieve matching text chunks from the index, which are then fed into the LLM to produce a context-informed answer. [8] [7]

This approach is powerful for enterprise use cases. For example: - **Knowledge Retrieval Q&A:** Instead of training a custom model on all company data, RAG can use a pre-trained LLM and just supply relevant facts from company documents to answer user questions [9]. - **Document Summarization:** If you need to summarize a large document or a collection of documents, a RAG setup can retrieve the most important parts of the text for the LLM to summarize, rather than feeding the entire text at once. This makes summarization more efficient and focused (e.g., "summarize everything about *X* from our knowledge base" will retrieve passages about *X* first). - **Code Generation/Assistance:** An LLM (like a code-generating model) augmented with retrieval can pull in relevant code snippets or documentation from your codebase. This way, when asking for a function or troubleshooting an error, the model has access to the actual internal APIs or previous code examples, resulting in more accurate and customized code suggestions.

In summary, RAG is a way to **teach** a generative model about your specific data on-the-fly, rather than **train** it into the model's weights. It's often more tractable and cost-effective than full fine-tuning for domain-specific knowledge [9], and it ensures the LLM's answers stay up-to-date with your data sources.

How NeMo Integrates NIM and RAG

NVIDIA's NeMo framework provides built-in support to create RAG pipelines using NIM microservices. NeMo's **Retriever** component is essentially a bundle of microservices (built with NIM) that handle the retrieval part of RAG [6]. These include:

- **NeMo Retriever Text Embedding NIM:** a microservice that generates high-quality text embeddings. This is used to vectorize documents and queries for semantic search [10]. (For example, NVIDIA offers models like `nv-embedada` or `nv-embedqa` which are fine-tuned embedding models for Q&A context).
- **NeMo Retriever Text Reranking NIM:** an optional microservice that re-orders or filters the retrieved results to improve relevance [11]. A reranker model takes the initial retrieved texts and the query, and scores them so that the most relevant passages can be identified. This step can improve accuracy, especially if your initial search pulled in some less-relevant chunks.
- **NIM for LLMs (Reasoning NIM):** the microservice that hosts the actual large language model for generation (e.g., a Llama or GPT model). This is what produces the final answer, given the user query + retrieved context.

All these pieces work together in a NeMo RAG pipeline. NeMo provides orchestration logic (and integration with tools like LangChain or LlamaIndex) to connect the dots: from calling the embedder service, to searching a vector store, to constructing a prompt for the LLM service [8] [12]. The **NeMo Data Store** service is another piece (a vector database for storing embeddings) included in the NeMo platform, though you can also use third-party vector databases like Milvus, Redis, FAISS, etc., depending on your needs.

To clarify, here's how these components come together for an enterprise RAG application: - **Indexing step:** Use the Text **Embedding NIM** to encode your documents into vectors and store them in a **vector database** (this could be an in-memory index or a scalable DB). This step is typically done in advance (offline or periodically) to prepare your knowledge base. - **Query (Retrieval) step:** For each query, use the same Text Embedding NIM to encode the question. Query the vector database for similar vectors, retrieving the top-K relevant document chunks. Optionally, send these candidates through the **Reranking NIM** to fine-tune which passages are most relevant [11]. - **Generation step:** Feed the final selected text chunks (as context) into the **LLM NIM** along with the user's query, typically by constructing a prompt that says something like: "*Here are some relevant documents:* `<doc snippets>` *\nAnswer the user's question:* `<question>`". The LLM then generates a response that incorporates the provided context.

NeMo makes it easier to implement this because it offers the NIM microservices (so you don't have to train your own models) and a unified API to call them. In fact, NIM microservices expose an **OpenAI-compatible API** interface [3]. This means you can call these services using standard tools or SDKs as if you were calling OpenAI's endpoints – a deliberate design to simplify integration. For example: - The **LLM NIM** can be accessed with an endpoint like `http://<host>:8000/v1/chat/completions` (for chat models) or `.../completions` (for raw text models), accepting the same JSON format as OpenAI's API. - The **Embedding NIM** listens at an endpoint like `http://<host>:8001/v1/embeddings`, again similar to OpenAI's embedding API.

Because of this, one integration path is using the **LangChain** library with NVIDIA's extension. NVIDIA provides a package `langchain-nvidia-ai-endpoints` which has convenience classes to call NIM

services within LangChain flows [13] [14]. We'll use that in our example, but keep in mind you could also use direct HTTP calls or the OpenAI Python SDK (pointing it to your NIM URL and API key) if you prefer.

Step-by-Step: Building a RAG Pipeline with NIM and NeMo

Let's go through a simple example of implementing RAG in Python. This example will use: - A **Text Embedding NIM** to generate embeddings, - A **vector store** (FAISS in-memory index) to store and search those embeddings, - An **LLM NIM** to generate answers, - The **LangChain** framework to tie these together (for convenience).

> **Prerequisites:** For this to run, you need access to NVIDIA's NIM services. You can either deploy the services on Azure (we discuss options later) or use NVIDIA's hosted API (via the NVIDIA API Catalog) with an API key. The code below assumes you have a local NIM deployment (or port-forwarded service) on `localhost` for both the embedder and LLM. Adjust the `base_url` to your endpoints, and ensure the `model` names match the ones you deployed or have access to.

1. Install necessary libraries. We'll need the core LangChain libraries and NVIDIA's extension package. You can install these via pip:

```
pip install langchain-core langchain-community langchain-nvidia-ai-endpoints
faiss-cpu numpy
```

The `langchain-nvidia-ai-endpoints` package includes connectors for NIM services. We also install `faiss-cpu` for a simple vector store, and `langchain-community` which contains various document loaders and other utilities.

2. Connect to the NIM microservices. We create instances of the embedding and LLM classes with the appropriate URLs and model names. For example, if we have an embedding model `nvidia/llama-3.2-nv-embedqa-1b-v2` running on port 8001, and a Llama-3.1 8B chat model on port 8000:

```
import os
from langchain_nvidia_ai_endpoints import NVIDIAEmbeddings, ChatNVIDIA

# If using NVIDIA's cloud API instead of self-hosted, set your API key:
# os.environ["NVIDIA_API_KEY"] = "<your NGC API key>"

# Connect to NeMo Text Embedding NIM (replace base_url with your service URL)
embedding_model = NVIDIAEmbeddings(
    model="nvidia/llama-3.2-nv-embedqa-1b-v2",        # model name of the
embedding NIM
    base_url="http://localhost:8001/v1"               # endpoint URL of
embedding service
)
```

```
# Connect to NIM for LLM (chat model)
llm_model = ChatNVIDIA(
    model="meta/llama-3.1-8b-instruct",              # model name of the LLM
NIM
    base_url="http://localhost:8000/v1"              # endpoint URL of LLM
service
)
```

A couple of notes on the above: - We used `ChatNVIDIA` for the LLM since it's a chat-capable model (accepts messages and can do multi-turn conversations). If you were using a base completion model, a similar class or the same class can handle it by formatting the inputs appropriately [14]. - The `model` strings (e.g., `"meta/llama-3.1-8b-instruct"`) identify which model weights the service should use. These would match what you deployed. NVIDIA NGC (NVIDIA's registry) provides many pretrained NIM containers; for example, `meta/llama-3.1-8b-instruct` refers to a Llama 3.1 model with 8B parameters. - If you were using the NVIDIA cloud API (not self-hosting), you wouldn't provide a base_url, you'd just use `ChatNVIDIA(model="model/name")` with your `NVIDIA_API_KEY` set, and it would call NVIDIA's hosted endpoint [15]. In our case, we assume self-hosted for an Azure deployment.

3. Prepare your data (documents). For demonstration, let's say we have some documents we want to use as the knowledge base. This could be a collection of text files, PDFs, or even code files. LangChain provides loaders to read various formats. Here, for simplicity, let's assume we have a single text document (you can replace this part with actual file loading):

```
from langchain.docstore.document import Document

# Example document content (this could be loaded from a PDF, etc.)
doc_text = """
    xyz company  is a firm specializing in data science and AI solutions.
One of its teams focuses on Azure cloud-based deployments of AI models.
It leverages NVIDIA's NeMo framework and NIM microservices for scalable AI
inference.
"""
documents = [Document(page_content=doc_text)]
```

We wrap the text in a LangChain `Document` object. In a real scenario, you might use `PyPDFLoader` (for PDFs), or other loaders to get documents from files or URLs [16]. After loading, it's common to **split documents into chunks** (especially if they are large), so each chunk can be embedded and retrieved independently. For example:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Split documents into smaller chunks for embedding
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
```

```
doc_chunks = text_splitter.split_documents(documents)
print(f"Number of chunks: {len(doc_chunks)}")
```

This will break the document into pieces of at most ~500 characters, overlapping by 50 characters to maintain context continuity. Splitting ensures that each chunk is reasonably sized for the embedding model and relevant retrieval [17] [18].

4. Embed the document chunks and store in a vector index. Using our `embedding_model`, we can convert each chunk of text into an embedding vector:

```
# Extract raw text from chunks
texts = [chunk.page_content for chunk in doc_chunks]

# Embed all the chunks (get a list of vectors)
embeddings = embedding_model.embed_documents(texts)
print(f"Generated {len(embeddings)} embeddings, each of length
{len(embeddings[0])}")
```

The `embed_documents` method sends the text to the Embedding NIM service and returns vectors (lists of floats). Now, we need to store these vectors so we can search them by similarity. We'll use FAISS (an in-memory vector store) for simplicity:

```
from langchain_community.vectorstores import FAISS

# Create a FAISS vector store from our chunks and embeddings
vector_store = FAISS.from_texts(texts, embedding_model)
# (Alternatively: FAISS.from_documents(doc_chunks, embedding=embedding_model))
```

This initializes a FAISS index under the hood and stores our chunk texts along with their embedding vectors [19]. If you had a large dataset or needed persistence, you might use a persistent database (Milvus, Pinecone, etc.), but FAISS is fine for an example or smaller scale deployment.

5. Perform a retrieval and generation (ask a question). Now the system is set up. We can simulate a user query and run it through the RAG pipeline:

```
# Example user question
query = "What does  xyz company  use NVIDIA NeMo for?"

# Step 5a: Embed the query to a vector
query_vector = embedding_model.embed_query(query)

# Step 5b: Use the vector store to find similar document chunks
matched_docs = vector_store.similarity_search_by_vector(query_vector, k=2)
# Get the text content of the top matches
```

```
context_snippets = [doc.page_content for doc in matched_docs]
print("Retrieved context:", context_snippets)
```

We take the query, embed it via `embed_query` (similar to `embed_documents` but for single text). Then we ask FAISS for the top-2 similar vectors. This returns the most relevant chunks of our documents. These chunks in `context_snippets` will serve as the **context** for the LLM.

Finally, we send the question and the retrieved context to the LLM. We need to format a prompt that the LLM will understand. Since we are using a chat model, we can provide a system message with instructions and a user message. The LangChain `ChatPromptTemplate` is one way to do this. Here's a simple prompt instructing the AI to use the context:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

# Construct a prompt template with placeholders for context and question
prompt = ChatPromptTemplate.from_messages([
    ("system",
     "You are an AI assistant. Answer the user's question based on the given
context. "
     "If the context does not contain the answer, say you don't know. "
     "Context: {context}"),
    ("user", "{question}")
])

# Format the context into the prompt
formatted_prompt = prompt.format_prompt(context="\n\n".join(context_snippets),
question=query)
# The prompt object can combine with the llm in a chain, but we'll call the llm
directly for simplicity:
response = llm_model(formatted_prompt.to_messages())
print("AI Answer:", response.content)
```

In the above: - We defined a system message instructing the model to use the context and not hallucinate answers (this is a basic form of instruction; you can refine it as needed). - We then provided the user's question as the user message. - We used `prompt.format_prompt(...)` to fill in the `{context}` and `{question}` placeholders with our retrieved snippets and the actual question. - Finally, we directly invoked `llm_model` with the formatted messages. (The `ChatNVIDIA` object is callable like a function; you pass in a list of message dicts or another supported format. Alternatively, LangChain allows creating a chain that pipes the retriever and prompt to the LLM as shown in NVIDIA's docs [20] [21].)

The `response.content` would contain the answer generated by the LLM, which ideally uses the context. For example, given our toy data above, the answer might be: " xyz company *uses NVIDIA NeMo to deploy and*

scale AI models via NIM microservices on Azure." (The exact output will depend on the model and how the prompt is worded.)

That's it – we've implemented a basic RAG flow: - We indexed a document (embedding & storing). - We took a query, retrieved relevant info. - We prompted an LLM with that info to get an answer.

Of course, in a real deployment there are more considerations: you may have many documents and need a proper database, you might want to chain multiple prompts or refine answers (e.g., ask the model to cite sources, or do a second pass to refine the answer), and you'll integrate this into an application (perhaps exposing it via an API or a chat interface). NVIDIA's RAG reference implementations include features like multi-turn conversation handling, citations, guardrails for safety, etc. [22] [23], but the core idea remains the same as this simple pipeline.

Deployment on Azure and Alternative Environments

Kritika has access to an Azure subscription, so how can she run NIM and NeMo there? There are a few options to consider:

- **Azure Kubernetes Service (AKS):** NVIDIA provides a *NIM Operator* and Helm charts to deploy NIM microservices on Kubernetes [24] [25]. This is a great approach if you want to scale out or run multiple services. In fact, NVIDIA has step-by-step guides for deploying NIM on AKS [26]. You would typically:
- Set up an AKS cluster with nodes that have NVIDIA GPUs (or use Azure NC-series VM nodes).
- Install the NVIDIA NGC Helm charts or NIM Operator on the cluster.
- Deploy the specific NIM services you need (e.g., one for LLM, one for embedding, etc.) by creating the corresponding Kubernetes Custom Resources (the operator makes this easier by handling the container details [27] [28]).
- Deploy a vector database service (like Milvus) in the cluster, if using one for retrieval [29].
- Optionally deploy NVIDIA's sample RAG application (they provide a Helm chart for a multi-turn QA chatbot that wires everything together [30] [31]).

- **Azure Machine Learning (AzureML):** If you prefer a managed approach without handling raw Kubernetes, NVIDIA also has a guide to deploy NIM services on AzureML [26]. Essentially, you can use AzureML endpoints or VMs to host the Docker containers of the NIM microservices. AzureML can manage the environment and you can scale or version deployments through AzureML's interface.

- **Azure VM or Container Instances:** For a simple trial, you could also use an Azure VM with GPUs (e.g., a Standard_NC series VM) and run the NIM container images directly via Docker. For example, to run an LLM NIM container, you'd obtain the image (from NVIDIA NGC, which requires login/API key) and run it with the proper commands (specifying model name, etc.). This might be the quickest way to get started if Kubernetes is too heavy for initial testing. Just be mindful of GPU drivers and NVIDIA Container Toolkit setup on the VM.

- **NVIDIA API Catalog (cloud option):** As mentioned earlier, you have the choice to use NVIDIA's hosted inference service (NVIDIA AI Foundations/API Catalog) by obtaining an API key [32]. This doesn't require deploying anything – you simply call the API over the internet. The downside is that

your data queries will go to an external service and there may be cost or rate limitations (the API key comes with some free credits). For internal organizational use with sensitive data, self-hosting via NIM is usually preferred.

In Kritika's case, since she has an Azure subscription and likely access to GPUs, deploying NIM on an **AKS cluster** might provide the right balance of scalability and manageability for enterprise use. The NIM Operator can simplify lifecycle management (e.g., scaling up the LLM service or updating models) [33]. It also supports features like model caching (so that if you spin up multiple instances of a service, the model weights are downloaded only once and shared, saving memory) [34].

One thing to note: **Access to NIM containers** – NVIDIA NIM is part of the NVIDIA AI Enterprise offering, which means the Docker images and Helm charts are gated. Kritika should ensure she or her organization has the necessary NGC (NVIDIA GPU Cloud) access. Being an NVIDIA Developer Program member is often enough to pull certain images [35], but some might require an enterprise license. It's worth checking the documentation and possibly reaching out to NVIDIA reps if there's any issue pulling images.

If Azure setup is not immediately available for testing, an alternative is to use local resources: - You could simulate on a local machine with a suitable GPU by running NIM containers via Docker Compose or kind (Kubernetes in Docker) for a quick POC. - Or use smaller open-source models in a similar RAG pipeline to prototype the flow, then swap in the NVIDIA NIM services when moving to Azure GPUs (for instance, use SentenceTransformers for embedding and a smaller LLM locally, just for a functional test of the pipeline logic).

Conclusion

In summary, NVIDIA's NIM and NeMo provide a robust way to implement retrieval-augmented generation in an enterprise setting with **minimal custom model training**. NIM microservices allow **plug-and-play deployment** of powerful models (for language, vision, etc.) on Azure, with standard APIs for integration [1]. RAG enables these models to leverage your organization's private data (documents, knowledge bases, code) in real time, leading to more accurate and context-aware outcomes [7].

By using NeMo's retriever components (embedding and reranking services) alongside an LLM service, you can build applications for **knowledge retrieval** (like internal chatbots that actually know your company's data), **document summarization**, and **code generation assistance** that are tailored to your domain. We walked through a simple Python example where we indexed documents and used a query to get an answer – this blueprint can be expanded to multi-turn conversations, larger document sets, or integrated with user interfaces.

Kritika can follow the tutorial above to prototype a solution and then consider deploying it on Azure for wider testing. NVIDIA's documentation and tools (like the NIM Operator and Azure deployment guides) will be helpful in that journey [26]. With this approach, xyz company can harness state-of-the-art NVIDIA AI models while keeping data secure and achieving high performance on Azure's GPU infrastructure.

References:

1. NVIDIA Developer Blog – *Enhancing RAG Applications with NVIDIA NIM* [9] [1]

2. NVIDIA NeMo Documentation – *RAG Pipeline Overview* [7] [8]
3. NVIDIA NeMo Retriever Documentation – *Overview & NIM Microservices* [6] [10]
4. NVIDIA NeMo Text Embedding NIM Documentation – *OpenAI API compatibility & Usage* [3] [14]
5. NVIDIA NIM Operator Documentation – *Deploying NIM on Azure (AKS/AzureML)* [26]

---

[1] [4] [9] Enhancing RAG Applications with NVIDIA NIM | NVIDIA Technical Blog
https://developer.nvidia.com/blog/enhancing-rag-applications-with-nvidia-nim/

[2] [5] [32] RAG from Scratch using NVIDIA NIM | by Mayada Khatib | Medium
https://medium.com/@mayadakhatib/rag-from-scratch-using-nvidia-nim-99142af3b7bb

[3] [10] [11] Overview of NeMo Retriever Text Embedding NIM — NeMo Retriever Text Embedding NIM
https://docs.nvidia.com/nim/nemo-retriever/text-embedding/latest/overview.html

[6] NVIDIA NeMo Retriever - NVIDIA Docs
https://docs.nvidia.com/nemo/retriever/index.html

[7] [8] RAG Pipeline Overview — NVIDIA NeMo Framework User Guide 24.07 documentation
https://docs.nvidia.com/nemo-framework/user-guide/24.07/rag/ragoverview.html

[12] [13] [14] [15] [16] [17] [18] [19] [20] [21] LangChain Playbook — NeMo Retriever Text Embedding NIM
https://docs.nvidia.com/nim/nemo-retriever/text-embedding/1.3.0/playbook.html

[22] GitHub - NVIDIA-AI-Blueprints/rag: This NVIDIA RAG blueprint serves as a reference solution for a foundational Retrieval Augmented Generation (RAG) pipeline.
https://github.com/NVIDIA-AI-Blueprints/rag

[23] Building an Intelligent Customer Support Agent with Glean APIs and NVIDIA NIM Microservices
https://www.glean.com/blog/glean-nvidia-nim-microservices-customer-support-agent

[24] [25] [27] [28] [33] [34] NVIDIA NIM Operator — NVIDIA NIM Operator
https://docs.nvidia.com/nim-operator/latest/index.html

[26] Tutorials — NVIDIA NIM for Large Language Models (LLMs)
https://docs.nvidia.com/nim/large-language-models/1.0.0/tutorials.html

[29] [30] [31] [35] Sample RAG Application — NVIDIA NIM Operator
https://docs.nvidia.com/nim-operator/latest/sample-rag.html