ChatGPT

Uso de modelos IBM Granite en un sistema RAG local

Introducción al problema y objetivos del RAG local

La **Generación Aumentada por Recuperación (RAG)** combina un modelo de lenguaje con una base de conocimiento externa para responder preguntas con información actualizada o específica [1]. En este caso se cuenta con dos modelos de IBM **Granite** descargados vía Docker: un modelo *LLM* instructivo (Granite-4.0-h-Micro, ~3B parámetros) y un modelo de *embeddings* multilingüe (Granite-Embedding-278M) para representar textos como vectores. El objetivo es integrar ambos modelos en un entorno local (solo CPU) para indexar documentos de tamaño medio en una base de vectores Postgres (extensión **pgvector**) y permitir consultas tipo examen. A continuación, se detallan las características de estos modelos, su configuración mediante Docker Model Runner y las estrategias para usarlos con frameworks como **LangChain** o **LlamaIndex** en Python.

Modelos IBM Granite 4.0 y Embeddings Granite 278M

Granite-4.0-h-Micro (3B) es un modelo *LLM* de IBM optimizado para seguir instrucciones y con capacidad multilenguaje. Forma parte de la familia Granite 4.0 y soporta contexto extenso (hasta 128k tokens) gracias a su arquitectura híbrida Mamba-2 [2] [3]. Este modelo ha sido ajustado con refuerzo y *fine-tuning* para tareas conversacionales, incluyendo *tool calling*, y está diseñado para desempeñarse bien en variadas tareas *enterprise*. En particular, Granite-4.0 Micro demuestra **capacidades generalistas sólidas** (resumen, clasificación, extracción, código, etc.) y es eficaz en **QA/RAG** (preguntas-respuestas con recuperación de datos) [4]. Es un modelo ligero (3B parámetros densos) pensado para ejecutarse localmente con recursos modestos (CPU o GPUs de gama media) manteniendo un rendimiento competitivo [5] [6].

Por su parte, el **Granite-Embedding-278M-Multilingual** es un modelo de *sentence embeddings* de 278 millones de parámetros entrenado por IBM para generar vectores semánticos de alta calidad [7]. Sigue una arquitectura tipo XLM-RoBERTa biencoder (solo codificador) que produce embeddings de **768 dimensiones** por texto [7] [8]. Está optimizado para **búsqueda semántica multilingüe y recuperación de información** en **12 idiomas** (incluyendo español, inglés, francés, alemán, portugués, chino, etc.) [9]. Esto lo hace ideal para aplicaciones de RAG *cross-lingual*, donde puede indexar y comparar textos en distintos idiomas de forma consistente [10]. El modelo fue lanzado bajo licencia Apache 2.0 y construido con datos éticamente curados para uso empresarial [11] [12]. En resumen, este embedding permite convertir documentos y consultas en vectores numéricos comparables, habilitando búsquedas por similitud cósica en Postgres/pgvector en un espacio de 768 dimensiones [7] [8].

Ambos modelos se distribuyen como **imágenes de Docker** verificadas (namespace `ai/` en Docker Hub). Granite 4.0 Micro (3B) viene en variantes cuantizadas (ej. `:3B-Q4_K_M` de ~1.8GB) aptas para CPU [13] [14], y el embedding 278M (~530MB en precisión F16) requiere unos ~0.2GB de RAM [15]. Estas imágenes utilizan Docker Model Runner para exponer una **API local compatible con OpenAI** [16], lo que simplifica su uso sin

credenciales de IBM ni HuggingFace. Antes de usar los modelos, asegúrese de **habilitar Docker Model Runner** en su entorno Docker Desktop y permitir acceso por TCP al puerto configurado (por defecto **12434**): por ejemplo: `docker desktop enable model-runner --tcp 12434` habilita el acceso API desde el host en el puerto 12434 [17]. Tras esto, puede **descargar** (`docker model pull`) y **ejecutar** (`docker model run`) los modelos Granite localmente como servicios de inferencia.

Uso de la API local de Docker Model Runner (OpenAI-compatible)

Docker Model Runner expone los modelos por medio de endpoints HTTP que imitan la especificación de la API de OpenAI, incluyendo *chat completions* y *embeddings*. Esto es extremadamente útil ya que permite reutilizar código y SDKs existentes diseñados para OpenAI, pero apuntándolos al endpoint local [18]. El endpoint base típico es `http://localhost:12434/engines/llama.cpp/v1` (si se usó el motor por defecto basado en llama.cpp) [19]. A continuación se explica cómo invocar cada modelo:

- **Obtener *embeddings***: Para generar vectores con Granite-Embedding-278M, se utiliza el endpoint `/v1/embeddings`. Por ejemplo, mediante `curl` se enviaría una petición POST con JSON indicando el modelo y el texto de entrada:

```
curl -X POST "http://localhost:12434/engines/llama.cpp/v1/embeddings" \
     -H "Content-Type: application/json" \
     -d '{ "model": "ai/granite-embedding-multilingual", "input": "Tu texto a
vectorizar aquí" }'
```

Este llamado retornará un JSON con el vector de 768 dimensiones en `data[0].embedding` [20]. Equivalentemente, usando la librería Python de OpenAI, primero configuremos la clave API (puede ser un valor dummy no vacío) y la URL base local:

```
import openai
openai.api_key = "mi_clave"  # cualquier valor no vacío
openai.api_base = "http://localhost:12434/engines/llama.cpp/v1"
respuesta = openai.Embedding.create(
    model="ai/granite-embedding-multilingual",
    input=["¿Cuál es la capital de Francia?"]
)
vector = respuesta["data"][0]["embedding"]
print(len(vector))  # debería imprimir 768
```

El código anterior ilustra la obtención de un embedding usando el modelo local (aquí vectorizando la pregunta de ejemplo). Internamente, la llamada hace un POST a `/embeddings` de la API local y devuelve el vector correspondiente [21]. *Nota:* Es importante que la columna vectorial en PostgreSQL esté definida con la dimensión correcta (768) para almacenar estos embeddings. La extensión pgvector requiere especificar el tipo como `VECTOR(768)` o similar, acorde al tamaño fijo de salida del modelo [8].

- **Generar respuestas con el LLM**: Para usar Granite-4.0 Micro como modelo generativo, se emplea el endpoint de *completions* o *chat completions*. Granite 4.0 es un modelo orientado a instrucciones/

diálogo, por lo que es conveniente usar la modalidad de chat. Siguiendo la interfaz OpenAI, se envía un historial de mensajes con roles (`system`, `user`, etc.) a `/v1/chat/completions`. Por ejemplo, utilizando la librería OpenAI en Python:

```
mensaje_sistema = {"role": "system", "content": "Eres un asistente útil."}
mensaje_usuario = {"role": "user", "content": "¿Cuál es la capital de Francia?"}
respuesta_chat = openai.ChatCompletion.create(
    model="ai/granite-4.0-h-micro",
    messages=[mensaje_sistema, mensaje_usuario]
)
texto_respuesta = respuesta_chat["choices"][0]["message"]["content"]
print(texto_respuesta)
```

Dado que ya configuramos `openai.api_base` al endpoint local, la llamada anterior envía la solicitud al modelo Granite local. El `model="ai/granite-4.0-h-micro"` indica al Model Runner qué modelo usar (cargándolo en memoria si no lo está). En este ejemplo, el modelo debería responder `"La capital de Francia es París."` u otra formulación en función de su conocimiento. El Docker Model Runner soporta este modo *chat* interactivo; de hecho, se puede iniciar una sesión REPL con `docker model run ai/granite-4.0-micro` para probar manualmente múltiples prompts [22]. Bajo el capó, **la API es compatible con OpenAI**: el Model Runner recibe la petición y la procesa usando el modelo local, de forma que la respuesta se obtiene como si fuera de la API de OpenAI [18] [23]. Es necesario proporcionar algún valor en `api_key` porque el SDK de OpenAI lo exige, aunque el Model Runner no lo valida realmente (podemos usar un placeholder) [24].

En resumen, gracias a esta API local, **no se requieren credenciales de IBM ni HuggingFace**. Se aprovechan clientes existentes (como `openai` Python SDK, o equivalentes en otros lenguajes) apuntándolos a `http://localhost:12434/...` para invocar los modelos Granite en local [25] [26]. Esto resuelve problemas comunes de configuración de embeddings: por ejemplo, asegurarse de usar exactamente el mismo modelo tanto para indexar documentos como para embeddear consultas (evitando incompatibilidades de dimensión o vector espacio) [27] [28].

Integración con LangChain o LlamaIndex y Postgres (pgvector)

Con la comunicación API funcionando, podemos integrar los modelos en un **pipeline RAG** usando frameworks populares:

- **Usando LangChain**: LangChain permite definir fácilmente un vector store respaldado por Postgres. Por ejemplo, tras obtener una instancia de embeddings (`embeddings = OpenAIEmbeddings(model="ai/granite-embedding-multilingual", openai_api_base="http://localhost:12434/...", api_key="dummy")`), se puede crear una tienda PGVector así:

```
from langchain_postgres import PGVector
vector_store = PGVector(
    embeddings=embeddings,
```

```
    collection_name="mis_documentos",
    connection_string="postgresql+psycopg2://usuario:pass@localhost:5432/dbname"
)
```

Este objeto `PGVector` se encargará de almacenar y buscar embeddings en la tabla Postgres configurada (creándola si es necesario). En el tutorial de Zilliz se demuestra esta configuración para pgvector: se pasa la instancia de `embeddings` y la cadena de conexión, junto con un nombre de colección (tabla) [29]. Luego, se pueden **indexar documentos** dividiéndolos en fragmentos (usando, por ejemplo, `RecursiveCharacterTextSplitter`) y llamando a `vector_store.add_texts([...])`, lo cual internamente usará `embeddings.embed_documents` para generar vectores e insertarlos en la base de datos. Finalmente, en tiempo de consulta, se utiliza `vector_store.similarity_search(query, k)` que: (1) llamará a `embeddings.embed_query(query)` para vectorizar la pregunta, (2) ejecutará una búsqueda de vecinos más cercanos (k-NN) en Postgres (por similitud coseno, usualmente) para recuperar los fragmentos más relevantes, y (3) retornará esos documentos. Estos documentos relevantes pueden concatenarse en un *prompt* junto con la pregunta del usuario, y pasarse al LLM Granite para generar la respuesta final. En la práctica, LangChain ofrece abstracciones como **Retrievers** o cadenas QA que simplifican estos pasos, pero bajo el capó sigue esta lógica [30] [31].

Ejemplo: Integrando todo, podríamos construir una cadena QA así: `qa_chain = RetrievalQA.from_chain_type(llm=chat_llm, retriever=vector_store.as_retriever())`. Aquí `chat_llm` sería una instancia de `ChatOpenAI` configurada para el modelo Granite local (por ejemplo, `ChatOpenAI(model="ai/granite-4.0-h-micro", openai_api_base="http://localhost:12434/...", api_key="x")`), y el retriever viene del PGVector. De este modo, al llamar `qa_chain({"query": pregunta})`, LangChain obtendrá embeddings de la pregunta, buscará en Postgres los textos más similares y luego formateará la pregunta + contexto para que el LLM genere la respuesta. Esta arquitectura es modular y puede tunearse (por ejemplo, usar re-rankers, *prompt* específico de sistema, límite de tokens, etc.).

- **Usando LlamaIndex (GPT Index)**: LlamaIndex es otra librería que facilita la construcción de pipelines RAG. La ventaja es que permite un manejo más transparente de índices y puede ofrecer un flujo más personalizable. LlamaIndex **soporta nativamente Postgres/pgvector** a través de su clase `PGVectorStore` [32]. El proceso sería similar: crear un `PGVectorStore` conectado a la base de datos Postgres con pgvector, y construir un índice vectorial. Por ejemplo, su documentación muestra cómo inicializar la store: `vector_store = PGVectorStore.from_params(database="mi_db", table="mis_docs", user="...", password="...", ...)` y luego se crea un `VectorStoreIndex` pasando ese store y un objeto de embeddings (embed_model) [33] [34]. En LlamaIndex, se puede especificar un modelo de embedding personalizado fácilmente. Una forma es aprovechar el mismo enfoque OpenAI: estableciendo las variables de entorno `OPENAI_API_KEY` (dummy) y `OPENAI_API_BASE` apuntando al Model Runner, LlamaIndex puede utilizar su `OpenAIEmbedding` por defecto para llamar al endpoint local. Alternativamente, se podría implementar una clase de embeddings sencilla que llame al endpoint REST manualmente (usando `requests` o similar) y la pasar como `embed_model`. Una vez construido el índice (index.construct_index(...) o similar), las consultas se resuelven con métodos del index (`index.query("mi pregunta")`), que internamente realizan la búsqueda vectorial en Postgres y consultan al LLM configurado para generar la respuesta. Cabe

destacar que, según algunos desarrolladores, LlamaIndex ofrece abstracciones más consistentes para RAG en 2025, mientras LangChain tiende a fragmentarse en muchos componentes [35] [36]. No obstante, **ambos frameworks pueden lograr la solución deseada**. La elección puede depender de la preferencia personal o detalles de implementación: por ejemplo, LangChain ya integra comodamente PGVector y cuenta con `langchain-ibm` (para Watsonx API) aunque en este caso usamos endpoint local; LlamaIndex ofrece un flujo end-to-end con índices persistentes y a veces menor "boilerplate".

En cualquiera de los casos, **el punto crítico es la correcta configuración del modelo de embeddings**, que fue la fuente de problemas en intentos previos. Los errores comunes incluyen: no usar el mismo modelo para indexar y para consultas, no alinear la dimensión del vector entre modelo y base de datos, o llamados incorrectos a la API. Para evitar esto:

- Use siempre `"ai/granite-embedding-multilingual"` tanto al indexar documentos como al embeddar las preguntas del usuario (no mezcle con otro modelo). Los embeddings Granite de 278M producen vectores de tamaño fijo 768, como confirman su tarjeta de modelo [37].
- Verifique que la columna en Postgres esté definida con `VECTOR(768)` y que pgvector esté habilitado (`CREATE EXTENSION pgvector;`). Si usa las clases de LangChain/LlamaIndex, estas típicamente crean la tabla con la columna `embedding vector(768)` automáticamente [38]. Si lo hace manual, puede crear una tabla por ejemplo: `CREATE TABLE mis_docs (id serial PRIMARY KEY, content text, embedding vector(768));`.
- Asegúrese de haber habilitado el acceso al endpoint local. Si al llamar desde Python obtiene errores de conexión, revise que Docker Model Runner esté activo y escuchando en `localhost:12434` (el comando `docker model list` muestra modelos cargados; también puede usar `docker model logs` para ver registros [39]). Recuerde que para **conexiones desde el host** es necesario el flag `--tcp 12434` mencionado [40] [26].
- En caso de dudas, puede probar primero fuera del framework: por ejemplo, ejecutar un embed con `openai.Embedding.create` y una consulta de chat con `openai.ChatCompletion.create` directamente, para confirmar que el modelo responde correctamente. Si eso funciona, integrarlo a LangChain/LlamaIndex ya es cuestión de pasar los parámetros correctos (API base, modelo, etc.) a las clases correspondientes.

Conclusiones y recomendaciones

Implementar un sistema RAG local con modelos Granite es factible y puede aprovechar completamente infraestructura open-source. IBM ha diseñado Granite 4.0 Micro para ser **ligero pero capaz en tareas de RAG**, lo cual se alinea con nuestro caso de uso [4]. De hecho, Docker enfatiza que combinando Granite con Model Runner se pueden construir *chatbots inteligentes con bases de conocimiento privadas* de forma eficiente [41]. La comunidad ya ha explorado casos similares: por ejemplo, usar **granite-embedding-278M junto a LLMs open-source** para chatbots legales o de documentación [42]. Estas experiencias sugieren que Granite ofrece una buena calidad relativa a su tamaño, especialmente al restringir el dominio (p.ej., responder preguntas tipo examen con materiales dados).

En cuanto a la elección **LangChain vs LlamaIndex**: ambos frameworks soportan la integración con Postgres/pgvector y permitirán lograr el objetivo. LangChain brinda módulos ya hechos para orquestar el RAG (p. ej. `PGVector` VectorStore y chains de QA) con mínima codificación manual [43]. LlamaIndex por su lado proporciona una interfaz de alto nivel para construir el índice y puede facilitar ciertas optimizaciones

(por ejemplo, filtrado por metadatos, re-rankers). Si ya comenzó con LlamaIndex pero tuvo problemas con la parte de embeddings, una recomendación es intentar primero con LangChain/OpenAI SDK para verificar el pipeline, dado que **LangChain con OpenAIEmbeddings** permite especificar directamente el `openai_api_base` al endpoint local (lo que encapsula la configuración) [23] [44] . Una vez validado, puede replicar la misma lógica en LlamaIndex asegurándose de configurar las variables de entorno o el `ServiceContext` con el embedding correcto.

En definitiva, utilizando los **modelos Granite locales vía Docker** y almacenando los vectores en **PostgreSQL (pgvector)**, es posible construir un sistema RAG completamente offline. Asegúrese de alinear correctamente cada componente (modelo de embedding, dimensión vectorial, consultas a la API local) y podrá indexar sus documentos y responder preguntas de forma efectiva. Granite 4.0 Micro, pese a su tamaño reducido, fue pensado precisamente para casos de uso de **RAG en entornos locales** con datos privados [45] [41] , por lo que resulta una elección adecuada. Con esta configuración, se logra un equilibrio entre privacidad (todos los datos y modelos residen localmente), costo (modelos open-source de libre uso) y rendimiento suficiente para un volumen moderado de documentos y consultas.

Fuentes y ejemplos relevantes:

- Documentación oficial de Docker Model Runner e IBM Granite (Docker Hub y blogs de Docker/IBM) sobre la disponibilidad de Granite 4.0 en Docker y uso vía API OpenAI [16] [46] .
- Tutoriales que combinan **LangChain + pgvector + granite-embedding** (ej. blog de Zilliz) mostrando configuración de embeddings de IBM Watsonx y almacenamiento en Postgres [42] [43] .
- Guía de Docker Model Runner para *embeddings* locales, incluyendo comandos `curl` y ejemplo en Python usando `openai` SDK para obtener embeddings con modelos en Docker [20] [21] .
- Referencia de IBM/Docker sobre Granite-Embedding-278M: modelo multilingüe 768-D ideal para búsqueda semántica en 12 idiomas [7] [9] .
- Posts técnicos sobre RAG con Postgres (pgvector) e integraciones en LlamaIndex vs LangChain, discutiendo mejores prácticas de almacenamiento y recuperación eficientes en entornos productivos [28] [32] .

---

[1] [29] [38] [42] [43] Build RAG Chatbot with LangChain, pgvector, OpenAI GPT-4o mini, and IBM granite-embedding-278m-multilingual
https://zilliz.com/tutorials/rag/langchain-and-pgvector-and-openai-gpt-4o-mini-and-ibm-granite-embedding-278m-multilingual

[2] [3] [5] [6] [16] [22] [41] [45] [46] IBM Granite 4.0 Models Now Available on Docker Hub | Docker
https://www.docker.com/blog/ibm-granite-4-0-models-now-available-on-docker-hub/

[4] [13] [14] ai/granite-4.0-h-micro - Docker Image
https://hub.docker.com/r/ai/granite-4.0-h-micro

[7] [8] [9] [10] [11] [12] [15] ai/granite-embedding-multilingual - Docker Image
https://hub.docker.com/r/ai/granite-embedding-multilingual

[17] [18] [19] [21] [23] [24] [25] [26] [39] [40] [44] Run AI Models Locally Using Docker Model runner | Medium
https://thelearningfellow.medium.com/run-ai-models-locally-with-ease-using-docker-model-runner-b7a3a43a32c8

[20] ai/embeddinggemma - Docker Image
https://hub.docker.com/r/ai/embeddinggemma

27 28 32 35 36 Production RAG with a Postgres Vector Store and Open-Source Models
https://christophergs.com/blog/production-rag-with-postgres-vector-store-open-source-models

30 31 OpenAIEmbeddings - Docs by LangChain
https://docs.langchain.com/oss/python/integrations/text_embedding/openai

33 34 Postgres Vector Store | LlamaIndex Python Documentation
https://developers.llamaindex.ai/python/examples/vector_stores/postgres/

37 IBM Granite Embedding 278m Multilingual - AWS Marketplace
https://aws.amazon.com/marketplace/pp/prodview-lw6sucdfy5ep6