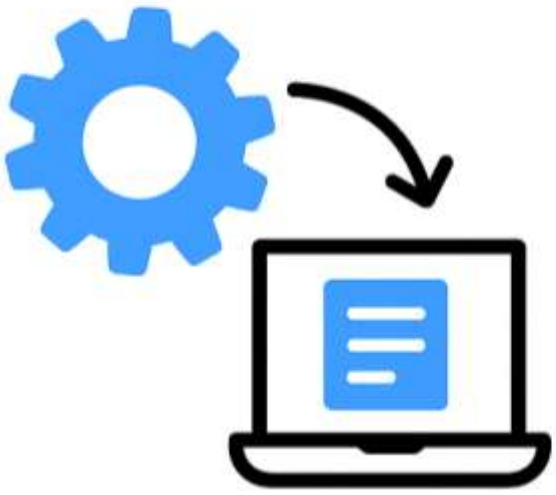


Retrieval-Augmented Generation RAG

Generación Aumentada con Recuperación

La guía completa de Retrieval Augmented Generation (RAG) en Azure Confidential Virtual Machines

Arquitectura de referencia: Implementar RAG confidencial en Azure con aceleración por hardware (Procesadores AMD EPYC de 4.ª generación, Intel® TDX en procesadores Intel® Xeon Scalable de 4.ª generación y GPU NVIDIA H100 Tensor Core)

1. "La guía completa de Retrieval Augmented Generation (RAG) en Azure Confidential Virtual Machines"

- **Retrieval Augmented Generation (RAG)** → Técnica de IA que combina modelos de lenguaje con búsqueda de información en bases de datos o documentos externos para dar respuestas más precisas y actualizadas.
- **Azure Confidential Virtual Machines (VMs confidenciales de Azure)** → Servidores virtuales en la nube de Microsoft que usan hardware y software diseñados para **proteger datos y código incluso mientras se están ejecutando**, gracias a entornos de ejecución seguros (*confidential computing*).

Traducción en contexto: Es una guía sobre cómo usar RAG dentro de máquinas virtuales seguras en Azure, protegiendo datos sensibles.

---

2. "Arquitectura de referencia"

- Significa que no es solo teoría, sino un diseño recomendado que describe **qué componentes usar, cómo se conectan y qué configuraciones son óptimas** para implementar RAG de manera segura en Azure.
- Es como un plano técnico listo para adaptar a tu proyecto.

---

3. "Implementar RAG confidencial en Azure con aceleración por hardware"

- "Confidencial" → se refiere a que los datos, el modelo y el código estarán protegidos mientras están en uso.
- "Aceleración por hardware" → usar procesadores y GPUs especiales que procesan más rápido y con mejor rendimiento las tareas de IA.

---

4. Tecnologías de hardware mencionadas

- **AMD 4th Gen EPYC processors** → procesadores de alto rendimiento de AMD, optimizados para cargas de trabajo de IA, big data y entornos de nube.
- **Intel® TDX (Trust Domain Extensions) en 4th Gen Intel® Xeon Scalable Processors** → tecnología de Intel para aislar y proteger entornos de ejecución, asegurando que incluso el sistema operativo anfitrión no pueda acceder a los datos o código en uso.

- **NVIDIA H100 Tensor Core GPUs** → tarjetas gráficas especializadas para IA y aprendizaje profundo, con alto rendimiento en entrenamiento e inferencia de modelos de gran tamaño.

Este título describe una guía que explica cómo montar un sistema de RAG dentro de entornos seguros de Azure, usando una arquitectura recomendada y hardware de última generación (procesadores AMD e Intel con funciones de seguridad avanzada y GPUs NVIDIA H100) para obtener máxima velocidad y protección de datos.

Introducción

Una de las brechas más críticas en los Modelos de Lenguaje Extensos (LLMs) tradicionales es que dependen de conocimiento estático ya contenido en ellos. Básicamente, pueden ser muy buenos para comprender y responder a instrucciones (prompts), pero a menudo fallan al proporcionar información más reciente o altamente específica.

Aquí es donde entra **Retrieval Augmentation Generation (RAG)**; RAG aborda estas brechas críticas en los LLM tradicionales incorporando información actual y nueva que sirve como una fuente confiable de verdad para estos modelos.

RAG mejora los modelos de IA generativa utilizando fuentes de conocimiento externas como documentos y bases de datos. Estas bases de conocimiento externas proporcionan información verificable y actualizada que puede mejorar los modelos de aprendizaje automático (ML) al reducir errores, simplificar la implementación y disminuir el costo de un reentrenamiento continuo. Para aprender más sobre RAG y entenderlo en mayor profundidad, puedes leer nuestro blog que cubre esta técnica.

Construir una infraestructura sólida de IA generativa, como las usadas para RAG, puede ser complejo y desafiante. Requiere una cuidadosa consideración de la pila tecnológica, los datos, la escalabilidad, la ética y la seguridad. En cuanto a la pila tecnológica, el hardware, los sistemas operativos, los servicios en la nube y los servicios de IA generativa deben ser resilientes y eficientes según la escala que las empresas requieran.

Dadas las preocupaciones que tienen las empresas sobre el costo, el bloqueo de proveedores, la soberanía de datos, el cumplimiento regulatorio y la complejidad de los proyectos de IA, el software de código abierto representa una poderosa forma de alcanzar los objetivos del proyecto mientras se mantiene el control total sobre los datos sensibles y se evitan restricciones y sobrecostos pesados. Hay varias opciones en el mercado, pero en este documento técnico nos centraremos en **Charmed OpenSearch** y **KServe**.

Al implementar un caso de uso de IA generativa y desarrollar una infraestructura robusta, es crucial priorizar la seguridad y la confidencialidad tanto de los datos como de los modelos de aprendizaje automático. Estos activos son invaluables, y protegerlos es esencial para mantener la confianza, garantizar el cumplimiento legal, apoyar la continuidad del negocio y prevenir ataques maliciosos. RAG puede ser particularmente vulnerable en este sentido, lo cual es especialmente preocupante para aplicaciones críticas en sectores como los servicios financieros y los servicios públicos.

Por supuesto, las empresas también tienen preocupaciones sobre los riesgos de realizar tareas de aprendizaje automático en la nube. Por ejemplo, pueden preocuparse por el posible compromiso de sus datos sensibles o de su propiedad intelectual. En algunos casos, existen retos regulatorios que considerar, como regulaciones estrictas de la industria que pueden prohibir el intercambio de datos sensibles. Esto dificulta, o incluso hace imposible, utilizar grandes cantidades de datos privados valiosos, lo que puede limitar fuertemente el verdadero potencial de la IA en dominios cruciales.

Confidential AI aborda directamente este problema, proporcionando un entorno de ejecución con raíz en hardware que abarca tanto la Unidad Central de Procesamiento (CPU) como la Unidad de Procesamiento Gráfico (GPU). Este entorno mejora la protección de los datos y el código de IA durante la ejecución, ayudando a resguardarlos contra software del sistema con privilegios (como el hipervisor o el sistema operativo anfitrión) y contra operadores privilegiados en la nube.

Objetivo

Esta guía se centra en la creación de un flujo de trabajo RAG utilizando herramientas de código abierto. Además, la guía enfatiza la importancia de la seguridad y la confidencialidad de los datos, y presenta innovaciones relacionadas con la seguridad, como la computación confidencial (*confidential computing*), que pueden ayudar a proteger tu implementación de RAG.

Está dirigida a entusiastas de los datos, ingenieros, científicos y profesionales del aprendizaje automático que deseen desarrollar soluciones RAG en plataformas de nube pública como Azure, utilizando herramientas empresariales de código abierto que no forman parte de la oferta nativa de microservicios de Azure.

Esta guía puede apoyar diversos proyectos, incluidos **pruebas de concepto**, **desarrollo** y **producción**.

Ten en cuenta que, aunque esta guía resalta herramientas de código abierto específicas, también pueden utilizarse otras herramientas no mencionadas como alternativas. Si decides usar herramientas diferentes, asegúrate de ajustar las especificaciones de hardware —como la capacidad de almacenamiento, la potencia de cómputo y la configuración— para cumplir con los requisitos específicos de tu caso de uso.

Solución de referencia RAG

Cuando se realiza una consulta en un chatbot de IA, el sistema basado en RAG primero recupera información relevante de un conjunto de datos o base de conocimiento, y luego utiliza esta información para guiar y orientar la generación de la respuesta.

El sistema RAG está compuesto por dos componentes clave:

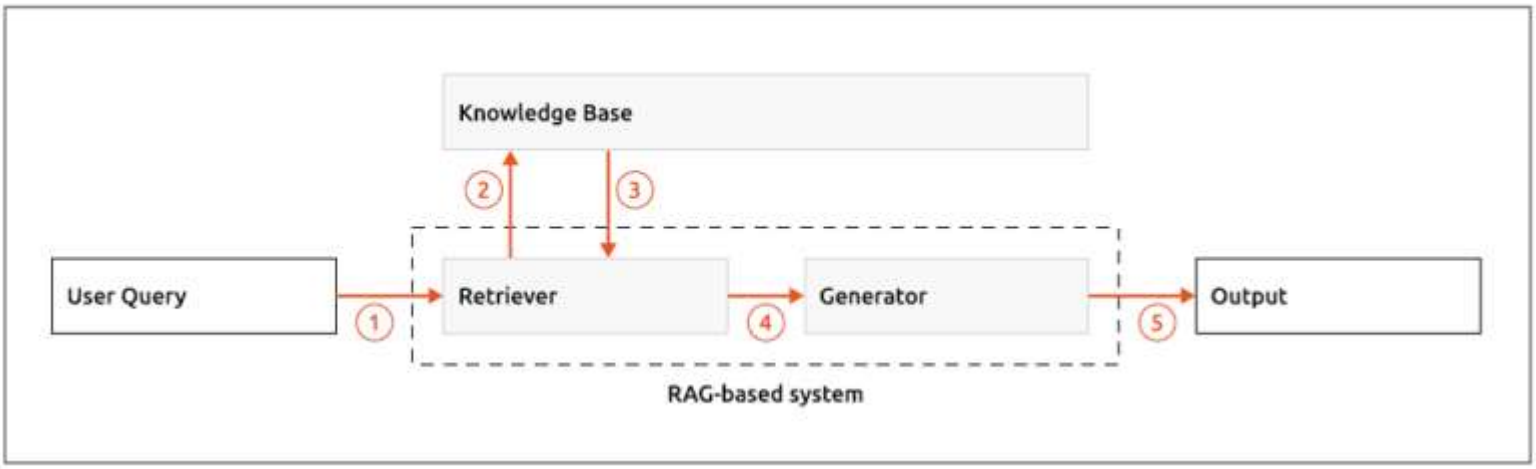
1. **Retriever (Recuperador)** → Es responsable de localizar fragmentos de información relevantes que puedan ayudar a responder la consulta del usuario. Busca en una base de datos para seleccionar la información más pertinente.
2. **Generator (Generador)** → Es un modelo de lenguaje grande (*Large Language Model*) que produce la respuesta final.

Antes de usar tu sistema basado en RAG, primero debes crear tu **base de conocimiento**, que está formada por datos externos que no están incluidos en los datos de entrenamiento de tu LLM. Estos datos externos pueden provenir de diversas fuentes, incluyendo documentos, bases de datos y llamadas a API.

La mayoría de los sistemas RAG utilizan una técnica de IA llamada **model embedding** (modelo de incrustaciones), que convierte los datos en representaciones numéricas y los almacena en una **base de datos vectorial**.

Al usar un modelo de embeddings, puedes crear un modelo de conocimiento que sea fácil de entender y rápido de recuperar en el contexto de la IA.

Una vez que tienes tu base de conocimiento y tu base de datos vectorial configuradas, ya puedes ejecutar tu proceso RAG; a continuación se presenta un flujo conceptual:



Este flujo conceptual sigue 5 pasos generales:

1. **El usuario introduce un *prompt* o consulta.**
2. **El *Retriever* (Recuperador)** busca información relevante en una base de conocimiento. La relevancia puede determinarse mediante cálculos matemáticos vectoriales y representaciones a través de funciones de búsqueda y bases de datos vectoriales.
3. **Se recupera la información relevante** para proporcionar un contexto enriquecido al generador.
4. **La consulta y el *prompt*** ahora se enriquecen con este contexto y están listos para ser aumentados para su uso con un modelo de lenguaje grande, utilizando técnicas de *prompt engineering* (ingeniería de *prompts*). El *prompt* aumentado permite que el modelo de lenguaje responda con precisión a tu consulta.
5. **Finalmente, se entrega la respuesta generada en texto** al usuario.

Solución de referencia avanzada de RAG y GenAI con código abierto

RAG puede utilizarse en diversas aplicaciones, como chatbots de IA, búsqueda semántica, resumen de datos e incluso generación de código. La solución de referencia que se presenta a continuación describe cómo RAG puede combinarse con arquitecturas de referencia avanzadas de IA generativa para crear proyectos de LLM optimizados que ofrezcan soluciones contextuales a diversos casos de uso de GenAI.

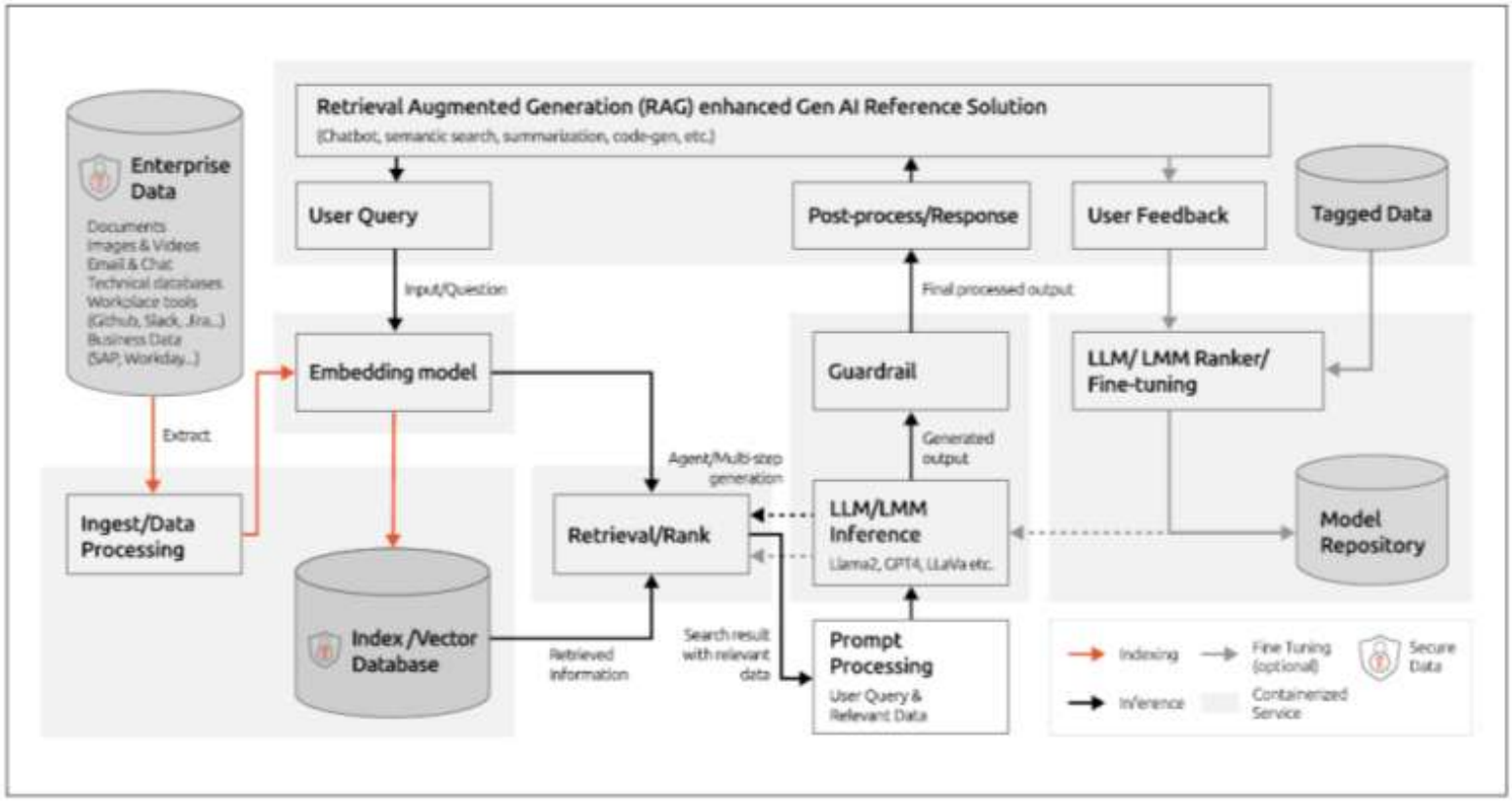


Figure 1: RAG enhanced GenAI Ref solution (source: https://opea.dev/)

Explicación del diagrama “RAG enhanced GenAI Reference Solution”

Este diagrama describe cómo funciona un sistema de **Retrieval-Augmented Generation (RAG)**, que combina modelos de lenguaje grandes (**LLM**) con bases de datos de búsqueda para dar respuestas más precisas, usando información interna de la empresa.

El *blueprint* de GenAI mencionado anteriormente fue publicado por **OPEA** (*Open Platform for Enterprise AI*), un proyecto de la **Linux Foundation**. El objetivo de crear este *blueprint* es establecer un marco de bloques de construcción componibles para sistemas de IA generativa de última generación, que incluyen LLM, almacenamiento de datos y motores de *prompts*.

Además, proporciona un *blueprint* para RAG y describe flujos de trabajo de extremo a extremo. La versión 1.1 recientemente lanzada del proyecto OPEA presentó múltiples proyectos de GenAI que demuestran cómo los sistemas RAG pueden mejorarse mediante herramientas de código abierto.

Cada servicio dentro de los bloques tiene tareas específicas que realizar, y existen diversas soluciones de código abierto disponibles que pueden ayudar a **acelerar estos servicios según las necesidades de la empresa**.

---

1. Enterprise Data (Datos de la Empresa)

- Aquí está la información interna: documentos, imágenes, videos, correos, chats, bases técnicas, datos de herramientas de trabajo, y sistemas de negocio (como SAP o Workday).
- **Función:** Fuente de conocimiento para el sistema.

---

2. Ingest/Data Processing (Ingesta/Procesamiento de Datos)

- Extrae y limpia los datos para que puedan usarse en el sistema.
- **Función:** Convertir la información bruta en un formato estructurado.

---

3. Embedding Model (Modelo de Embeddings)

- Transforma los datos y las consultas en vectores numéricos que el sistema pueda comparar.
- **Función:** Representar el significado del texto en un espacio matemático.

---

4. Index/Vector Database (Base de Datos de Vectores)

- Guarda los embeddings de todos los documentos procesados.
- **Función:** Permitir búsquedas rápidas de información relevante.

---

5. User Query (Consulta del Usuario)

- Pregunta o instrucción que el usuario introduce en el sistema.
- **Función:** Punto de partida de la interacción.

---

6. Retrieval/Rank (Recuperación/Clasificación)

- Busca en la base de datos de vectores la información más relevante y la ordena por relevancia.
- **Función:** Filtrar los datos más útiles para la respuesta.

---

7. Prompt Processing (Procesamiento del Prompt)

- Combina la consulta del usuario y la información recuperada en una sola entrada para el modelo de lenguaje.
- Función: Preparar la entrada que recibirá el LLM.

---

8. LLM/LMM Inference (Inferencia de LLM/LMM)
- El modelo (por ejemplo, LLaMA2, GPT-4, LLaVA) genera una respuesta usando la información del prompt.
- **Función:** Crear contenido o responder preguntas usando IA.

---

9. Guardrail (Barandilla de Seguridad)
- Filtros y reglas para asegurar que la respuesta sea segura, precisa y cumpla con las normas.
- **Función:** Evitar respuestas dañinas, sesgadas o erróneas.

---

10. Post-process/Response (Post-procesado/Respuesta)
- Ajusta el formato final de la respuesta para presentarla al usuario.
- **Función:** Mejorar legibilidad y coherencia.

---

11. User Feedback (Retroalimentación del Usuario)
- El usuario califica o corrige la respuesta.
- **Función:** Permitir mejoras continuas.

---

12. Tagged Data (Datos Etiquetados)
- Respuestas y datos evaluados que sirven para reentrenar o ajustar el modelo.
- **Función:** Fuente para mejorar precisión.

---

13. LLM/LMM Ranker/Fine-tuning (Clasificador/Entrenamiento Fino de LLM/LMM)
- Usa la retroalimentación para ajustar o refinar el modelo.
- **Función:** Optimizar la calidad de las respuestas con aprendizaje supervisado.

---

14. Model Repository (Repositorio de Modelos)
- Almacena las versiones entrenadas y afinadas de los modelos.
- **Función:** Tener acceso a los modelos actualizados.

Leyenda de colores y flechas:
- **Flechas rojas:** Indexación de datos.
- **Flechas negras:** Flujo de inferencia (consulta → respuesta).
- **Flechas grises:** Proceso opcional de ajuste fino (fine-tuning).
- **Icono de escudo:** Datos seguros.

Solución de referencia con Charmed OpenSearch y KServe

Al construir un proyecto de IA generativa, como un sistema RAG y arquitecturas de referencia avanzadas de IA generativa, es fundamental incluir múltiples componentes y servicios. Estos componentes normalmente abarcan bases de datos, bases de conocimiento, sistemas de recuperación, bases de datos vectoriales, modelos de *embeddings*, modelos de lenguaje grandes (LLMs), motores de inferencia, procesamiento de *prompts*, y servicios de *guardrail* y *fine-tuning*, entre otros.

RAG permite a los usuarios elegir los servicios y aplicaciones de RAG más adecuados para sus casos de uso específicos. El flujo de trabajo de referencia descrito a continuación utiliza principalmente dos herramientas de código abierto: **Charmed OpenSearch** y **KServe**.

En el flujo de trabajo RAG que se muestra a continuación, el *fine-tuning* no es obligatorio; sin embargo, puede mejorar el rendimiento de los LLM a medida que el proyecto escala.

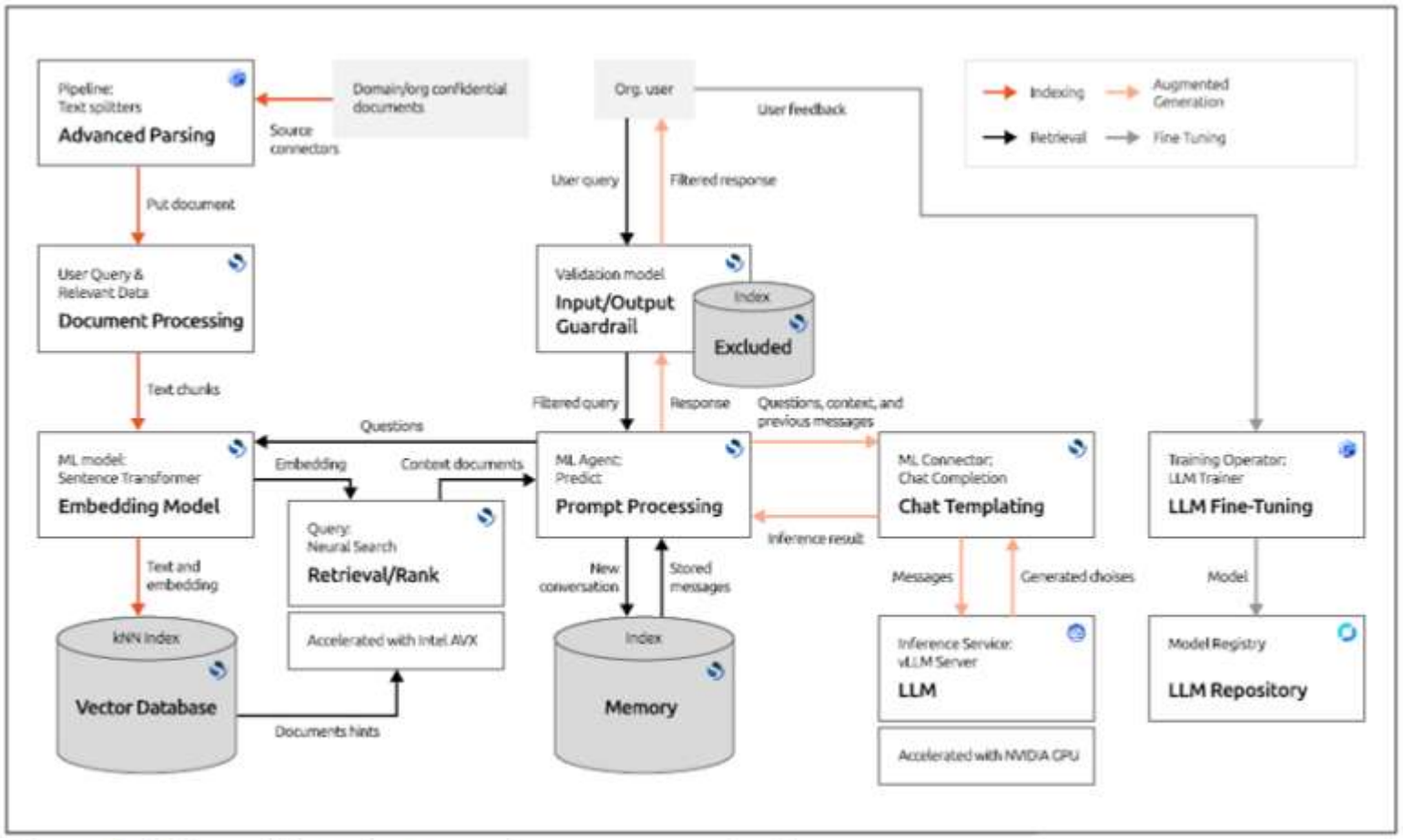


Figure 2: RAG workflow diagram using open source tools

Explicación del flujo RAG del diagrama

El diagrama describe un **sistema RAG (Retrieval-Augmented Generation)** usando diferentes componentes y cómo interactúan entre sí para procesar consultas de usuario, buscar información relevante y generar respuestas optimizadas.

1.- Ingreso de documentos

- **Advanced Parsing** *(Análisis Avanzado)*
  Aquí los documentos de origen (corporativos, confidenciales, externos) se dividen en partes manejables usando *text splitters* y conectores de datos.
  Salida: Documentos procesados listos para indexar.
- **Document Processing** *(Procesamiento de Documentos)*
  Procesa el contenido del documento y lo prepara para ser convertido en *embeddings*.
  Entrada: Documentos o consultas de usuario.
  Salida: Texto limpio y estructurado.

2.- Generación de Embeddings y búsqueda

- **Embedding Model** *(Modelo de Embeddings)*
  Convierte el texto en representaciones vectoriales (embeddings) usando modelos tipo *Sentence Transformer*.
  Salida: Embeddings del texto.
- **Vector Database** *(Base de Datos Vectorial)*
  Almacena embeddings para búsquedas rápidas mediante índices *kNN*.
  Entrada: Embeddings y documentos.
  Salida: Documentos relevantes según similitud.
- **Retrieval/Rank** *(Recuperación y Clasificación)*
  Realiza búsqueda neuronal (*Neural Search*) y clasifica resultados relevantes, acelerado por Intel AVX.
  Salida: Documentos más relevantes para la consulta.

3.- Gestión de contexto y control

- **Memory** *(Memoria)*
  Guarda el historial de conversaciones, contexto previo y mensajes anteriores para que el LLM tenga coherencia en sus respuestas.
- **Input/Output Guardrail** *(Control de Entrada/Salida)*
  Filtra y valida la información antes de enviarla al modelo, aplicando reglas de seguridad y políticas de la organización.
- **Excluded** *(Excluidos)*
  Almacena contenido filtrado que no debe usarse por temas de seguridad o confidencialidad.

4.- Procesamiento y generación de respuesta

- **Prompt Processing** *(Procesamiento de Prompts)*
  Toma la consulta del usuario, documentos relevantes y contexto previo, y construye el *prompt* final para el LLM.
- **Chat Templating** *(Plantillas de Chat)*
  Usa plantillas predefinidas para estructurar la conversación y mejorar la consistencia de las respuestas.
- **LLM** *(Modelo de Lenguaje Grande)*
  Motor principal de generación de texto, acelerado por GPU NVIDIA. Recibe el *prompt* final y genera la respuesta.

5.- Ajuste y mejoras del modelo

- **LLM Fine-Tuning** *(Ajuste Fino del LLM)*
  (Opcional) Entrenamiento adicional del modelo para mejorar su desempeño en dominios específicos.
- **LLM Repository** *(Repositorio de LLM)*
  Almacena las versiones y modelos entrenados para su uso en producción.

Flujos en el diagrama (colores de flechas)

- **Indexing** *(Indexación)* → Procesar e insertar datos en la base vectorial.
- **Retrieval** *(Recuperación)* → Buscar información relevante para la consulta.
- **Augmented Generation** *(Generación Aumentada)* → Producir respuesta usando contexto recuperado.
- **Fine Tuning** *(Ajuste fino)* → Entrenamiento adicional del LLM.

La tabla a continuación describe todos los servicios RAG resaltados en el diagrama de flujo anterior y asigna las soluciones de código abierto que se utilizan en esta guía.

Servicio	Descripción	Soluciones de código abierto
Advanced parsing (Análisis avanzado)	*Text splitters* son técnicas avanzadas de análisis para el documento que va al sistema RAG. De esta forma, el documento puede estar más limpio, enfocado y proveer una entrada más informativa.	Charmed Kubeflow: *Text splitters*
Ingest/data processing (Ingesta/procesamiento de datos)	La ingesta o procesamiento de datos es una capa de canalización de datos. Es responsable de la extracción, limpieza y eliminación de datos innecesarios que se van a procesar.	Charmed OpenSearch para procesamiento de documentos
Embedding model (Modelo de incrustación)	Es un modelo de *machine learning* que convierte datos en bruto en representaciones vectoriales.	Charmed OpenSearch – *Sentence transformer*
Retrieval and ranking (Recuperación y clasificación)	Recupera los datos de la base de conocimiento y clasifica la relevancia de la información obtenida en función de puntajes de relevancia.	Charmed OpenSearch con FAISS (*Facebook AI Similarity Search*)
Vector database (Base de datos vectorial)	Almacena incrustaciones vectoriales para que los datos puedan ser fácilmente buscados por los servicios de "recuperación y clasificación".	Charmed OpenSearch – *KNN Index* como base de datos vectorial
Prompt processing (Procesamiento de *prompts*)	Da formato a las consultas y textos recuperados en un formato legible y estructurado para el LLM.	Charmed OpenSearch – *OpenSearch: ML - agent predict*
LLM (Modelo de Lenguaje Extenso)	Proporciona la respuesta final utilizando múltiples modelos de IA generativa.	GPT, Llama, Deepseek
LLM inference (Inferencia de LLM)	Operacionaliza el *machine learning* en producción procesando datos en tiempo de ejecución en un modelo, para que este genere una salida.	Charmed Kubeflow con KServe
Guardrail (Barandilla o filtro de seguridad)	Garantiza contenido ético en la respuesta de la IA generativa creando un filtro para entradas y salidas.	Charmed OpenSearch: modelo de validación *guardrail*
LLM Fine-tuning (Ajuste fino de LLM)	Proceso de tomar un modelo preentrenado y seguir entrenándolo con un conjunto de datos más pequeño y específico.	Charmed Kubeflow
Model repository (Repositorio de modelos)	Se usa para almacenar y versionar modelos de *machine learning* entrenados, especialmente en el proceso de ajuste fino. Puede rastrear el ciclo de vida del modelo desde su despliegue hasta su retiro.	Charmed Kubeflow, Charmed MLFlow
Framework for building LLM application (Marco para construir aplicaciones con LLM)	Simplifica el flujo de trabajo, *prompts* y servicios de LLM para que sea más fácil construirlos.	LangChain

Esta tabla ofrece una visión general de los componentes clave involucrados en la construcción de un sistema RAG y una solución de referencia avanzada de GenAI, junto con las soluciones de código abierto asociadas a cada servicio.
Cada servicio realiza una tarea específica que puede mejorar la configuración de tu LLM, ya sea en lo relacionado con la gestión y preparación de datos, la incorporación (*embedding*) de un modelo en tu base de datos o la mejora del propio LLM.
La guía de implementación que se presenta a continuación cubrirá la mayoría de los servicios, excepto los siguientes: repositorio de modelos, ajuste fino de LLM (*fine-tuning*) y *text splitters*.

La velocidad de innovación en este campo, especialmente dentro de la comunidad de código abierto, se ha vuelto exponencial. Es crucial mantenerse actualizado con los últimos avances, incluidos nuevos modelos y las soluciones RAG emergentes.

Componente RAG: Charmed OpenSearch

Charmed OpenSearch se utilizará principalmente en este despliegue del flujo de trabajo RAG. Charmed OpenSearch es un *operator* que se basa en la versión *upstream* de OpenSearch, integrando automatización para optimizar la implementación, gestión y orquestación de clústeres en producción.

El *operator* mejora la eficiencia, la consistencia y la seguridad.

Entre sus funciones destacadas se incluyen: alta disponibilidad, escalado fluido para despliegues de cualquier tamaño, cifrado HTTP y de datos en tránsito, compatibilidad multicloud, actualizaciones seguras sin tiempo de inactividad, gestión de roles y complementos, y visualización de datos a través de **Charmed OpenSearch Dashboards**.

Con el *operator* Charmed OpenSearch (también conocido como *charm*), puedes implementar y ejecutar OpenSearch en máquinas físicas y virtuales (VM) y en otros entornos de nube o similares a la nube, incluidos AWS, Azure, Google Cloud, OpenStack y VMware.
Para la siguiente sección, la guía de implementación, se utilizarán instancias de máquinas virtuales de **Azure**.

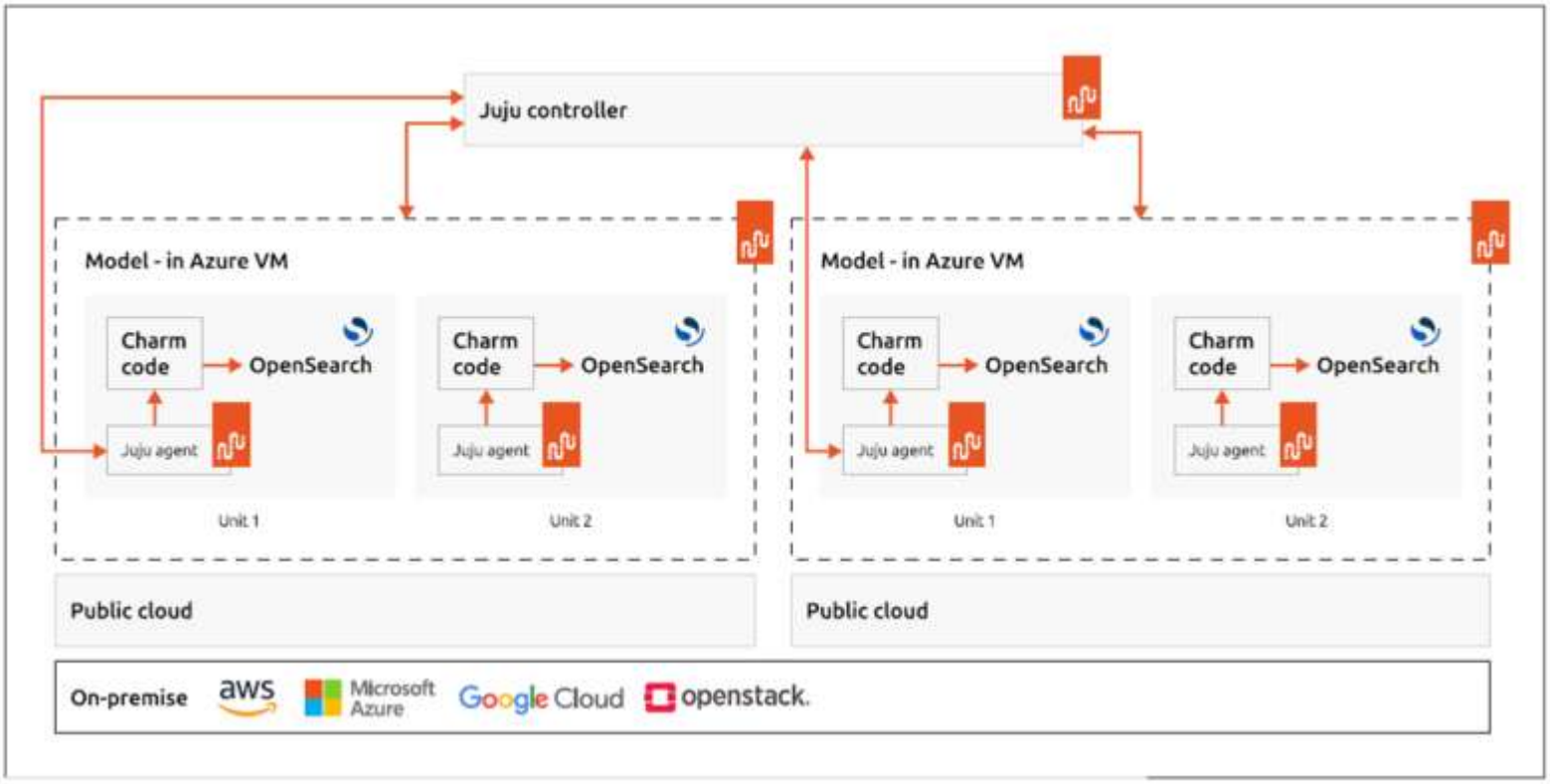


Figure 3: Charmed OpenSearch architecture

Descripción general del diagrama

La imagen representa cómo se despliega **Charmed OpenSearch** usando **Juju** en máquinas virtuales de Azure (Azure VM), dentro de un flujo RAG. También muestra que la misma arquitectura puede funcionar tanto **on-premise** (en servidores propios) como en varias nubes públicas: AWS, Microsoft Azure, Google Cloud y OpenStack.

Componentes y flujo

1. **Juju Controller** *(Controlador Juju)*
   - Es el cerebro que administra el despliegue y la orquestación.
   - Se encarga de coordinar los modelos, unidades y *charms* en diferentes entornos.

2. **Model – in Azure VM** *(Modelo – en máquina virtual de Azure)*
   - Cada "modelo" es una agrupación lógica de servicios gestionados por Juju.
   - Dentro del modelo hay **unidades** (*Units*), que son instancias concretas del servicio.

3. **Unit 1 y Unit 2** *(Unidad 1 y Unidad 2)*
   - Cada unidad representa una instancia de **OpenSearch** desplegada en una VM.
   - Estas unidades incluyen:
     - **Charm code** *(Código del charm)* → El conjunto de scripts y lógica que automatiza la instalación, configuración y gestión de OpenSearch.
     - **OpenSearch** → El motor de búsqueda y análisis.
     - **Juju agent** *(Agente Juju)* → Software que ejecuta instrucciones del controlador y aplica cambios en la unidad.

4. **Charm code → OpenSearch** *(Código del charm → OpenSearch)*
   - El charm instala y configura OpenSearch automáticamente.
   - Se encarga de tareas como escalado, seguridad y actualizaciones.

5. **Public Cloud** *(Nube pública)* y **On-premise** *(Infraestructura local)*
   - La misma arquitectura se puede desplegar:
     - En la nube pública (AWS, Azure, Google Cloud, OpenStack).
     - En servidores propios (on-premise).
     - Esto da flexibilidad para entornos híbridos o multicloud.

En resumen:

El diagrama muestra cómo Juju Controller gestiona varios modelos de OpenSearch, cada uno compuesto por múltiples unidades distribuidas en Azure VM, pero con la capacidad de operar en otras nubes o en infraestructura local. Cada unidad ejecuta un charm que instala y configura OpenSearch, y es controlada por un Juju agent que recibe órdenes del controlador.

Componente RAG: KServe

KServe es una solución *cloud-native* (nativa de la nube) dentro del ecosistema Kubeflow que sirve modelos de aprendizaje automático (*machine learning*). Al aprovechar Kubernetes, KServe opera de manera eficiente en entornos nativos de la nube.

Puede utilizarse para diversos fines, incluyendo:

- Despliegue de modelos
- Control de versiones de modelos de *machine learning*
- Inferencia de modelos LLM (*Large Language Models*)
- Monitoreo de modelos

En el caso de uso RAG abordado en esta guía, utilizaremos KServe para realizar inferencia sobre un LLM. Específicamente, procesará un LLM previamente entrenado para hacer predicciones basadas en nuevos datos.

Esto resalta la necesidad de contar con un sistema de inferencia de LLM robusto, capaz de trabajar con LLM locales y públicos.
El sistema debe ser escalable, manejar alta concurrencia, ofrecer respuestas de baja latencia y proporcionar respuestas precisas a consultas relacionadas con LLM.
En la guía de despliegue, te guiaremos paso a paso en la construcción y despliegue de un servicio RAG utilizando Charmed OpenSearch y KServe. Charmed Kubeflow de Canonical es compatible de forma nativa con KServe.

RAG Security & Confidentiality – Confidential AI con Confidential Computing

1. Contexto

- En entornos tradicionales de nube pública, el código y los datos corren el riesgo de ser expuestos debido a:
- Vulnerabilidades en el sistema operativo, hipervisor o firmware.
- Acceso no autorizado por parte de administradores maliciosos.
- Ataques físicos a la memoria del sistema.

Confidential Computing rompe con este modelo, protegiendo el código y los datos *mientras están en uso*, y ahora también extiende esa protección a **GPU** gracias a la tecnología de **NVIDIA H100**.

---

2. Arquitectura de Confidential AI en Azure

La solución combina **CPU-TEE** y **GPU-TEE**, asegurando datos:

- **En uso**
- **En tránsito**
- **En reposo**

A. CPU-TEE (Intel® TDX y AMD SEV-SNP)

Ubuntu Confidential VMs en Azure pueden ejecutarse sobre:

1. **Intel® TDX – 4th Gen Xeon**
   - **Aislamiento de memoria** con cifrado AES-128 en el controlador de memoria.
   - **Control de acceso por hardware** con nuevas instrucciones para auditar tareas críticas.
   - **Atestación remota** para verificar que el entorno es confiable antes de procesar datos.
2. **AMD SEV-SNP – 4th Gen EPYC**
   - **Confidencialidad en tiempo de ejecución**: cifrado AES-128 en DRAM.
   - **Integridad en tiempo de ejecución**: protección contra corrupción de datos y ataques de replay.
   - **Protección total del espacio de direcciones** mediante claves gestionadas por hardware.

B. GPU-TEE (NVIDIA H100 Tensor Core)

1.- Asegura confidencialidad e integridad de la computación dentro de la GPU.
2.- Extiende la protección de **Confidential Computing** desde CPU a GPU.
3.- Comunicación PCIe cifrada entre VM y GPU, con integridad garantizada.

C. Mecanismos clave de seguridad

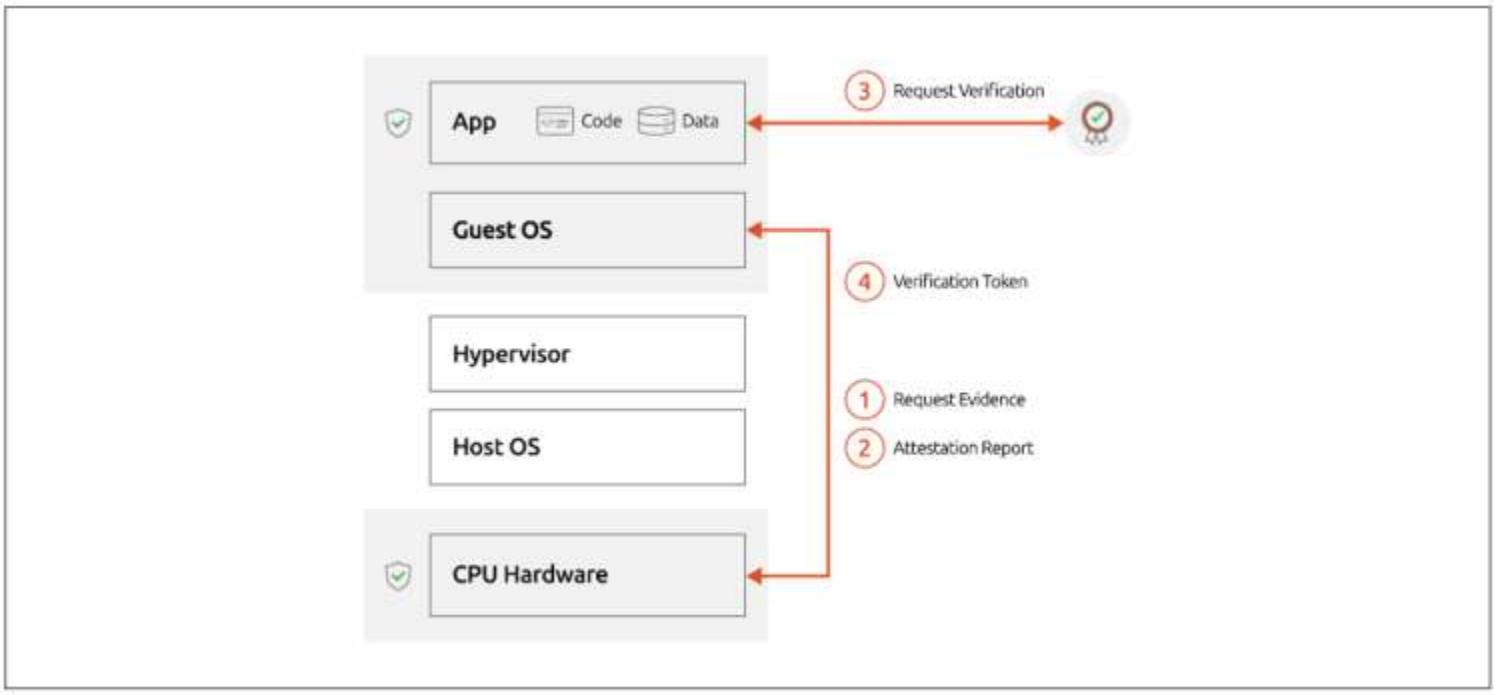
1.- Cifrado de memoria (CPU y GPU) – AES-128 gestionado por hardware.
2.- Protección contra accesos no autorizados – Incluso por administradores del cloud.
3.- Integridad de datos – Prevención de corrupción y ataques de repetición.
4.- Atestación criptográfica – Verificación remota de CPU y GPU antes de usar datos.

---

3. Aplicación en RAG

Para **RAG seguro**, esto significa que:

.- Los documentos en la base de conocimiento y las consultas se procesan en entornos cifrados.
.- Ni el proveedor cloud ni terceros pueden acceder al contenido.
.- Todo el flujo — indexación, búsqueda y generación — ocurre dentro de un TEE (Trusted Execution Environment).



Descripción del diagrama

1. **Request Evidence** (*Solicitar Evidencia*)
   - El **Host OS** (Sistema Operativo del Host) solicita al **CPU Hardware** evidencia criptográfica sobre el estado y la integridad del entorno seguro.
   - Esta evidencia contiene mediciones del hardware y del software que se está ejecutando en el enclave o entorno seguro.
2. **Attestation Report** (*Informe de Atestación*)
   - El **CPU Hardware** devuelve un **Attestation Report** al **Host OS**.
   - Este informe incluye firmas criptográficas que prueban que el hardware y el software son auténticos y no han sido manipulados.
3. **Request Verification** (*Solicitar Verificación*)
   - La **App** (Aplicación) que está en el **Guest OS** (Sistema Operativo Invitado) envía el informe de atestación a un **Verificador externo** para que valide su autenticidad.
   - Esto se hace para asegurar que un tercero confiable confirme que todo el entorno es seguro.
4. **Verification Token** (*Token de Verificación*)
   - El verificador externo analiza el **Attestation Report** y, si todo es válido, devuelve un **Verification Token**.
   - Este token confirma que la aplicación, el código y los datos se ejecutan en un entorno confiable y seguro.

Resumen visual

- **CPU Hardware** → Garantiza seguridad física y genera pruebas criptográficas.
- **Host OS / Hypervisor** → Administra el hardware y coordina la solicitud de evidencias.
- **Guest OS** → Donde se ejecuta la aplicación y maneja los datos dentro de un entorno aislado.
- **App** → Recibe la confirmación de que todo está protegido antes de procesar datos sensibles.

Conclusión – Confidential AI en RAG

Al **integrar CPU-TEE y GPU-TEE** en una solución cohesiva —como las implementaciones de **RAG (Retrieval-Augmented Generation)**— la **IA confidencial** deja de ser un concepto teórico y se convierte en una opción **viable y práctica**.

Esto permite que las organizaciones:

- Aprovechen el poder de la IA generativa y de búsqueda aumentada.
- Mantengan **los más altos estándares de seguridad y confidencialidad** de datos.
- Protejan tanto **datos en uso** como **procesos de inferencia** frente a amenazas internas y externas.

Además, la **Confidential AI** puede potenciarse aún más con **primitivas criptográficas** como:

- **Privacidad diferencial (Differential Privacy)** – para prevenir filtraciones indirectas de datos sensibles a través de patrones de salida.
- **Cifrado homomórfico** – para procesar datos cifrados sin necesidad de descifrarlos.
- **Firmas digitales y verificaciones de integridad** – para garantizar la autenticidad de modelos y resultados.

En conjunto, esto crea un ecosistema en el que **la inteligencia artificial trabaja de forma potente, escalable y segura**, incluso en entornos de nube pública, sin comprometer la confidencialidad.

Máquinas Virtuales Confidenciales en Azure

Microsoft Azure ha anunciado la disponibilidad general de sus **máquinas virtuales confidenciales** con **GPU NVIDIA H100 Tensor Core**, impulsadas por **Ubuntu**.

Esta oferta combina la protección basada en hardware de los procesadores AMD EPYC con la última tecnología de GPU de NVIDIA para habilitar cargas de trabajo de IA seguras y de alto rendimiento en la nube.

De manera similar, Azure también anunció que Intel® TDX en procesadores Intel® Xeon Scalable de 4.ª generación está disponible en vista previa pública.
La combinación de estas tecnologías permite que sectores sensibles adopten la IA, abordando preocupaciones previas sobre la privacidad crítica de los datos.

Objetivo:

Desplegar el flujo de trabajo RAG en máquinas virtuales confidenciales de Microsoft® Azure Cloud

El objetivo es implementar el flujo de trabajo **RAG** (descrito anteriormente) en las **máquinas virtuales confidenciales** de Microsoft® Azure Cloud.

Las instancias recomendadas de Azure que se enumeran a continuación representan los **requisitos mínimos** para ejecutar **tres nodos de OpenSearch** y **un nodo de KServe**.

Azure es una de las nubes públicas más populares.
Al usar Azure, los usuarios y desarrolladores de RAG pueden beneficiarse de desplegar sistemas **RAG escalables, de alto rendimiento y seguros**, capaces de manejar tareas complejas de **NLP (Procesamiento de Lenguaje Natural)**.

Azure ofrece:

.- Aislamiento de red para flujos de trabajo RAG.
.- Seguridad de datos mediante cifrado de disco y almacenamiento.
.- Procesamiento de datos de alto rendimiento con gran ancho de banda de red.
.- Funciones nativas de monitoreo y registro (logging)**.**
.- Sistemas de cumplimiento normativo que incluyen diversas certificaciones.

Aplicación	Instancias de Azure
Charmed OpenSearch	**GPU NVIDIA H100 Tensor Core**
	NCCads_H100_v5: https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/nccadsh100v5-series?tabs=sizebasic
	Procesadores AMD EPYC de 3.ª generación
	DCasv5: https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/dcasv5-series?tabs=sizebasic
	ECasv5: https://learn.microsoft.com/en-us/azure/virtual-machines/ecasv5-ecadsv5-series
	ECadsv5: https://learn.microsoft.com/en-us/azure/virtual-machines/ecasv5-ecadsv5-series
	Intel® TDX en procesadores Xeon Scalable de 4.ª generación
	Serie DCesv5: https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/dcesv6-series?tabs=sizebasic
	Series ECesv5 y ECedsv5: https://learn.microsoft.com/en-us/azure/virtual-machines/ecesv5-ecedsv5-series
KServe	**GPU NVIDIA H100 Tensor Core**
	Serie NCCads_H100_v5: https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/nccadsh100v5-series?tabs=sizebasic

Guía de implementación

En esta guía, te guiaremos paso a paso para configurar un **pipeline RAG**.

Usaremos **Charmed OpenSearch** para la recuperación eficiente de búsquedas y **KServe** para la inferencia de *machine learning*.

Para seguir y completar esta guía, necesitarás:

- Una suscripción activa a **Azure**, con los permisos necesarios para crear máquinas virtuales (VMs), redes y realizar el registro de aplicaciones en **Entra ID**.
- Cualquier máquina con **Ubuntu** que pueda usar **Juju CLI** y operar la implementación posteriormente. Esto puede ser una VM, un portátil o cualquier otro equipo.
- Conocimientos del ecosistema **Juju**, **OpenSearch** y **KServe**.

En el momento de redactar este tutorial, la última versión estable de **Juju** es **3.5/stable**, y será la que utilicemos para el *bootstrap* de OpenSearch.

Prerrequisitos

1. Instalar Juju y configurar las credenciales de Azure

bash

```
$ sudo snap install juju
```

2. Registrar la aplicación CLI en Microsoft Entra ID

Esto permitirá que **Juju CLI** pueda crear, modificar y eliminar recursos en Azure (usando Azure CLI).

bash Copiar Editar

```
$ az ad app create --display-name jujucli
$ appid=$(az ad app list --display-name jujucli --query "[].{id: appId}" -o tsv)
$ az ad app credential list --id $appid --query keyId -o tsv
$ az ad app credential reset --id $appid
```

Ejemplo de salida:

```json
{
  "appId": "xxxxxx",
  "password": "qqqqq",
  "tenant": "zzzzz"
}
```

3. Crear el *service principal* (Azure CLI)

```bash
$ az ad sp create --id $appid
$ spObjectId=$(az ad sp show --id $appid --query id -o tsv)
```

4. Conceder a esta aplicación el rol de Propietario en un recurso

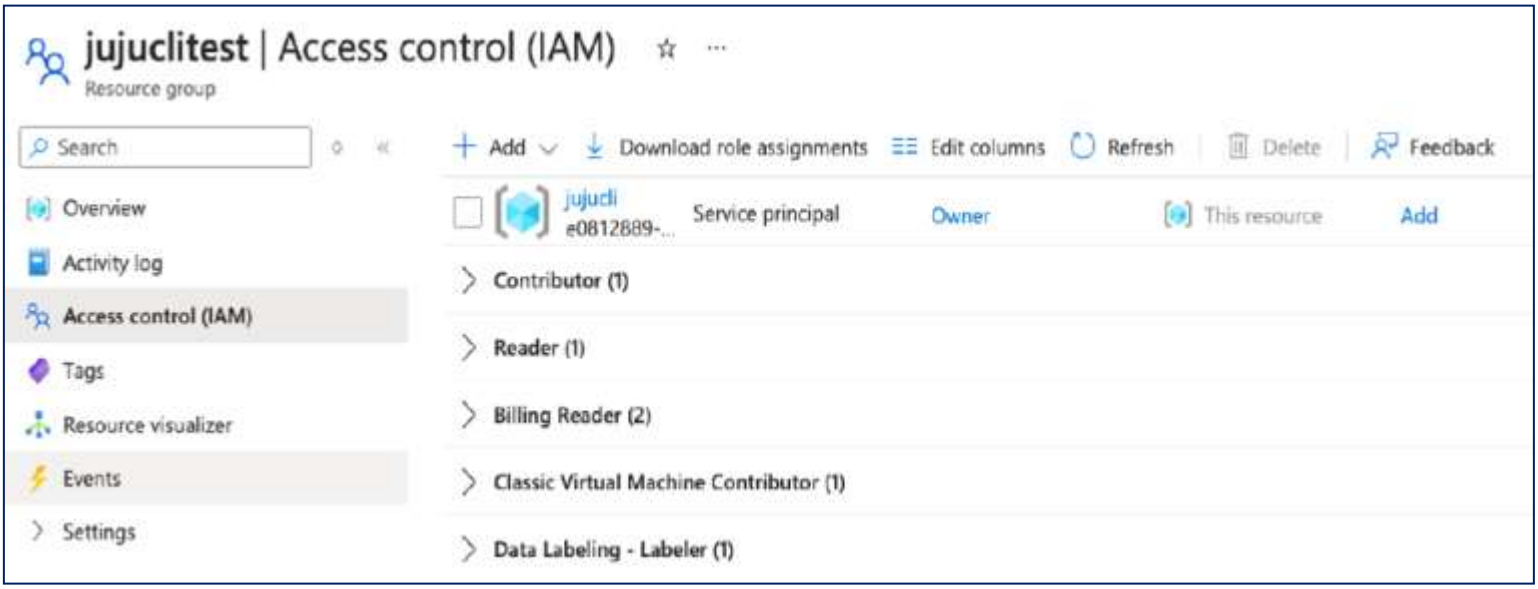
Puede ser en:

1. Un **grupo de recursos** específico
2. Toda la suscripción

```
$ az role assignment create --assignee $spObjectId --role Owner --scope /subscriptions/xxxxxx/resourcegroups/RG

$ az role assignment create --assignee $spObjectId --role Owner --scope /subscriptions/xxxxxx
```

5. Verificación en Azure

Dentro del grupo de recursos (o en la suscripción), en la pestaña **Access Control (IAM)**, deberíamos ver esta aplicación con el rol de **Owner**.



La imagen que se muestra corresponde a la sección **Access control (IAM)** de Azure para un **grupo de recursos** llamado jujucilitest.

Esta sección sirve para ver y gestionar los permisos de acceso que tienen usuarios, grupos o aplicaciones sobre ese recurso.

Se explica cada parte:

Encabezado

- jujucilitest → nombre del grupo de recursos.
- Access control (IAM) → es la pestaña donde se gestionan permisos y roles en Azure.

Elemento principal en la lista

- **jujucili** → es un *service principal* (una identidad de aplicación registrada en Microsoft Entra ID) que aquí se está usando para que **Juju CLI** pueda interactuar con Azure.
- **Owner** (*Propietario*) → rol asignado. Significa que esta identidad tiene **control total** sobre el recurso (puede crear, modificar o eliminar recursos y también asignar permisos a otros).

Otros roles que aparecen en la lista

Estos roles no necesariamente pertenecen al mismo *service principal*, sino a otros usuarios o aplicaciones que también tienen permisos en este grupo de recursos:

1. **Contributor (1)** → puede crear y administrar recursos, pero **no** puede asignar permisos a otros.
2. **Reader (1)** → solo puede ver los recursos, no modificarlos.
3. **Billing Reader (2)** → puede ver información de facturación.
4. **Classic Virtual Machine Contributor (1)** → permisos especiales para administrar máquinas virtuales clásicas.
5. **Data Labeling – Labeler (1)** → rol relacionado con el etiquetado de datos para *machine learning*.

En resumen:

La imagen confirma que tu *service principal* jujucili está configurado correctamente con rol **Owner** sobre el grupo de recursos jujucilitest, lo cual es un paso clave para que Juju pueda crear y administrar infraestructura en Azure sin restricciones.

Paso 6 – Crear el archivo `credentials.yaml`

Este archivo es un **documento de configuración** que guarda las credenciales de Azure que obtuviste antes (Application ID, contraseña secreta, y Subscription ID).
En el cliente (tu máquina donde usas Juju), el contenido sería algo como:

```yaml
credentials:
  azure:
    azure_cloud:
      auth-type: service-principal-secret
      application-id: xxxxx
      application-password: qqqqq
      subscription-id: zzzzz
```

- **application-id** → Es el **Client ID** del *service principal* (en tu caso, el de `jujucili`).
- **application-password** → Es el **Client Secret** que creaste para ese *service principal*.
- **subscription-id** → Es el ID de tu suscripción de Azure.
- **auth-type**: `service-principal-secret` indica que la autenticación será mediante un *service principal* y una contraseña secreta.

Paso 7 – Añadir credenciales a Juju

Con este comando:

```bash
juju add-credential -f credentials.yaml --client azure
```

- `-f credentials.yaml` → le pasas a Juju el archivo que acabas de crear.
- `--client azure` → le dices a Juju que estas credenciales son para el proveedor `azure` y se guardarán en el cliente local.

Esto guarda de forma segura los datos en el almacén de credenciales de Juju.

Paso 8 – Especificar región por defecto en Azure

```bash
juju default-region azure canadaeast
```

- Establece `canadaeast` como la región predeterminada de Azure para despliegues con Juju.
- Puedes cambiarla a cualquier otra región de Azure que soporte los recursos que quieres usar.

📌 En resumen:

1. El *service principal* que viste en la imagen es la identidad que Azure reconoce.
2. `credentials.yaml` le dice a Juju cómo autenticarse usando ese *service principal*.
3. `juju add-credential` importa esas credenciales.
4. `juju default-region` fija dónde se desplegarán las aplicaciones.

Iniciar (bootstrap) el controlador Juju y crear el modelo Juju para Charmed OpenSearch

Desplegar Charmed OpenSearch y configurar los servicios RAG

1.- Iniciar (bootstrap) el controlador Juju y crear el modelo para Charmed OpenSearch

Nota: El *controlador Juju* es como el "cerebro" que gestiona todos los despliegues y servicios. Se debe iniciar antes de poder instalar cualquier aplicación o servicio.

Ejecutar desde la **máquina cliente** (tu PC o servidor desde el que te conectas a Azure):

```bash
juju bootstrap --constraints allocate-public-ip=false --config resource-group-name=RG --config network=VNET azure
```

Explicación de parámetros:

- **allocate-public-ip=false:** Evita que las máquinas tengan IP pública por defecto (más seguro).
- **resource-group-name=RG:** Indica el *Resource Group* (grupo de recursos) de Azure donde se desplegará todo.
- **network=VNET:** Especifica la *Virtual Network* donde estarán los servicios.
- **azure:** Es el nombre del proveedor de nube donde se despliega (en este caso, Azure).

2.- Agregar el modelo

Nota: Un *modelo Juju* es como un “proyecto” o “espacio de trabajo” independiente donde se despliega un conjunto de aplicaciones relacionadas.

Desde la máquina cliente:

```
bash
juju add-model opensearch --config resource-group-name=RG --config network=VNET
```

Esto crea un modelo llamado **opensearch** dentro del mismo *Resource Group* y *VNET* que definimos antes.

3.- Configurar el modelo con parámetros para Charmed OpenSearch

Nota: Algunos servicios necesitan ajustes del sistema operativo antes de funcionar correctamente.
cloudinit-userdata es un archivo que Juju usa para aplicar configuraciones a las máquinas al momento de crearlas.

Crea el archivo con los parámetros necesarios:

```
bash

cat <<EOF > cloudinit-userdata.yaml
cloudinit-userdata: |
  postruncmd:
    - echo "vm.max_map_count=262144" | tee -a /etc/sysctl.conf
    - echo "vm.swappiness=0" | tee -a /etc/sysctl.conf
    - echo "net.ipv4.tcp_retries2=5" | tee -a /etc/sysctl.conf
    - echo "fs.file-max=1048576" | tee -a /etc/sysctl.conf
    - sudo sysctl -p
EOF
```

Explicación de parámetros del sistema:

- **vm.max_map_count=262144:** Ajusta el número máximo de áreas de memoria asignadas, necesario para OpenSearch.
- **vm.swappiness=0:** Indica que el sistema evite usar la memoria de intercambio (swap) salvo que sea necesario.
  net.ipv4.tcp_retries2=5: Reduce el número de reintentos TCP para conexiones lentas o fallidas.
- **fs.file-max=1048576:** Aumenta el número máximo de descriptores de archivo, útil para manejar muchas conexiones.

4.- Aplicar la configuración al modelo

Desde la máquina cliente:

```
bash
juju model-config --file=./cloudinit-userdata.yaml
```

Nota: Este comando carga las configuraciones del archivo y las aplica al modelo, asegurando que cada máquina que se despliegue cumpla con estos parámetros desde el inicio.

Desplegar Charmed OpenSearch Dashboards (opcional, pero recomendado)

Añadir máquinas y desplegar Charmed OpenSearch en alta disponibilidad

1.- Añadir máquinas al modelo Juju

Nota: Antes de desplegar Charmed OpenSearch, necesitamos máquinas (VMs) en Azure para alojar cada nodo.
Usaremos **3 máquinas** para configurar un clúster **altamente disponible**.
Esto significa que si una falla, las otras seguirán funcionando y manteniendo los datos accesibles.

Ejecutar en la máquina cliente:

```
bash
juju add-machine --constraints="instance-type=Standard_DC4es_v5 allocate-public-ip=false
zones=canadaeast" -n 3
```

Explicación de parámetros:

- **instance-type=Standard_DC4es_v5:** Tipo de máquina virtual en Azure, con CPU y RAM adecuadas para cargas de búsqueda y análisis.
- **allocate-public-ip=false:** Evita asignar IP pública (mejora seguridad).
- **zones=canadaeast:** Especifica la región o zona de disponibilidad en Azure.
- **-n 3:** Crea 3 máquinas idénticas con estas características.

Consejo: Usar varias zonas de disponibilidad puede mejorar la resiliencia, pero aquí usamos la misma para simplificar.

2.- Desplegar Charmed OpenSearch en las máquinas creadas

Nota: Ahora asignaremos la aplicación **OpenSearch** a estas 3 máquinas para que cada una sea un nodo del clúster.

Ejecutar:

```
bash
juju deploy opensearch --channel 2/edge --to 0,1,2
```

Explicación de parámetros:

- **opensearch:** Nombre del charm (paquete administrado por Juju) que vamos a instalar.
- **--channel 2/edge:** Usa la versión de la serie 2.x del charm en su canal *edge* (últimas actualizaciones, pero menos estable que *stable*).
- **--to 0,1,2:** Indica que los nodos de OpenSearch se instalen en las máquinas **0, 1 y 2** (las que acabamos de crear con add-machine).

Resultado esperado:

Después de esto, tendrás un clúster de OpenSearch de 3 nodos, cada uno en una máquina distinta, listos para indexar y buscar datos con alta disponibilidad.

Configurar TLS y finalizar el despliegue de Charmed OpenSearch

3.- Habilitar comunicación segura con TLS

Nota: Los nodos de un clúster de OpenSearch intercambian información entre sí constantemente.

Para evitar que los datos viajen sin cifrar, se requiere **TLS** (*Transport Layer Security*).
Si TLS no está configurado, el charm de OpenSearch quedará **bloqueado** (*blocked*) hasta que reciba certificados válidos.

Para este tutorial, usaremos el operador **tls-certificates-operator** con un certificado **autofirmado** (self-signed).

a) Desplegar el operador de certificados TLS

```
bash
juju deploy tls-certificates-operator
```

Esto instala el servicio que generará y gestionará certificados.

b) Configurar el operador para usar certificados autofirmados

```
bash
juju config tls-certificates-operator generate-self-signed-certificates="true" ca-common-name="Tutorial CA"
```

Explicación de parámetros:

- generate-self-signed-certificates="true": Indica que el operador debe generar sus propios certificados sin una autoridad externa.
- ca-common-name="Tutorial CA": Nombre identificador de la autoridad certificadora interna que emitirá los certificados.

Nota de producción: En entornos reales, lo ideal es usar certificados de una **CA externa confiable**.

c) Relacionar el operador TLS con OpenSearch

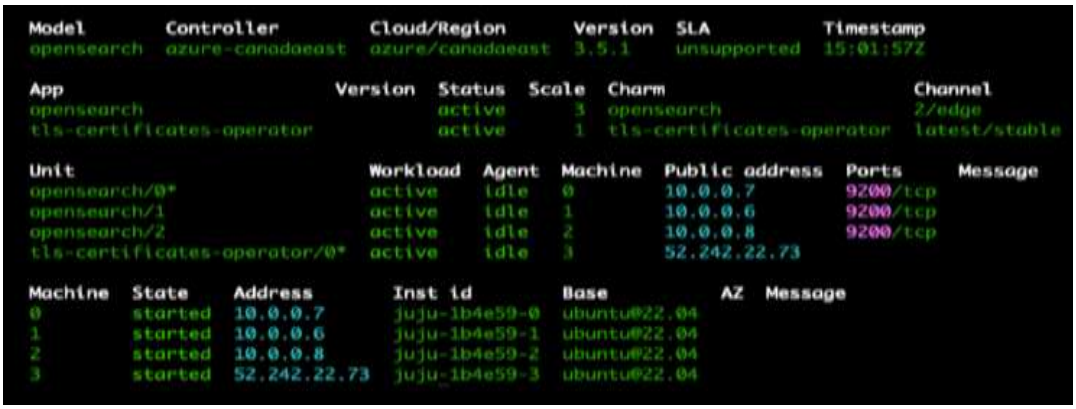
```
bash
juju relate tls-certificates-operator opensearch
```

Esto conecta ambos charms para que OpenSearch reciba automáticamente los certificados y los use en todas las comunicaciones entre nodos.

4.- Verificar que OpenSearch esté activo

Después de unos minutos (mientras Juju aplica las configuraciones y reinicia los servicios), el clúster de Charmed OpenSearch debería aparecer en estado:

En la imagen siguiente podemos confirmar que el estado final esperado del despliegue se cumplió.



Verificación del estado de Charmed OpenSearch con TLS configurado

Después de ejecutar:

```
bash
juju status
```

vemos lo siguiente en la salida:

- **Modelo:** opensearch
- **Aplicaciones:**
  - opensearch → **active**, con 3 unidades (una en cada máquina 0, 1 y 2).
  - tls-certificates-operator → **active**, con 1 unidad (en máquina 3).
- **Máquinas:**
  - 0 → IP interna 10.0.0.7
  - 1 → IP interna 10.0.0.6
  - 2 → IP interna 10.0.0.8
  - 3 → IP pública 52.242.22.73 (donde corre el operador TLS)
- **Puertos abiertos:** 9200/tcp para los 3 nodos de OpenSearch.

Interpretación:

- El clúster de OpenSearch está activo y distribuido en tres nodos, con comunicación segura vía TLS.
- El operador de certificados está activo y entregó correctamente los certificados a OpenSearch.
- Todos los agentes (Agent) están en estado **idle** (sin tareas pendientes), lo que indica que la configuración se completó.

Obtener credenciales y registrar modelos para RAG en OpenSearch

5.- Obtener credenciales del usuario administrador

Nota: Para interactuar con OpenSearch (por ejemplo, registrar modelos) necesitamos credenciales de administrador.

Juju facilita esto porque el charm de OpenSearch guarda la contraseña de admin de forma segura.

Ejecutar en la máquina cliente:

```
bash
juju run opensearch/leader get-password
```

Ejemplo de salida:

```
pgsql
# Ejemplo de uso de variables de entorno
password: os-password (saved)
username: admin
```

Ahora exportamos las credenciales y la dirección de OpenSearch como variables de entorno, para no escribirlas en cada comando:

```
bash
# Configuración de variables de entorno para OpenStack
export OS_PASSWORD=<os-password>
export OS_USERNAME=admin
export OS_ENDPOINT=https://10.0.0.7:9200
```

Tip: 10.0.0.7 es la IP interna de uno de los nodos OpenSearch.
En producción podrías usar un *load balancer* o un nombre DNS en lugar de una IP fija.

6.- Registrar un grupo de modelos

Nota: Un *grupo de modelos* es una categoría en OpenSearch para organizar modelos de Machine Learning (ML).
Esto facilita la gestión cuando tienes varios modelos para distintas tareas.

Ejecutar:

```
bash
curl -u "$OS_USERNAME:$OS_PASSWORD" \
-XPOST "$OS_ENDPOINT/_plugins/_ml/model_groups/_register" \
-H 'Content-Type: application/json' -d'
{
  "name": "rag-model-group",
  "description": "A model group for RAG use case models"
}'
```

Ejemplo de respuesta:

```
Json
{"model_group_id":"lW0H55EBiw8MvxrDTQdM","status":"CREATED"}
```

Guarda el model_group_id (lW0H55EBiw8MvxrDTQdM) para el siguiente paso.

7.- Registrar un modelo *Sentence Transformer*

Nota: En un sistema RAG, un *Sentence Transformer* convierte texto en vectores numéricos para búsquedas semánticas.

Elegimos un modelo preentrenado proporcionado por OpenSearch:

```
bash
huggingface/sentence-transformers/msmarco-distilbert-base-tas-b
```

Este trabaja con un espacio vectorial de **768 dimensiones**.

Ejecutar:

```
bash
CopiarEditar
curl -u "$OS_USERNAME:$OS_PASSWORD" \
-XPOST "$OS_ENDPOINT/_plugins/_ml/models/_register" \
-H 'Content-Type: application/json' -d'
{
  "name": "huggingface/sentence-transformers/msmarco-distilbert-base-tas-b",
  "version": "1.0.2",
  "model_group_id": "lW0H55EBiw8MvxrDTQdM",
  "model_format": "TORCH_SCRIPT"
}'
```

Ejemplo de respuesta:

```
json
{"task_id":"lm0H55EBiw8MvxrDqwf5","status":"CREATED"}
```

Guarda el task_id (lm0H55EBiw8MvxrDqwf5) para comprobar el progreso.

8.- Verificar el estado del registro del modelo

Ejecutar:

```
bash
curl -u "$OS_USERNAME:$OS_PASSWORD" \
-XGET "$OS_ENDPOINT/_plugins/_ml/tasks/lm0H55EBiw8MvxrDqwf5"
```

Ejemplo de respuesta:

```
json
{
  "model_id":"l20H55EBiw8MvxrDrwcL",
  "task_type":"REGISTER_MODEL",
  "function_name":"TEXT_EMBEDDING",
  "state":"COMPLETED",
  "worker_node":["hy5M6dC9SqW2HR0QCRt5lw"],
  "create_time":1726157925367,
  "last_update_time":1726157967875,
  "is_async":true
}
```

Interpretación de campos clave:

- "state": "COMPLETED" → El modelo se registró con éxito.
- "model_id" → Identificador único del modelo (lo necesitarás para usarlo en consultas).
- "function_name": "TEXT_EMBEDDING" → Indica que el modelo convierte texto en vectores.

9: Desplegar el modelo *Sentence Transformer*

Nota: En el paso anterior registramos el modelo en OpenSearch, pero aún no está activo.

Registrar un modelo significa que OpenSearch sabe que existe y tiene sus metadatos.

Desplegarlo significa cargarlo en memoria en los nodos para que pueda responder consultas inmediatamente.

a) Ejecutar el despliegue

Usa el model_id que obtuviste en el paso anterior (en este ejemplo, l20H55EBiw8MvxrDrwcL):

```
bash
curl -u "$OS_USERNAME:$OS_PASSWORD" \
-XPOST "$OS_ENDPOINT/_plugins/_ml/models/l20H55EBiw8MvxrDrwcL/_deploy"
```

Ejemplo de respuesta:

```
json
{
  "task_id":"mG0I55EBiw8MvxrDlAeV",
  "task_type":"DEPLOY_MODEL",
  "status":"CREATED"
}
```

b) Interpretar la respuesta

- **"task_id"** → Identificador de la tarea de despliegue (puedes usarlo para comprobar el progreso).
- **"task_type": "DEPLOY_MODEL"** → Indica que es un despliegue, no un registro.
- **"status": "CREATED"** → La tarea fue creada y está en ejecución.

Tip: Igual que en pasos anteriores, puedes verificar el estado de este despliegue con:

```
bash
CopiarEditar
curl -u "$OS_USERNAME:$OS_PASSWORD" \
-XGET "$OS_ENDPOINT/_plugins/_ml/tasks/mG0I55EBiw8MvxrDlAeV"
```

Cuando el estado pase a **"COMPLETED"**, el modelo estará cargado y listo para recibir solicitudes de *text embedding*.

10.- Probar el modelo desplegado (*Predict API*)

Una vez que el modelo *Sentence Transformer* está desplegado, podemos enviarle texto y obtener su **vector de embeddings**.

Este vector es una representación numérica (dimensión 768 en este ejemplo) que OpenSearch usará para búsquedas semánticas o similares.

a) Ejecutar la prueba con *Predict API*

Usa el model_id del modelo desplegado (en este ejemplo l20H55EBiw8MvxrDrwcL):

```
bash
curl -u "$OS_USERNAME:$OS_PASSWORD" \
-XPOST "$OS_ENDPOINT/_plugins/_ml/_predict/text_embedding/l20H55EBiw8MvxrDrwcL" \
-H 'Content-Type: application/json' -d'
{
  "text_docs": ["What is Ubuntu Pro?"],
  "return_number": true,
  "target_response": ["sentence_embedding"]
}'
```

b) Interpretar los parámetros

- "text_docs" → Lista de textos a convertir en embeddings.
- "return_number": true → Devuelve los valores numéricos reales del vector.
- "target_response": ["sentence_embedding"] → Especifica que queremos el embedding resultante.

c) Ejemplo de respuesta

```
json
{
  "inference_results": [
    {
      "output": [
        {
          "name": "sentence_embedding",
          "data_type": "FLOAT32",
          "shape": [768],
          "data": [ ... ]
        }
      ]
    }
  ]
}
```

- **"shape":** [768] → El vector tiene 768 dimensiones.
- **"data":** [...] → Contiene los valores numéricos del embedding, que usarás para indexar o comparar texto.

Tip práctico:
Si quieres probar con varias frases, puedes poner más entradas en "text_docs". Por ejemplo:

```
json
"text_docs": [
  "What is Ubuntu Pro?",
  "How to install OpenSearch?"
]
```

Recibirás un embedding por cada frase, en el mismo orden en que las enviaste.

Esto último quiere decir que:
Cuando envías varias frases en la propiedad "text_docs" de la solicitud, OpenSearch procesa **cada frase por separado** con el modelo de *sentence embedding*.

- Cada frase se convierte en un **vector numérico** (embedding) independiente.
- El orden en la respuesta **coincide exactamente** con el orden en que enviaste las frases en la lista.
- Es decir, el *primer vector* corresponde a la *primera frase*, el *segundo vector* a la *segunda frase*, y así sucesivamente.

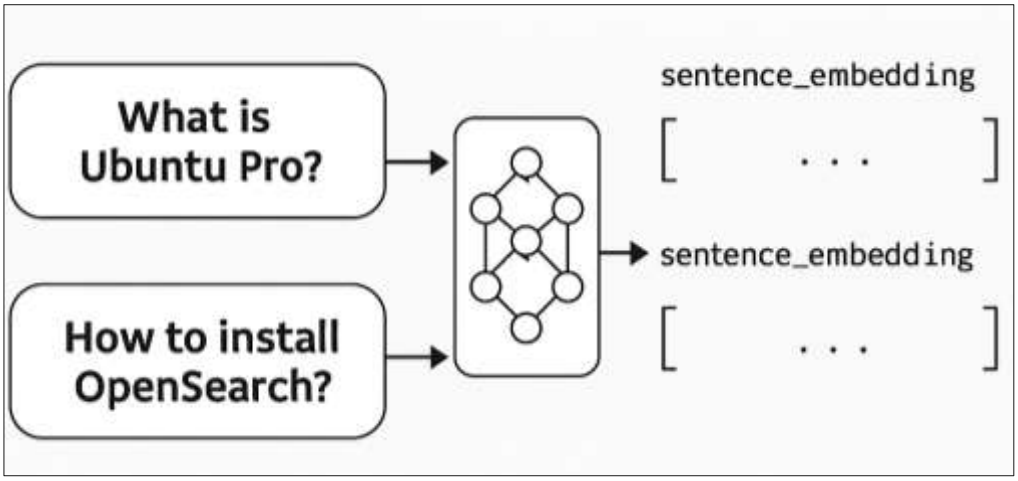
Ejemplo:

```
json
CopiarEditar
"text_docs": [
 "What is Ubuntu Pro?",
 "How to install OpenSearch?"
]
```

Respuesta simplificada:

```
json
CopiarEditar
"inference_results": [
 {
  "output": [
   {
    "name": "sentence_embedding",
    "shape": [768],
    "data": [ ... ]   // Embedding de la frase 1
   },
   {
    "name": "sentence_embedding",
    "shape": [768],
    "data": [ ... ]   // Embedding de la frase 2
   }
  ]
 }
]
```

Así puedes saber **qué vector pertenece a qué texto** sin necesidad de buscar o reordenar nada.



- Cuando el texto dice: *"Recibirás un embedding por cada frase, en el mismo orden en que las enviaste"*, se refiere a que el modelo de *sentence transformer* procesa cada frase que le envías y genera un **vector numérico (embedding)** para cada una.
- Si envías **una sola frase**, recibirás **un solo vector**.
- Si envías **varias frases en un arreglo** (text_docs), el modelo devolverá un **vector por cada frase** y **los pondrá en la misma secuencia** en que las enviaste.

Esto es importante porque el embedding **no mezcla ni reordena** las frases:

1. Frase 1 → Vector 1
2. Frase 2 → Vector 2
3. Frase 3 → Vector 3

Cada vector tendrá siempre la misma **dimensionalidad** que el modelo (en tu ejemplo, 768 números en cada embedding).

El siguiente es un ejemplo con 2 frases y cómo quedarían sus embeddings alineados uno con otro.

Imaginemos que se envían estas dos frases al modelo:

```
Json

CopiarEditar
"text_docs": [
 "What is Ubuntu Pro?",
 "How do I install OpenSearch?"
]
```

El modelo devolverá algo parecido a:

```
Json
CopiarEditar
"inference_results": [
  {
    "output": [
      {
        "name": "sentence_embedding",
        "data_type": "FLOAT32",
        "shape": [768],
        "data": [0.0123, -0.0567, 0.0789, ..., 0.0456]  ← Embedding de la frase 1
      },
      {
        "name": "sentence_embedding",
        "data_type": "FLOAT32",
        "shape": [768],
        "data": [-0.0345, 0.0890, -0.1203, ..., -0.0112]  ← Embedding de la frase 2
      }
    ]
  }
]
```

Aspectos Clave del ejemplo:

1. **Frase 1** ("What is Ubuntu Pro?") → Primer vector de 768 valores.
2. **Frase 2** ("How do I install OpenSearch?") → Segundo vector de 768 valores.
3. El orden de los embeddings **coincide exactamente** con el orden de las frases que enviaste.

Esto es crucial en aplicaciones RAG, porque si cambias el orden de los embeddings respecto a las frases originales, las búsquedas y relaciones semánticas dejarían de tener sentido.

Qué significan estos valores de los vectores correspondientes a cada una de las frases:

```
"data": [0.0123, -0.0567, 0.0789, ..., 0.0456]  ← Embedding de la frase 1
"data": [-0.0345, 0.0890, -0.1203, ..., -0.0112]  ← Embedding de la frase 2
```

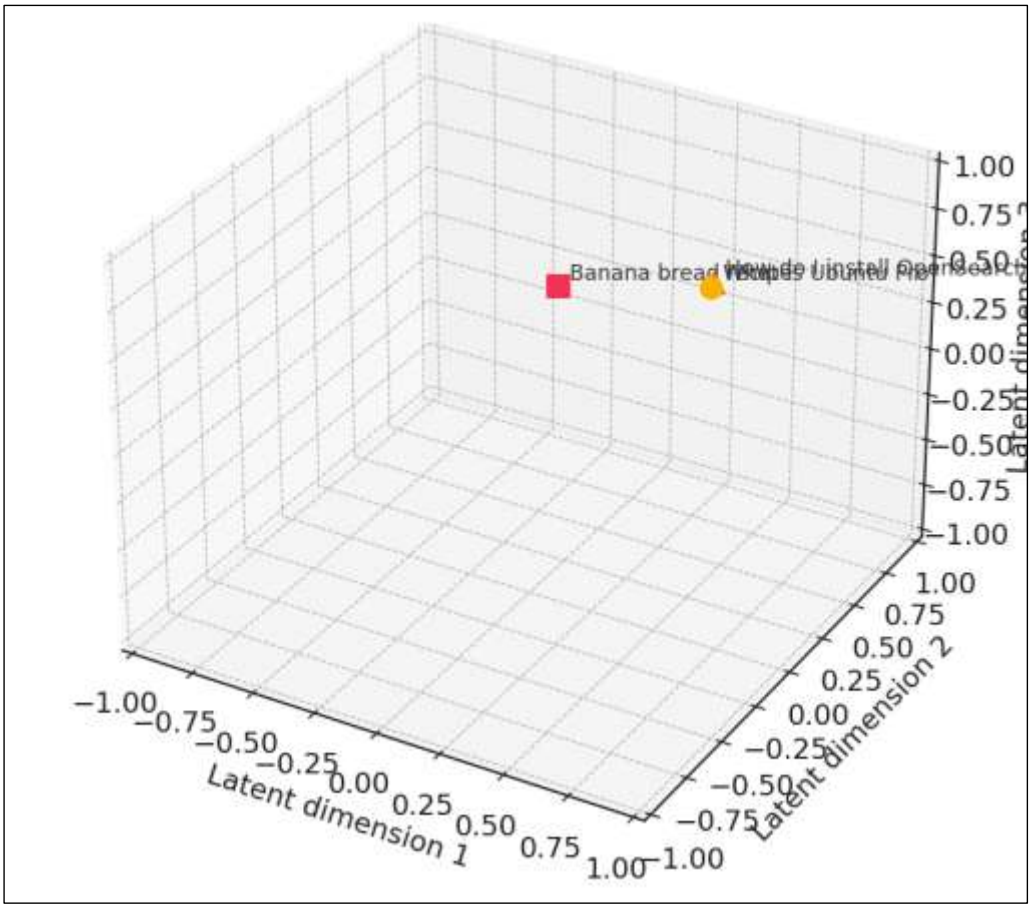
En un *embedding*, cada valor del vector representa una coordenada en un espacio matemático de alta dimensión (en nuestro caso, 768 dimensiones).

Pensemos esto así:

- Cada dimensión captura un rasgo abstracto del significado de la frase (por ejemplo, cierta relación semántica, contexto, tono, etc.).
- No es que la dimensión 1 sea “positividad” o la dimensión 2 sea “longitud de frase” — no tienen un significado humano directo.
- Juntas, las 768 coordenadas forman un punto único que representa tu frase en ese espacio semántico.
- Frases con significados parecidos estarán más cerca entre sí en ese espacio, lo que facilita búsquedas y comparaciones por similitud.

En resumen: Esos números son la “huella matemática” de tu texto, diseñada para que las máquinas entiendan y comparen significado.

Analogía de Embedding en 3D (Conceptualización)



Cómo leerlo (explicación clara y completa)

1.- Qué son esos números

- Cada embedding es un **vector**; en tu modelo real tiene **768** coordenadas.
- En la ilustración lo reducimos a **3 dimensiones** para que sea visible: piensa en tres ejes llamados “Latent dimension 1/2/3”.
- Los valores no son “etiquetas humanas” (no existe “la dimensión de positividad”). Son rasgos **abstractos** aprendidos por el modelo para ubicar significados.

2.- Qué muestra el gráfico

- Cada oración es un **punto** en ese espacio.
- Si dos oraciones “significan cosas parecidas”, sus puntos quedan **cerca**.
- En el ejemplo, *“What is Ubuntu Pro?”* y *“How do I install OpenSearch?”* aparecen próximas; *“Banana bread recipe”* queda lejos.

3.- Cómo medir parecido (lo que importa de esos valores)

Los números sirven para calcular **proximidad** entre vectores:

3.1.- Distancia euclidiana *(menor = más parecido).*

$d(a,b) = \|a-b\| = d(\mathbf{a},\mathbf{b}) = \lVert \mathbf{a}-\mathbf{b}\rVert d(a,b)$

¿Qué significa?

Cuando tienes dos vectores, por ejemplo:

```
ini
a = [a₁, a₂, a₃, ..., aₙ]
b = [b₁, b₂, b₃, ..., bₙ]
```

La distancia euclidiana es:

1. Restar componente a componente: $(a_1 - b_1), (a_2 - b_2), (a_3 - b_3), ..., (a_n - b_n)$
2. Elevar cada diferencia al cuadrado.
3. Sumar todos esos cuadrados.
4. Sacar la raíz cuadrada del total.

Un ejemplo pequeño (3 dimensiones)

Digamos que:

```
ini
a = [2, 3, 5]
b = [1, 1, 2]
```

Cálculo paso a paso:

1. **Diferencias:** (2-1) = 1, (3-1) = 2, (5-2) = 3
2. **Cuadrados:** $1^2 = 1, 2^2 = 4, 3^2 = 9$
3. **Suma:** 1 + 4 + 9 = 14
4. **Raíz cuadrada:** $\sqrt{14} \approx 3.74$

Eso significa que la **distancia euclidiana** entre esos vectores es aproximadamente **3.74**.

Por qué se escribió así antes: d(a,b) = .|| a−b||

es simplemente la forma “compacta” de decir "distancia euclidiana = longitud del vector diferencia".

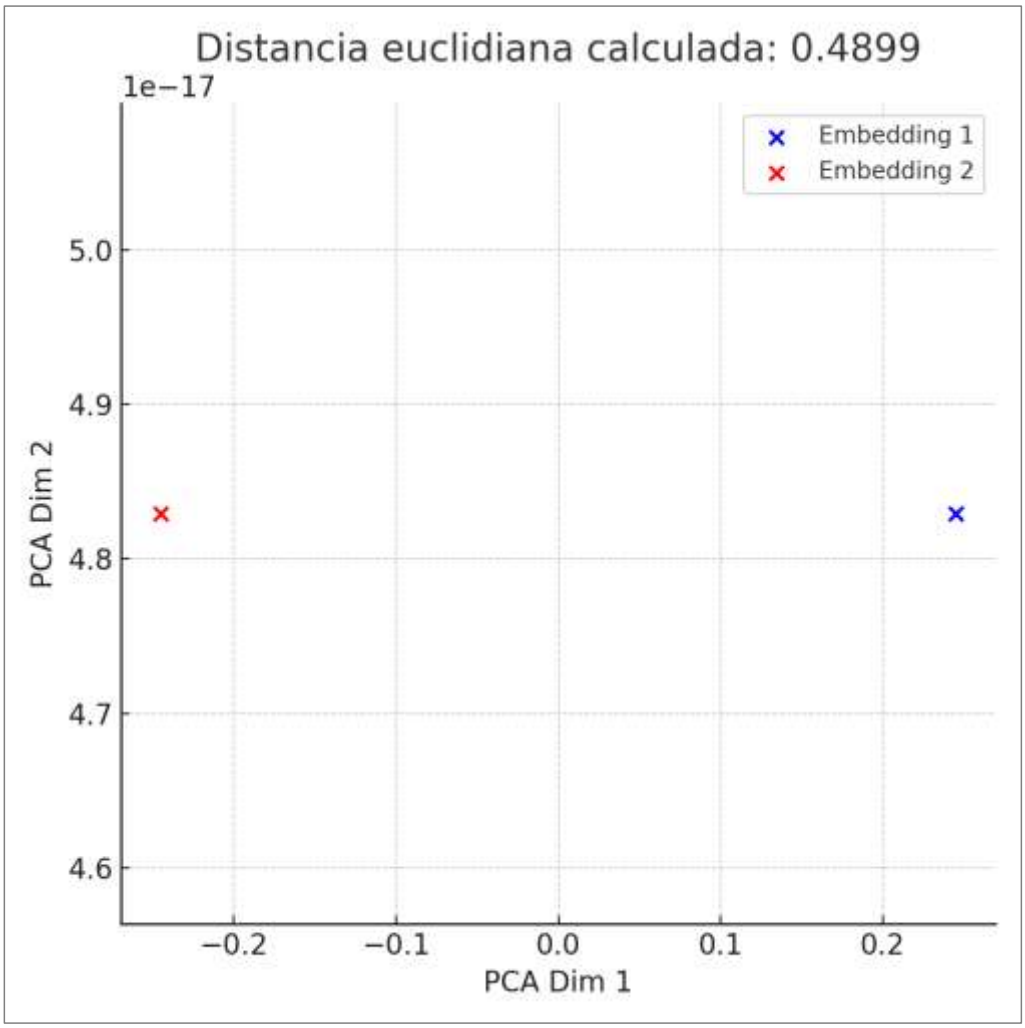
- **||...||** (doble barra) significa “longitud del vector”.
- **a - b** es el vector diferencia.
- Esa longitud se calcula con el método que acabo de mostrar **(cuadrar, sumar, raíz).**

Python

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Ejemplo: dos "mini embeddings" de 5 dimensiones (normalmente serían de 768)
embedding_1 = np.array([0.2, 0.8, -0.5, 0.3, 0.1])
embedding_2 = np.array([0.4, 0.5, -0.2, 0.4, 0.0])

# Cálculo paso a paso de la distancia euclidiana
diff = embedding_1 - embedding_2
squared_diff = diff ** 2
sum_squared_diff = np.sum(squared_diff)
distance = np.sqrt(sum_squared_diff)
# Reducción a 2D para graficar
pca = PCA(n_components=2)
reduced = pca.fit_transform(np.vstack([embedding_1, embedding_2]))

plt.figure(figsize=(6,6))
plt.scatter(reduced[0,0], reduced[0,1], color="blue", label="Embedding 1")
plt.scatter(reduced[1,0], reduced[1,1], color="red", label="Embedding 2")
for i, txt in enumerate(["E1", "E2"]):
    plt.annotate(txt, (reduced[i,0]+0.01, reduced[i,1]+0.01))

plt.title(f"Distancia euclidiana calculada: {distance:.4f}")
plt.xlabel("PCA Dim 1")
plt.ylabel("PCA Dim 2")
plt.legend()
plt.grid(True)
plt.tight_layout()

file_path = "/mnt/data/distancia_embeddings_demo.png"
plt.savefig(file_path)

file_path, diff, squared_diff, sum_squared_diff, distance
```

Resultado esperado

```
('/mnt/data/distancia_embeddings_demo.png',
 array([-0.2,  0.3, -0.3, -0.1,  0.1]),
 array([0.04, 0.09, 0.09, 0.01, 0.01]),
 0.24000000000000005,
 0.48989794855663565)
```



Paso a paso del cálculo de la distancia euclidiana entre dos embeddings

Supongamos que tenemos dos vectores (embeddings):

E1 = [0.2, 0.8, −0.5, 0.3, 0.1]
E2 = [0.4, 0.5, −0.2, 0.4, 0.0]

Paso 1 – Restar componente a componente:

E1−E2 = [−0.2, 0.3, −0.3, −0.1, 0.1]

Paso 2 – Elevar cada diferencia al cuadrado:

$[(-0.2)^2, (0.3)^2, (-0.3)^2, (-0.1)^2, (0.1)^2]$ = [0.04, 0.09, 0.09, 0.01, 0.01]

Paso 3 – Sumar todos los cuadrados:

0.04 + 0.09 + 0.09 + 0.01 + 0.01 = 0.24

Paso 4 – Sacar raíz cuadrada:

$\sqrt{0.24} \approx 0.4899$

Este número **0.4899** es la **distancia euclidiana** entre los dos embeddings.
Cuanto más pequeño, más parecidos son.

Interpretación

- **Cada valor** del embedding representa una característica numérica de la frase que el modelo ha aprendido (no es un "significado" directo palabra a palabra).
- La distancia euclidiana es una forma de medir **qué tan "lejos" o "cerca"** están dos frases en el espacio semántico.
- Si la distancia es **cercana a 0**, significa que las frases son muy similares en significado.
- Si es **grande**, significa que son semánticamente diferentes.

3.2.- Similitud coseno *(mayor, cercano a 1 = más parecido).*

cos_sim(a,b) = a·b ||a|| ||b||

En la **tabla** que te mostré, verás que el par de preguntas técnicas tiene **distancia baja** y **similitud coseno alta**, mientras que cualquiera de ellas con *"Banana bread recipe"* muestra lo contrario. Eso es exactamente lo que explotan las búsquedas semánticas y el RAG.

4.- Cómo usarlo en la práctica (OpenSearch + RAG)

Al indexar documentos, guardas también sus vectores (con la misma dimensionalidad que el modelo, p. ej., 768).

Al consultar, conviertes la pregunta del usuario a un vector y buscas en el índice los vectores más cercanos (kNN por coseno/inner product).

Los textos con vectores más cercanos son tu contexto para el LLM.

5.- Ideas clave para recordar

- Los valores individuales **no son interpretables**; **las relaciones** (distancias/similitudes) sí.
- El orden en la respuesta del *Predict API* **coincide** con el orden de tus entradas, así puedes alinear cada frase con su embedding sin ambigüedad.
- Mantén consistente el **modelo** y la **dimensionalidad** entre indexación y consulta.

11.- Crear una *"ingest pipeline"*

En OpenSearch para que, cada vez que se ingrese un documento, automáticamente se genere su *embedding* usando el modelo de *sentence transformer* que ya se desplegó.

Comando:

```
bash
curl -u '$OS_USERNAME:$OS_PASSWORD' \
-XPUT "$OS_ENDPOINT/_ingest/pipeline/rag-ingest-pipeline" \
-H 'Content-Type: application/json' -d'
{
  "description": "An RAG ingest pipeline",
  "processors": [
    {
      "text_embedding": {
        "model_id": "l20H55EBiw8MvxrDrwcL",
        "field_map": {
          "text": "sentence_embedding"
        }
      }
    }
  ]
}'
```

Se explica línea por línea:

1.- Autenticación y destino

- `-u '$OS_USERNAME:$OS_PASSWORD'` → Envía el usuario y contraseña de OpenSearch.
- `"$OS_ENDPOINT/_ingest/pipeline/rag-ingest-pipeline"` → Indica que vas a crear un *pipeline* llamado **`rag-ingest-pipeline`**.

2.- Método HTTP

- `-XPUT` → Crea o sobrescribe el *pipeline*.

3.- Cabecera

- `-H 'Content-Type: application/json'` → Le dice a OpenSearch que estás enviando datos en formato JSON.

4.- Contenido del pipeline

- `"description"` → Texto descriptivo, aquí indica que es para RAG (*Retrieval-Augmented Generation*).
- `"processors"` → Lista de pasos que OpenSearch ejecutará cada vez que ingrese un documento.
  - **`text_embedding`** → Tipo de procesador que usa un modelo ML para convertir texto en un vector.
    - `"model_id": "l20H55EBiw8MvxrDrwcL"` → Es el ID del modelo que desplegaste antes.
    - `"field_map": { "text": "sentence_embedding" }` → Le dice al procesador:
      - Toma el contenido del campo `"text"` de cada documento que llegue.
      - Guarda el embedding generado en el campo `"sentence_embedding"`.

En resumen

Con esto, ya no necesitas llamar manualmente a la API de *predict* para cada texto.

Si indexas un documento como:

```
Json

CopiarEditar
{
  "text": "What is Ubuntu Pro?"
}
```

OpenSearch automáticamente generará algo como:

```
json

{
  "text": "What is Ubuntu Pro?",
  "sentence_embedding": [0.0123, -0.987, ... , 0.1345] // vector de 768 dimensiones
}
```

12.- Crear un índice kNN con el *pipeline* de ingesta.

Este paso es muy importante porque es donde se prepara el índice vectorial en OpenSearch para poder buscar por similitud semántica usando los embeddings generados en el paso anterior.

Comando:

```
Json

# Crear un índice KNN en OpenSearch con FAISS y AVX2
# Asegúrate de tener las variables de entorno OS_USERNAME, OS_PASSWORD y OS_ENDPOINT configuradas
curl -u '$OS_USERNAME:$OS_PASSWORD' \
-XPUT "$OS_ENDPOINT/knn-faiss-avx2-index1" \
-H 'Content-Type: application/json' -d'
{
  "settings": {
    "index.knn": true,
    "default_pipeline": "rag-ingest-pipeline"
  },
  "mappings": {
    "properties": {
      "id": {
        "type": "text"
      },
      "sentence_embedding": {
        "type": "knn_vector",
        "dimension": 768,
        "method": {
          "name": "hnsw",
          "engine": "faiss",
          "space_type": "l2",
          "parameters": {
            "encoder": {
              "name": "sq",
              "parameters": {
                "type": "fp16"
              }
            },
            "ef_construction": 256,
            "m": 8
          }
        }
      },
      "text": {
        "type": "text"
      }
    }
  }
}'
```

12.1.- settings

- index.knn: true → Activa el soporte de búsquedas k-Nearest Neighbors (kNN) en este índice.
- default_pipeline: "rag-ingest-pipeline"` → Asocia este índice al pipeline creado antes.

Esto significa que cada documento insertado pasará por el pipeline y recibirá automáticamente su embedding.

12.2.- mappings

Define la estructura y tipos de datos de cada campo del índice.

Id

Tipo: "text" → Un campo normal para guardar un identificador (puede ser alfanumérico).
sentence_embedding

- **Tipo:** "knn_vector" → Es un vector que se usará para hacer búsquedas de similitud.
- **dimension:** 768 → Coincide con la dimensionalidad del modelo *sentence transformer* que usaste.
- **method** → Configura el motor de búsqueda vectorial.
    - **"name":** "hnsw" → Algoritmo **Hierarchical Navigable Small World** para búsqueda rápida en grandes volúmenes.
    - **"engine":** "faiss" → Usas FAISS, una librería optimizada de Facebook para búsqueda vectorial.
    - **"space_type":** "l2" → La métrica de distancia es **L2** (*Euclidean distance*).
    - **"parameters":**
        - **"encoder":** { "name": "sq", "parameters": { "type": "fp16" } } → Comprime el vector usando cuantización escalar (*scalar quantization*) y media precisión (*float16*) para ahorrar memoria.
        - **"ef_construction":** 256 → Controla la precisión/velocidad durante la construcción del índice.
        - **"m":** 8 → Número de conexiones entre nodos en el grafo HNSW (impacta precisión y memoria).

Text

- **Tipo:** "text" → Guarda el texto original, útil para devolver resultados legibles.

12.3.- Respuesta

```
json

{"acknowledged":true,"shards_acknowledged":true,"index":"knn-faiss-avx2-index1"}
```

Esto confirma que el índice ***knn-faiss-avx2-index1*** se creó correctamente.

En resumen:

Este índice es el "contenedor" donde se ha de guardar documentos que ya estarán enriquecidos con embeddings generados automáticamente.

Luego se podrán hacer consultas del tipo: *"Dame los 5 documentos más similares a este texto"* y OpenSearch te devolverá resultados basados en el significado, no solo en palabras exactas.

13.- Carga de documentos en el índice *(que luego puedan usarse como contexto para la generación aumentada RAG).*

En un flujo RAG, el modelo no genera la respuesta solo desde lo que "sabe" por entrenamiento, sino que primero busca en una base de datos vectorial (el índice kNN que creaste) fragmentos relevantes que funcionen como contexto.

Comando:

```
bash

# Ejemplo de uso de cURL para enviar una solicitud POST a un endpoint específico
# Asegúrate de reemplazar las variables de entorno con tus credenciales y endpoint
reales
# Reemplaza las siguientes variables con tus credenciales y endpoint
OS_USERNAME="your_username"
OS_PASSWORD="your_password"
OS_ENDPOINT="https://your-endpoint.com/api"

# Ejemplo de cURL para enviar un documento a un índice específico
# Asegúrate de que el índice y el documento sean correctos según tu configuración
curl -u '$OS_USERNAME:$OS_PASSWORD' \
-XPOST "$OS_ENDPOINT/knn-faiss-avx2-index1/_doc" \
-H 'Content-Type: application/json' -d'
{
"text": "..."
}'
```

1.- curl

→ Es la herramienta en consola para enviar solicitudes HTTP. Aquí se usa para enviar datos al servidor de OpenSearch.

2.- -u '$OS_USERNAME:$OS_PASSWORD'

→ Indica autenticación básica con usuario y contraseña que ya exportaste antes como variables de entorno (OS_USERNAME y OS_PASSWORD).

3.- -XPOST

→ Especifica que esta es una solicitud HTTP POST (crear un nuevo documento).

4.- "$OS_ENDPOINT/knn-faiss-avx2-index1/_doc"

→ URL del índice donde se guardará el documento:

- knn-faiss-avx2-index1 es el nombre del índice que creaste en el paso 12.
- /_doc indica que quieres insertar un documento nuevo.

5.- -H 'Content-Type: application/json'

→ Le dice al servidor que el contenido que envías está en formato JSON.

6.- -d '{ "text": "..." }'

→ Es el cuerpo del documento que quieres indexar. Aquí "text" es el campo que definiste en la estructura del índice y "..." sería el contenido real (por ejemplo, un párrafo de información).

¿Qué pasa internamente?

Cuando envías este documento:

- El **pipeline de ingesta** que configuraste antes (paso 11) toma el campo `"text"`.
- Ese texto pasa por el **modelo de embeddings** que registraste y despliegaste.
- Se genera un **vector de 768 dimensiones** (embedding).
- Ese vector se almacena en el campo `"sentence_embedding"` dentro del índice **kNN**.
- Esto permite que más adelante puedas buscar *semánticamente* documentos parecidos a una consulta, no solo por coincidencia exacta de palabras.

En resumen:

Este paso es como "llenar la base de conocimiento vectorial" con textos para que, cuando un usuario pregunte algo, el sistema pueda encontrar fragmentos relevantes y dárselos al modelo de lenguaje como contexto para que genere una respuesta más precisa.

14.- Probar la Búsqueda Neural

```
bash

curl --insecure -u '$OS_USERNAME:$OS_PASSWORD' \
-XGET "$OS_ENDPOINT/knn-faiss-avx2-index1/_search" \
-H 'Content-Type: application/json' -d'
{
  "_source": {
    "excludes": [
      "sentence_embedding"
    ]
  },
  "query": {
    "neural": {
      "sentence_embedding": {
        "query_text": "¿Qué es Ubuntu Pro?",
        "model_id": "l20H55EBiw8MvxrDrwcL",
        "k": 15
      }
    }
  }
}'
```

Explicación del comando curl

- **curl** → Herramienta de línea de comandos para hacer solicitudes HTTP.
- **--insecure** → Permite conexiones a servidores con certificados SSL no verificados (útil en entornos de prueba).
- **-u '$OS_USERNAME:$OS_PASSWORD'** → Envía credenciales básicas (*Basic Auth*) para autenticarse contra OpenSearch.
- **-XGET** → Especifica que se realizará una petición HTTP GET.
- **"$OS_ENDPOINT/knn-faiss-avx2-index1/_search"** → URL del endpoint de búsqueda en el índice knn-faiss-avx2-index1.
- **-H 'Content-Type: application/json'** → Indica que el cuerpo de la solicitud está en formato JSON.
- **-d' ... '** → Datos enviados en la solicitud (en este caso, la consulta en JSON).

Explicación del JSON de la solicitud

```
json
"_source": {
  "excludes": [
    "sentence_embedding"
  ]
}
```

- **_source** → Define qué campos devolver o excluir en la respuesta.
- **excludes** → Lista de campos a excluir; aquí se excluye "sentence_embedding" para no devolver los vectores de embeddings.

```
json
"query": {
  "neural": {
    "sentence_embedding": {
      "query_text": "¿Qué es Ubuntu Pro?",
      "model_id": "l20H55EBiw8MvxrDrwcL",
      "k": 15
    }
  }
}
```

- **query** → Bloque que contiene el tipo de consulta.
- **neural** → Indica que se usará búsqueda neuronal (vectorial).
- **sentence_embedding** → Campo del índice que contiene los vectores de las oraciones.
- **query_text** → Texto que se quiere buscar: "¿Qué es Ubuntu Pro?".
- **model_id** → ID del modelo de embeddings usado para generar los vectores de consulta.
- **k** → Número de resultados más similares que se deben devolver (top-k).

```
JSON
{
  "took": 1044,
  "timed_out": false,

  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },

  "hits": {
    "total": {
      "value": 5,
      "relation": "eq"
    },
    "max_score": 0.024580367,
    "hits": [ ... ]
  }
}
```

Campos principales

- **took**:
  Tiempo total que tomó ejecutar la consulta, en milisegundos.
  En este caso: 1044 ms (aprox. 1.044 segundos).
- **timed_out**:
  Indica si la consulta excedió el tiempo límite y se interrumpió (true) o si se completó a tiempo (false).
  Aquí es false, por lo que la búsqueda se completó correctamente.

Información sobre los *shards* (_shards)

- En OpenSearch, los índices están divididos en fragmentos (*shards*) para distribuir la carga.
- **total:** Número total de *shards* involucrados en la búsqueda (1 en este caso).
- **successful:** Cantidad de *shards* que respondieron exitosamente (1).
- **skipped:** *Shards* omitidos durante la búsqueda (0).
- **failed:** *Shards* que fallaron (0).

Resultados de la búsqueda (`hits`)

- **total:**
  - **value**: Número total de documentos que cumplen la búsqueda (5).
  - **relation**: Relación entre el valor mostrado y el real. Puede ser:
    - "eq" → Igual (el número es exacto).
    - "gte" → Mayor o igual (el número es un mínimo aproximado).
      En este caso: "eq" significa que son exactamente 5 resultados.
- **max_score:**
  Puntuación de relevancia más alta obtenida por un documento (0.024580367).
  Esta puntuación indica qué tan bien coincide el documento con la consulta, según el modelo vectorial.
- **hits:**
  Lista de los documentos encontrados, ordenados por relevancia.
  Aquí está representada como [....] porque el contenido se omite en el ejemplo.

15.- HabilitaR las direcciones privadas, añadir el endpoint del LLM al conector de confianza y crear un conector de modelo para tu LLM.

Consultar artículos sobre cómo desplegar tu propio LLM, por ejemplo Llama2, en Kubernetes usando Kserve y VLLM.

Para los propósitos de esta guía, asumimos que el LLM ya está alojado y es accesible mediante el endpoint de la API: `llama-2-13b-predictor.vllm.10.6.0.13.nip.io`.

```
bash
$ curl -u '$OS_USERNAME:$OS_PASSWORD' \
-XPUT "$OS_ENDPOINT/_cluster/settings" \
-H 'Content-Type: application/json' -d'
{
  "persistent": {
    "plugins.ml_commons.connector.private_ip_enabled": true,
    "plugins.ml_commons.trusted_connector_endpoints_regex": [
      "^http://.*[a-z0-9-]\\.10\\.6\\.0\\..*[a-z0-9-]\\.nip\\.io/.*$"
    ]
  }
}'
```

Explicación:

- Habilita el uso de **direcciones privadas** para conectores.
- Define un **patrón de endpoints confiables** usando expresión regular para que solo se acepten endpoints que cumplan con la estructura *.10.6.0.*.nip.io.

```
bash
$ curl -u '$OS_USERNAME:$OS_PASSWORD' \
-XPOST "$OS_ENDPOINT/_plugins/_ml/connectors/_create" \
-H 'Content-Type: application/json' -d'
{
  "name": "vLLM Connector",
  "description": "API vLLM compatible con OpenAI",
  "version": 2,
  "protocol": "http",
  "parameters": {
    "endpoint": "llama-2-13b-predictor.vllm.10.6.0.13.nip.io",
    "model": "TheBloke/Llama-2-13B-chat-GPTQ",
    "temperature": 0.03,
    "repetition_penalty": 1.13
  },
  "actions": [
    {
      "action_type": "predict",
      "method": "POST",
      "url": "http://${parameters.endpoint}/v1/chat/completions",
      "request_body": "{ \"model\": \"${parameters.model}\", \"messages\":
${parameters.prompt}, \"temperature\": ${parameters.temperature},
\"repetition_penalty\": ${parameters.repetition_penalty} }"
    }
  ]
}'
```

Explicación:

- **name**: Nombre del conector (vLLM Connector).
- **description**: Breve descripción del conector.
- **version**: Versión de la configuración del conector.
- **protocol**: Protocolo usado para conectarse al LLM (http).
- **parameters**: Parámetros del modelo LLM:
  - endpoint: Dirección del LLM.
  - model: Modelo específico que se utilizará.
  - temperature: Nivel de aleatoriedad en las respuestas (0.03 = muy determinista).
  - repetition_penalty: Penalización por repetición de tokens (1.13).
- **actions**: Define las acciones que el conector puede realizar, en este caso predict (predicción) enviando un POST al endpoint de chat.

Respuesta esperada

```
json
{"connector_id":"kG0A55EBiw8MvxrDgQeD"}
```

connector_id: Identificador único asignado al conector creado.

16.- Registrar el conector LLM.

```
bash

$ curl -u '$OS_USERNAME:$OS_PASSWORD' \
-XPOST "$OS_ENDPOINT/_plugins/_ml/models/_register" \
-H 'Content-Type: application/json' -d'
{
  "name": "Servicio de inferencia KServe: Llama-2-13B-chat",
  "function_name": "remote",
  "description": "Modelo de prueba RAG LLM",
  "connector_id": "kG0A55EBiw8MvxrDgQeD"
}'
```

Explicación:

- **name**: Nombre que se le asigna al modelo dentro de OpenSearch.
- **function_name**: Tipo de función; aquí remote indica que se usará un modelo externo vía conector.
- **description**: Breve descripción del modelo.
- **connector_id**: ID del conector que previamente creaste y que se usará para acceder al LLM.

Respuesta esperada:

```
json
{
 "task_id": "km0C55EBiw8MvxrDVgeF",
 "status": "CREATED",
 "model_id": "k20C55EBiw8MvxrDWQfl"
}
```

- **task_id**: Identificador de la tarea de registro del modelo.
- **status**: Estado de la operación (CREATED indica que se creó correctamente).
- **model_id**: Identificador único asignado al modelo registrado.

17.- Obtener el ID del modelo del conector LLM usando el ID de la tarea.

```
bash

$ curl -u '$OS_USERNAME:$OS_PASSWORD' \
-XGET "$OS_ENDPOINT/_plugins/_ml/tasks/km0C55EBiw8MvxrDVgeF"
```

Explicación de este comando

1.- curl → Herramienta de línea de comandos para realizar solicitudes HTTP.

2.- -u '$OS_USERNAME:$OS_PASSWORD' → Envía credenciales de usuario y contraseña para autenticarse en el servidor OpenSearch.

3.- -XGET → Indica que se hará una petición HTTP **GET** (obtener datos).

4.- **"$OS_ENDPOINT/_plugins/_ml/tasks/km0C55EBiw8MvxrDVgeF"** →

- **$OS_ENDPOINT**: Dirección base de tu clúster de OpenSearch.
- **_plugins/_ml/tasks/**: Ruta del plugin de Machine Learning de OpenSearch para consultar tareas.
- **km0C55EBiw8MvxrDVgeF**: Es el **ID de la tarea** que quieres consultar (en este caso, la tarea que registró el modelo LLM).

En resumen: Este comando consulta el estado y la información detallada de una tarea de Machine Learning (ML) en OpenSearch, usando el ID de tarea.

Es útil para:

- Saber si la tarea ya terminó o sigue en proceso.
- Ver el **model_id** asignado al modelo registrado.
- Ver en qué nodo se ejecutó y cuándo.

Respuesta

```
json

{
  "model_id": "k20C55EBiw8MvxrDWQfl",
  "task_type": "REGISTER_MODEL",
  "function_name": "REMOTE",
  "state": "COMPLETED",
  "worker_node": ["hy5M6dC9SqW2HR0QCRt5lw"],
  "create_time": 1726157575317,
  "last_update_time": 1726157577102,
  "is_async": false
}
```

Explicación de cada campo

- **model_id** → ID único del modelo registrado (k20C55EBiw8MvxrDWQfl).
- **task_type** → Tipo de tarea ejecutada (REGISTER_MODEL indica que era un registro de modelo).
- **function_name** → Tipo de función asociada al modelo (REMOTE significa que el modelo está alojado externamente y se accede por conector).
- **state** → Estado final de la tarea (COMPLETED = completada con éxito).
- **worker_node** → Lista de nodos de OpenSearch que ejecutaron la tarea.
- **create_time** → Marca de tiempo (timestamp) en milisegundos de creación de la tarea.
- **last_update_time** → Marca de tiempo de la última actualización de la tarea.
- **is_async** → Indica si la tarea fue asíncrona (false = fue síncrona).

18.- Desplegar el modelo del conector LLM.

```
bash
$ curl -u '$OS_USERNAME:$OS_PASSWORD' \
-XPOST "$OS_ENDPOINT/_plugins/_ml/models/k20C55EBiw8MvxrDWQfl/_deploy"
```

Explicación del comando

- **curl** → Herramienta para hacer solicitudes HTTP desde la terminal.
- **-u '$OS_USERNAME:$OS_PASSWORD'** → Autenticación básica con usuario y contraseña para OpenSearch.
- **-XPOST** → Indica que se envía una petición HTTP **POST** (para ejecutar una acción).
- **$OS_ENDPOINT/_plugins/_ml/models/k20C55EBiw8MvxrDWQfl/_deploy** →
  - **k20C55EBiw8MvxrDWQfl** es el **ID del modelo** obtenido en el registro anterior.
  - **/_deploy** es el endpoint del plugin de Machine Learning que ordena desplegar (activar) el modelo.

Respuesta

```
json
{
  "task_id": "lG0D55EBiw8MvxrDKQcd",
  "task_type": "DEPLOY_MODEL",
  "status": "COMPLETED"
}
```

Explicación de la respuesta JSON

- **task_id** → ID único de la tarea de despliegue.
- **task_type** → Tipo de tarea (DEPLOY_MODEL indica que se desplegó un modelo).
- **status** → Estado final de la tarea (COMPLETED significa que el despliegue se realizó con éxito).

19.- Crear un agente para el flujo conversacional.

```
bash
curl -u "$OS_USERNAME:$OS_PASSWORD" \
  -X POST "$OS_ENDPOINT/_plugins/_ml/agents/_register" \
  -H 'Content-Type: application/json' \

  -d '{
    "name": "test-agent-vector-mlmodel-rag",
    "type": "conversational_flow",
    "description": "Este es un agente de demostración para flujo conversacional",
    "app_type": "rag",
    "memory": { "type": "conversation_index" },
    "tools": [
      {
        "type": "VectorDBTool",
        "parameters": {
          "model_id": "l20H55EBiw8MvxrDrwcL",
          "index": "knn-faiss-avx2-index1",
          "embedding_field": "sentence_embedding",
          "source_field": ["text"],
          "input": "${parameters.question}",
          "doc_size": 10,
          "k": 30
        }
      },
      {
        "type": "MLModelTool",
        "description": "Una herramienta RAG LLM para responder preguntas",
        "parameters": {
          "model_id": "k20C55EBiw8MvxrDWQfl",
          "prompt": "[ { \"role\": \"Context:\", \"content\":
\"${parameters.VectorDBTool.output}\" }, { \"role\": \"Chat history:\", \"content\":
\"${parameters.chat_history:-}\" }, { \"role\": \"User instruction:\", \"content\":
\"${parameters.question}\" } ]"
        }
      }
    ]
  }'
```

Explicación del comando

1. Qué hace cada parte del curl

- **-u "$OS_USERNAME:$OS_PASSWORD"**
  Autenticación Basic. Al usar **comillas dobles**, Bash expande las variables. Con comillas simples no lo haría.
- **-X POST**
  Indica que crearás un **nuevo recurso** (registro del agente) en el plugin de ML.
- **"$OS_ENDPOINT/_plugins/_ml/agents/_register"**
  Ruta del **registro de agentes** del plugin de ML de OpenSearch.
  - "$OS_ENDPOINT" debe incluir esquema y host (p. ej. https://opensearch.mi-dominio.com).
- **-H 'Content-Type: application/json'**
  Cuerpo en JSON.
- **-d '…'**
  El **payload** JSON del agente. Se usa comilla simple exterior para evitar que el shell interprete el contenido (dentro ya están escapadas las comillas dobles necesarias para el JSON del prompt).

2. JSON del agente — campo por campo

Metadatos del agente

- **name**: identificador legible del agente.
- **type: "conversational_flow"**
  Indica que el agente orquesta **pasos** (herramientas) en un flujo conversacional.
- **description**: texto descriptivo.
- **app_type: "rag"**
  Señala que el patrón es **Retrieval-Augmented Generation**: primero recupera contexto, luego genera respuesta.

Memoria

- **memory.type: "conversation_index"**
  El agente puede **persistir/consultar** historial conversacional en un índice interno para dar continuidad al diálogo.

Herramientas (tools)

El agente encadena herramientas en orden:

a. **VectorDBTool** – Recuperación semántica
   - **parameters.model_id**: ID del **modelo de embeddings** usado para consultas vectoriales (consistente con el que indexaste).
   - **index**: nombre del índice vectorial (aquí, FAISS con AVX2).
   - **embedding_field**: campo del índice que contiene el vector (p. ej. "sentence_embedding").
   - **source_field**: campos textuales que quieres devolver como contexto (aquí solo "text").
   - **input: "${parameters.question}"**
     Placeholder: en tiempo de ejecución, el agente sustituye esto por la **pregunta del usuario**.
   - **doc_size** y **k**: hiperparámetros de recuperación.
     - **k**: cuántos **vecinos más cercanos** (candidatos) buscar.
     - **doc_size**: cuántos **documentos finales** incluir en el contexto (práctico para limitar el prompt). *(Según despliegue/implementación pueden matizarse; como regla práctica: empieza con k mayor que doc_size.)*

b. **MLModelTool** – Generación con LLM
   - description: etiqueta descriptiva.
   - parameters.model_id: ID del LLM desplegado (el que registraste y luego “deploy”).
   - parameters.prompt: plantilla de mensajes estilo Chat completions. Observa:
     - Se inyecta **contexto**: \"${parameters.VectorDBTool.output}\" (salida concatenada/estructurada de la recuperación).
     - Se incluye **historial**: \"${parameters.chat_history:-}\".
       El :- significa **valor por defecto vacío** si no hay historial.
     - Se pasa la **instrucción del usuario**: \"${parameters.question}\".
   - El agente formará un payload tipo OpenAI Chat (/v1/chat/completions) para el LLM con esos mensajes.

3. Flujo de ejecución del agente (resumen)

a. Llega una **pregunta** (parameters.question).
b. VectorDBTool busca **k** vecinos, selecciona **doc_size** y devuelve el **contexto**.
c. El agente **inyecta** ese contexto + historial + instrucción en el **prompt**.
d. MLModelTool llama al **LLM** con ese prompt y devuelve la **respuesta** final.

Pregunta e inicia el flujo conversacional con tu RAG.

1.- Finalmente, haz una pregunta para iniciar una nueva conversación.

```
Bash

curl -u "$OS_USERNAME:$OS_PASSWORD" \
  -XPOST "$OS_ENDPOINT/_plugins/_ml/agents/ym066JEBiw8MvxrDJwgR/_execute" \
  -H "Content-Type: application/json" \
  -d '{
    "parameters": {
      "question": "What is the best Linux distribution?"
    }
  }'
```

Explicación del comando Bash

- **curl -u "$OS_USERNAME:$OS_PASSWORD"**
  - -u envía usuario y contraseña (autenticación básica HTTP).
  - Aquí se usan variables de entorno ($OS_USERNAME y $OS_PASSWORD) para no exponer credenciales en texto plano.
- **-XPOST**
  - Especifica que la petición HTTP será **POST**, ya que estamos enviando datos.
- **"$OS_ENDPOINT/_plugins/_ml/agents/ym066JEBiw8MvxrDJwgR/_execute"**
  - Es la URL completa a la API de OpenSearch que ejecuta un **agente** ya registrado.
  - ym066JEBiw8MvxrDJwgR es el **ID del agente** que creaste en pasos anteriores.
- **-H "Content-Type: application/json"**
  - Indica que el cuerpo de la solicitud es JSON.
- **-d '{ "parameters": { "question": "What is the best Linux distribution?" } }'**
  - Cuerpo de la petición, que incluye el parámetro question con la pregunta que queremos hacer al agente.

Respuesta

```
json

{
  "inference_results": [
    {
      "output": [
        {
          "name": "memory_id",
          "result": "41miLZEBt5crDBEqzptK"
        },
        {
          "name": "parent_message_id",
          "result": "5FmiLZEBt5crDBEq0puo"
        },
        {
          "name": "MLModelTool",
          "result": "{\"id\": \"chatcmpl-9tdfAtRvy2ZIXI3v32vbofXFIoV1y\", \"object\":
\"chat.completion\", \"created\": 1723047532, \"model\": \"TheBloke/Llama-2-13B-chat-
GPTQ\", \"choices\": [{\"index\": 0, \"message\": {\"role\": \"assistant\", \"content\":
\"...\"}, \"finish_reason\": \"stop\"}], \"usage\": {\"prompt_tokens\": 361,
\"completion_tokens\": 92, \"total_tokens\": 453}}"
        }
      ]
    }
  ]
}
```

Campos importantes:

- **memory_id** → ID único para identificar esta conversación en la memoria del agente.
  - Si quieres continuar la charla, deberás enviarlo en la siguiente petición.
- **parent_message_id** → ID del mensaje anterior en la conversación.
  - Sirve para mantener el contexto del flujo de chat.
- **MLModelTool** → Contiene el resultado del modelo LLM.
  - Dentro de este string JSON hay:
    - **model** → Nombre del modelo usado (TheBloke/Llama-2-13B-chat-GPTQ).
    - **choices[0].message.content** → Aquí está la respuesta del modelo.
    - **usage** → Estadísticas de tokens usados (prompt, respuesta y total).

En resumen:

- El **bash** envía una pregunta a un agente RAG registrado en OpenSearch.
- El **JSON** devuelve la respuesta junto con los IDs para continuar la conversación y los datos técnicos del modelo que respondió.

2.- A continuación, usa el *memory id* para continuar la conversación.

Comando Bash (cURL)

```
Bash

$ curl -u $OS_USERNAME:$OS_PASSWORD \
-XPOST "$OS_ENDPOINT/_plugins/_ml/agents/ym066JEBiw8MvxrDJwgR/_execute" \
-H 'Content-Type: application/json' -d'
{
  "parameters": {
    "question": "Which one can be used for confidential VMs in Azure Cloud?",
    "memory_id": "41miLZEBt5crDBEqzptK",
    "message_history_limit": 3
  }
}'
```

Explicación del bash:

- **curl** → Herramienta para hacer solicitudes HTTP desde la terminal.
- **-u $OS_USERNAME:$OS_PASSWORD** → Autenticación básica usando usuario y contraseña almacenados en variables de entorno.
- **-XPOST** → Método HTTP usado: POST.
- **$OS_ENDPOINT/_plugins/_ml/agents/ym066JEBiw8MvxrDJwgR/_execute** → URL del endpoint del agente RAG creado antes, donde ym066JEBiw8MvxrDJwgR es el **ID del agente**.
- **-H 'Content-Type: application/json'** → Cabecera indicando que el cuerpo de la petición está en formato JSON.
- **-d '{ ... }'** → Cuerpo de la solicitud en JSON con los parámetros que se le envían al agente.

JSON enviado al agente

```
json
{
  "parameters": {
    "question": "Which one can be used for confidential VMs in Azure Cloud?",
    "memory_id": "41miLZEBt5crDBEqzptK",
    "message_history_limit": 3
  }
}
```

Explicación de la pregunta enviada al agente:

- **question** → Pregunta que quieres hacer al agente.
- **memory_id** → ID de memoria de conversación, obtenido de una interacción anterior. Permite que el agente recuerde el contexto y continúe el diálogo sin empezar desde cero.
- **message_history_limit** → Límite de mensajes anteriores que el agente debe usar como contexto (en este caso, solo los últimos 3).

Respuesta esperada (*inference_results*)

La respuesta del servidor, por ejemplo:

```
json

{
  "inference_results": [
    {
      "output": [
        { "name": "MLModelTool", "result": "Respuesta generada por el modelo" }
      ]
    }
  ]
}
```

Explicación de la respuesta esperada:

- **inference_results** → Contiene el resultado de la inferencia del agente.
- **output** → Lista de salidas generadas por las herramientas que el agente usó (VectorDB, LLM, etc.).
- **result** → Texto generado por el modelo, en este caso la respuesta a la pregunta sobre máquinas virtuales confidenciales en Azure.

Conclusión

Como hemos demostrado, RAG ofrece numerosos beneficios y se ha convertido en una técnica poderosa para mejorar las aplicaciones de IA generativa.

Las herramientas de código abierto, como OpenSearch y KServe, han sido fundamentales para impulsar la innovación dentro del ecosistema de RAG. Nuestra guía muestra su utilidad para habilitar RAG en tus herramientas de IA.

Elegir la herramienta correcta es esencial para optimizar tu sistema RAG, al igual que seleccionar la plataforma en la nube y el hardware adecuados.

Además, garantizar la confidencialidad y seguridad de los datos y de los modelos de aprendizaje automático es vital en los sistemas RAG.

La **IA confidencial** aborda este problema de la confidencialidad de datos y modelos de machine learning proporcionando un entorno de ejecución con raíz en hardware que abarca tanto la CPU como la GPU.

Este entorno mejora la protección de los datos y el código de IA en tiempo de ejecución, protegiéndolos contra el software del sistema con privilegios —como el hipervisor o el sistema operativo anfitrión— y contra operadores privilegiados en la nube.

Antes de lanzar tu proyecto —ya sea para una prueba de concepto, desarrollo, pruebas o producción completa— la planificación cuidadosa y la consideración de la infraestructura de IA son fundamentales.

Tomar decisiones informadas sobre el caso de uso general, las necesidades del negocio, las herramientas tecnológicas, los entornos en la nube y el hardware sentará las bases para una solución RAG exitosa y escalable.
En nuestro caso de uso, Azure ofrece un entorno sólido, mientras que aceleradores como Intel, AMD y NVIDIA ayudan a escalar tus servicios RAG para satisfacer distintos requisitos de seguridad, latencia, rendimiento y capacidad de cómputo.

Esta guía mostró cómo construir un sistema RAG usando herramientas de código abierto y los beneficios de desplegarlo en una máquina virtual confidencial.

Comprender estos conceptos fundamentales es clave para desarrollar soluciones adaptadas a tu caso de uso específico. Después de seguir esta guía, deberías ser capaz de implementar y configurar diversas herramientas de código abierto para crear un chatbot de IA.

RAG confidencial en Canonical

¿Qué es Canonical?

Canonical es la empresa británica de software que desarrolla y mantiene ***Ubuntu***, una de las distribuciones de Linux más populares en el mundo.

Además de Ubuntu, Canonical ofrece soluciones y servicios relacionados con ***infraestructura en la nube, IoT, seguridad y soporte empresarial***.

En el contexto de *Confidential RAG*, Canonical puede proporcionar entornos Linux (por ejemplo, Ubuntu Server) configurados para ***máquinas virtuales confidenciales*** en la nube, garantizando que los datos y modelos de IA estén protegidos durante su ejecución.

¿Qué es "Confidential RAG" en Canonical?

Aunque Canonical no utiliza explícitamente el término *"Confidential RAG"*, sus tecnologías de **confidential computing** sí aplican directamente al contexto de RAG (Retrieval-Augmented Generation), especialmente cuando se trata de proteger datos sensibles durante la consulta y generación. Aquí te explico cómo encaja:

1.- Computación confidencial con Ubuntu

Canonical impulsa el uso de **máquinas virtuales confidenciales** (CVMs) en la nube que aprovechan hardware especializado para proteger datos *en uso*:

- En Azure, se combinan **procesadores AMD EPYC con SEV-SNP** (para cifrar memoria en la CPU) con **GPUs NVIDIA H100**, extendiendo esa protección al procesamiento en la GPU Canonical+1.
- En entornos privados, Canonical también brinda soporte para **Intel TDX**, una tecnología de enclave confiable para proteger datos y código dentro del entorno virtual elvira.canonical.com.
- Canonical colabora activamente con la **Confidential Computing Consortium**, promoviendo estándares y herramientas open source en este espacio Canonical.

2.- ¿Cómo encaja todo esto con RAG?

"El **Confidential RAG**" representa la combinación de tecnologías RAG (búsqueda semántica + LLM) ejecutadas dentro de un entorno de **computación confidencial**, garantizando que:

- La **base de datos vectorial** y el **modelo de lenguaje** operen dentro de entornos protegidos como los CVMs.
- El **contexto y modelo** permanezcan cifrados o aislados incluso durante la ejecución, evitando accesos no autorizados desde el host o el hypervisor.

Aunque Canonical no use ese nombre formalmente, su infraestructura confidencial proporciona el entorno ideal para RAG seguro.

3.- Ventajas de ejecutar RAG en entornos confidenciales de Canonical

Beneficio	Detalles
Protección de datos en uso	SEV-SNP y TDX cifran memoria RAM y GPU mientras los procesos se ejecutan.
Alineación con RAG sensible	Perfecto para sectores regulados (sanitario, financiero) que manejan datos sensibles.
Infraestructura abierta	Ubuntu compatible con KServe, OpenSearch y frameworks RAG open source.
Confianza basada en hardware	Aislamiento reforzado y verificación criptográfica de integridad mediante attestation.

Conclusión

"Confidential RAG en Canonical" implica ejecutar pipelines RAG dentro de entornos protegidos de Ubuntu (tanto en CPU como GPU), ideal para workloads sensibles. Esto garantiza que tanto los **datos recuperados** como las **respuestas generadas por el LLM** estén protegidos durante todo el proceso.

La sinergia entre RAG + computing confidencial en Ubuntu hace posible construir sistemas de generación de texto robustos, seguros y habilitados para sectores donde la privacidad no es negociable.

Construya la arquitectura y aplicación RAG correctas con el taller de RAG de Canonical

Canonical ofrece un taller de 5 días diseñado para ayudarle a comenzar a construir sus sistemas RAG empresariales.
Al final del taller, tendrá una comprensión completa de la teoría, la arquitectura y las mejores prácticas de ***RAG*** y ***LLM***.

Juntos, desarrollaremos y desplegaremos soluciones adaptadas a sus necesidades específicas. Puede obtener más información sobre los objetivos y resultados del taller consultando nuestra hoja de datos del taller.

Descargue la hoja de datos aquí:
https://assets.ubuntu.com/v1/6714fdb2-Build%20an%20Optimised%20and%20Secure%20LLM%20with%20Retrieval%20Augmented%20Generation%20Data%20Sheet.pdf?_gl=1*13lg96r*_gcl_au*ODQxMDU3OTI1LjE3NTUyOTc2NzY.

Aprenda y utilice herramientas RAG de primera clase en cualquier hardware y nube

Aproveche los beneficios de RAG con herramientas de código abierto diseñadas para todo su ciclo de vida de datos y aprendizaje automático. Ejecute modelos de lenguaje grande habilitados para RAG en cualquier hardware y plataforma en la nube, ya sea en producción o a gran escala.

Canonical ofrece infraestructura de IA lista para empresas junto con herramientas de datos e IA de código abierto para ayudarle a iniciar sus proyectos RAG. https://canonical.com/solutions/ai/genai https://canonical.com/solutions/ai

Asegure su pila de IA con confianza

Mejore la seguridad de sus proyectos de GenAI mientras domina las mejores prácticas para gestionar su pila de software. Descubra formas de proteger su código, datos y modelos de aprendizaje automático en producción con Confidential AI. https://ubuntu.com/confidential-computing/ai