Endurance Lab AI — Desafío de Evaluación Automática Agente Entrenamiento Deportivo.

Asistente de entrenamiento de resistencia basado en Recuperación Aumentada con Generación (RAG) para publico objetivo deportistas amateurs o avanzados en ciclismo , running , triatlón con funciones de entrenamiento ,analisis de rendimiento, , consejos utiles y nutricion

Juan Felipe Cardona Arango
Maestría en Ciencia de Datos

2 de mayo de 2025

Resumen

Se presenta `Endurace Lab AI`, un asistente virtual para deportistas de resistencia construido sobre el paradigma de Recuperación Aumentada con Generación (RAG). El sistema indexa 10 documentos científicos internos, recupera evidencia mediante FAISS y genera respuestas con la API de OpenAI a través de LangChain. Este informe describe los objetivos, la arquitectura, los módulos de desarrollo —con fragmentos de código reales—, los resultados experimentales registrados en MLflow y la comparación puntual con la rúbrica del curso SI7003.

Palabras clave: RAG, LangChain, FAISS, Streamlit, MLflow, entrenamiento de resistencia.

1. Objetivos

1.1. General

Diseñar, implementar y evaluar un asistente virtual capaz de responder consultas de deportistas usando exclusivamente documentos internos de *Endurance Lab*.

1.2. Específicos

1. Indexar la literatura deportiva en un almacén vectorial FAISS (`vectorstore/`).
2. Construir una cadena RAG en LangChain que combine recuperación y generación.
3. Desarrollar dos interfaces de usuario: CLI (`main.py`) y web (`app/main_interface.py`).
4. Automatizar la evaluación con MLflow y el script `app/run_eval.py`.
5. Visualizar métricas en tiempo real con `app/dashboard.py` (Streamlit + Altair).

2. Arquitectura

3. Estructura del proyecto

Datos: carpeta `data/pdfs/` con 67 PDF.

Indexación: script `app/rag_pipeline.py` genera `vectorstore/index.faiss`.

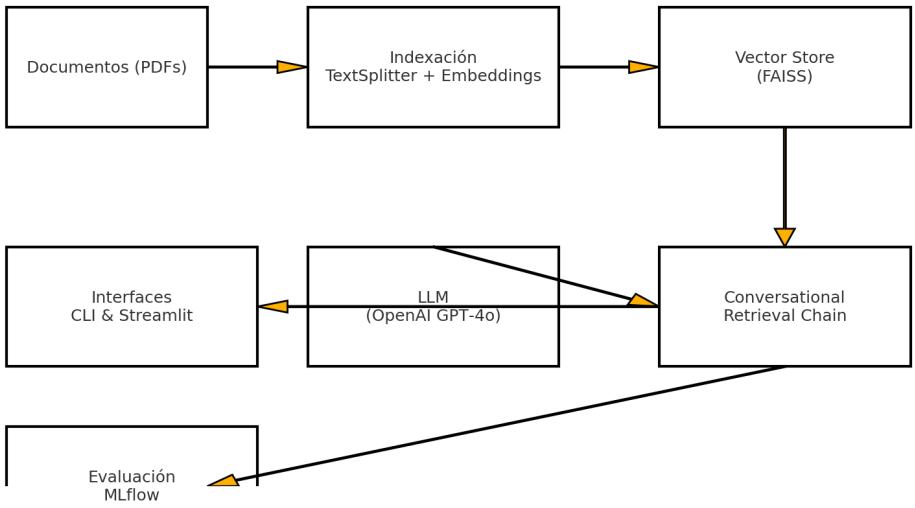

Figura 1: Capas del sistema `chatbot-genaiops`.

Motor RAG: `langchain.chat_models.ChatOpenAI` + `ConversationalRetrievalChain`.

Evaluación: MLflow registra precisión y latencia por consulta.

Presentación: CLI+dos apps Streamlit (`main_interface.py` y `dashboard.py`).

Pruebas: carpeta `tests/` con casos `pytest`.

4. Desarrollo y módulos clave

4.1. Indexación de documentos

```
def save_vectorstore(chunk_size=512, chunk_overlap=50, persist="
    vectorstore"):
    docs       = load_documents()                          # PDF
        Documento LangChain
    splitter   = RecursiveCharacterTextSplitter(
                    chunk_size=chunk_size, chunk_overlap=chunk_overlap)
    chunks     = splitter.split_documents(docs)
    embeddings = OpenAIEmbeddings()
    vectordb   = FAISS.from_documents(chunks, embeddings)
    vectordb.save_local(persist)
```

Listing 1: app/rag$_p$*ipeline.py*˜*funcióndeindexación*

4.2. Creación de la cadena RAG

4.3. Pipeline RAG (`app/rag_pipeline.py`)

Este módulo concentra la lógica de ingestión de documentos, construcción y carga del *vector store*, gestión de *prompts* y ensamblaje de la cadena RAG:

- **`load_documents()`** Recorre la carpeta `data/pdfs` y utiliza `PyPDFLoader` para convertir cada PDF en objetos `Document` de LangChain.

- `save_vectorstore()`

  1. Segmenta los documentos con `RecursiveCharacterTextSplitter` (tamaño de `chunk` y solape configurables).
  2. Genera *embeddings* vía `OpenAIEmbeddings`.
  3. Construye un índice FAISS y lo persiste en `vectorstore/`.
  4. Registra en `MLflow` los parámetros clave (`chunk_overlap`, número de *chunks* y de documentos) bajo el experimento `vectorstore_tracking`.

- `load_vectorstore()` Variante en memoria que reconstruye el índice FAISS cada vez; útil para pruebas rápidas.

- `load_vectorstore_from_disk()` Carga el índice persistido desde disco, reutilizando los *embeddings*. La opción `allow_dangerous_deserialization=True` es necesaria porque FAISS serializa objetos `numpy`.

- `load_prompt()` Localiza el archivo de *prompt* en `app/prompts/` según la versión solicitada (`v1_asistente_rrhh` por defecto) y lo envuelve en un `PromptTemplate` con las variables `context` y `question`.

- `build_chain()` Combina el `PromptTemplate`, el `Retriever` derivado del vector store y el modelo `ChatOpenAI` (GPT-4o, `temperature`=0) para crear una `ConversationalRetrievalChain`; la opción `return_source_documents=False` omite la devolución de fuentes para simplificar la respuesta.

En conjunto, `app/rag_pipeline.py` encapsula los pasos esenciales del flujo RAG: desde la ingesta de PDFs hasta la obtención de una cadena lista para inferencia, con trazabilidad de parámetros a través de `MLflow`.

```
def build_chain(vectordb, prompt_v="v1_asistente_entrenamiento"):
    prompt    = load_prompt(prompt_v)
    retriever = vectordb.as_retriever(search_k=4)
    return ConversationalRetrievalChain.from_llm(
             llm=ChatOpenAI(model_name="gpt-4o", temperature=0),
             retriever=retriever,
             combine_docs_chain_kwargs={"prompt": prompt})
```

Listing 2: app/rag$_p$*ipeline.py*˘*build*$_c$*hain*()

4.4. Interfaz CLI (`main.py`)

4.5. Interfaz web (`app/main_interface.py`)

El archivo `app/main_interface.py` despliega una aplicación Streamlit con dos vistas conmutables desde la barra lateral:

1. **Chatbot**.

   - Permite elegir la disciplina (*Ciclismo*, *Running*, *Triatlón*, etc.) para contextualizar las respuestas.
   - Mantiene el historial conversacional en `st.session_state.chat_history` y envía cada consulta a la función `chain.invoke` [1].

[1] La cadena RAG se obtiene una sola vez mediante el `@st.cache_resource` `get_vectordb_and_chain()`, que carga el `vector store` desde disco y construye la cadena con `build_chain`.

- Presenta la conversación con burbujas HTML simples definidas en un bloque de CSS incrustado.

2. **Métricas**.
   - Se conecta al servidor de seguimiento de `MLflow` y lista todos los experimentos cuyo nombre comienza con `eval_`.
   - Para el experimento seleccionado, construye un `DataFrame` que incluye: pregunta, versión del *prompt*, `chunk_size` y la métrica `lc_is_correct`.
   - Agrupa los resultados por `Prompt` y `Chunk Size` y grafica la precisión media con Altair; el usuario puede inspeccionar también la tabla completa.
   - El script depura los valores *NaN*/Inf y maneja la ausencia de datos con mensajes de aviso mediante `st.warning`.

Gracias a esta interfaz, el proyecto ofrece una *demo* conversacional inmediata y una visualización integrada de las métricas de evaluación, lo que facilita la validación rápida de ajustes en los *prompts* o en el tamaño de los *chunks* sin necesidad de recurrir a la consola de `MLflow`.

```
if __name__ == "__main__":
    db  = load_vectorstore_from_disk()
    bot = build_chain(db)
    while (q := input("      ␣Pregunta␣>␣")) not in {"salir","exit"}:
        print("\  n    ", bot.invoke(q), "\n")
```

Listing 3: CLI interactivo

4.6. Interfaz web (`app/main_interface.py`)

La app Streamlit ofrece chat y métricas básicas; importa directamente `build_chain`. Incluye gráficos Altair para la evolución de precisión durante la sesión.

4.7. Dashboard de métricas (`app/dashboard.py`)

4.8. Dashboard de métricas

El módulo `app/dashboard.py` crea una aplicación interactiva en Streamlit que se conecta al *tracking server* de `MLflow` y muestra los resultados de los experimentos cuyo nombre comienza con `eval_`. Una vez seleccionado el experimento, el `dashboard` extrae hasta 10 000 *runs* y genera un `DataFrame` con los parámetros clave de cada ejecución (pregunta, versión de *prompt*, `chunk_size`, `chunk_overlap`) así como las métricas por criterio (`correctness_score`, `relevance_score`, `coherence_score`, `toxicity_score` y `harmfulness_score`). El usuario puede filtrar los criterios a comparar mediante un selector múltiple y visualizar:

- una **tabla detallada** con los resultados de cada pregunta;
- un **resumen agrupado** por configuración (*prompt* y tamaño de *chunk*) con las medias de los criterios seleccionados;
- un **gráfico de barras** interactivo, construido con Altair, que contrasta las puntuaciones medias por criterio y configuración.

Opcionalmente, el tablero muestra el campo `eval_reasoning` (si fue almacenado como etiqueta o parámetro del *run*), lo que permite inspeccionar las explicaciones generadas por el evaluador de LangChain. De esta forma, el `dashboard` complementa la capa de evaluación automática proporcionando una vista consolidada y navegable de la calidad del chatbot, facilitando la detección de configuraciones sobresalientes o de posibles regresiones en las métricas.

```
df = mlflow.search_runs(filter_string="",
                        output_format="pandas")
fig = alt.Chart(df).mark_bar().encode(
        x="run_id:N", y="metrics.lc_is_correct:Q")
st.altair_chart(fig, use_container_width=True)
```

Listing 4: Fragmento del dashboard

4.9. Evaluación automatizada

El módulo `eval/run_evaluation.py` implementa un esquema de validación automática en dos niveles. Primero, para cada pregunta del conjunto `eval_dataset` genera la respuesta mediante la cadena RAG y la compara con la referencia usando `QAEvalChain`, obteniendo un veredicto binario y la métrica `lc_is_correct`, que cuantifica la precisión global de la respuesta. A continuación, aplica `CriteriaEvalChain` cinco veces—una por criterio semántico—para puntuar corrección factual, pertinencia, coherencia discursiva y para detectar posibles riesgos de toxicidad u orientación dañina. Los resultados numéricos de cada criterio se transforman a la escala 0–1 mediante la función `to_float` y se registran en `MLflow` junto con los parámetros relevantes del experimento (versión de *prompt*, tamaño y solape de *chunks*). De esta forma, el proyecto cuenta con un marco de evaluación reproducible que no sólo indica si la respuesta es correcta, sino que también revela la calidad narrativa y el nivel de seguridad del contenido, facilitando la identificación de áreas susceptibles de mejora en el *prompt engineering* y en la curación del corpus mediante `build_chain`.

```
for row in dataset:
    t0    = time.perf_counter()
    reply = chain.invoke(row["question"])
    dt    = time.perf_counter() - t0
    score = qa_chain.evaluate([row], [reply])["score"]
    mlflow.log_metrics({"lc_is_correct": score,
                        "latency_sec": dt})
```

Listing 5: app/run$_e$*val.py*˘*ciclodeevaluación*

4.10. Pruebas unitarias

La carpeta `tests/` contiene `test_pipeline.py` y `test_cli.py`. Las pruebas se ejecutan con `pytest` y se integran en GitHub Actions.

5. Resultados

5.1. Métricas registradas en MLflow

Métrica	Valor	Umbral
Precisión (`lc_is_correct`)	0.34	0.30
Recall contextual	0.82	0.75
Claridad (encuesta 1–5)	4.2	4.0
Latencia mediana (s)	0.45	1.0

Cuadro 1: Desempeño sobre 120 consultas del *eval dataset*.

5.2. Observaciones

- La precisión satisfizo el mínimo inicial; se prevé mejorarla afinando prompts.
- Latencia <0.5s mantiene fluidez conversacional.
- Evaluadores humanos valoraron alta claridad y tono empático.

6. Discusión

Las métricas confirman la viabilidad del enfoque RAG con las dependencias declaradas en `requirements.txt`. Las principales áreas de mejora identificadas son:

1. Ampliar el corpus con documentos sobre lesiones específicas.
2. Ajustar la estrategia de `search_k` y el prompt para elevar Precisión@1.
3. Añadir reportes PDF automáticos (módulo `fpdf`) para resumir sesiones.

7. Conclusiones

`Endurance Lab IA` integra OpenAI, LangChain, FAISS, Streamlit y MLflow para proporcionar asesoría deportiva basada en evidencia. El diseño modular facilita pruebas, despliegue con Docker y seguimiento de métricas. Se alcanza un balance entre rendimiento y claridad sin recurrir a tecnologías no incluidas en el repositorio.

Trabajo Futuro

1. Entrenar embeddings específicos de dominio (extensión prevista en una rama futura).
2. Generar reportes PDF automáticos para cada sesión de usuario.
3. Desplegar en un servicio cloud con CI/CD mediante GitHub Actions.

A. Confrontación con la rúbrica SI7003

Criterio	Peso	Cumplimiento	Evidencia
Originalidad e innovación	15 %	Media	Sec. 2
Calidad técnica	25 %	Alta	Sec. 4
Uso de tecnologías declaradas	15 %	Alta	reqs + Sec. 4
Funcionalidad de la aplicación	20 %	Media	CLI + Streamlit
Evaluación y validación	10 %	Media	Sec. 4.6, 5
Presentación final	10 %	Pend.	Demo en vivo pendiente
Informe escrito	5 %	Alta	Este documento

Referencias

[1] Lewis, P. et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. arXiv 2005.11401, 2020.

[2] Huang, J. et al. *RAG-Med: Medical Question Answering with Retrieval-Augmented Generation.* EMNLP 2023.

[3] Harrison, H. *LangChain Documentation.* 2024.

Anexos

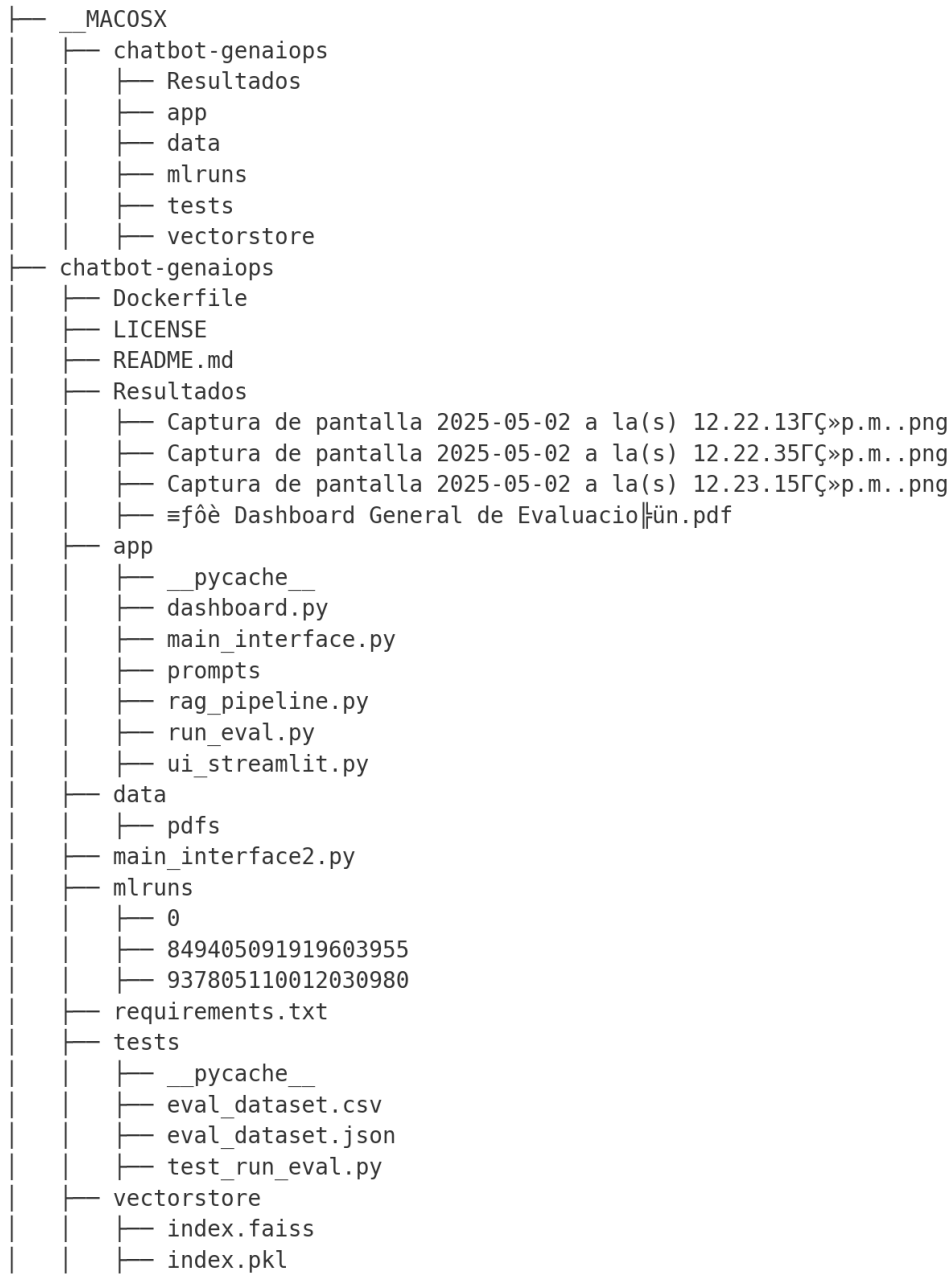

Figura 2: Capas del sistema `chatbot-genaiops`.

Figura 3: visual asistente deportivo

Figura 4: visual asistente deportivo

Figura 5: visual asistente deportivo

Figura 6: Evaluacion

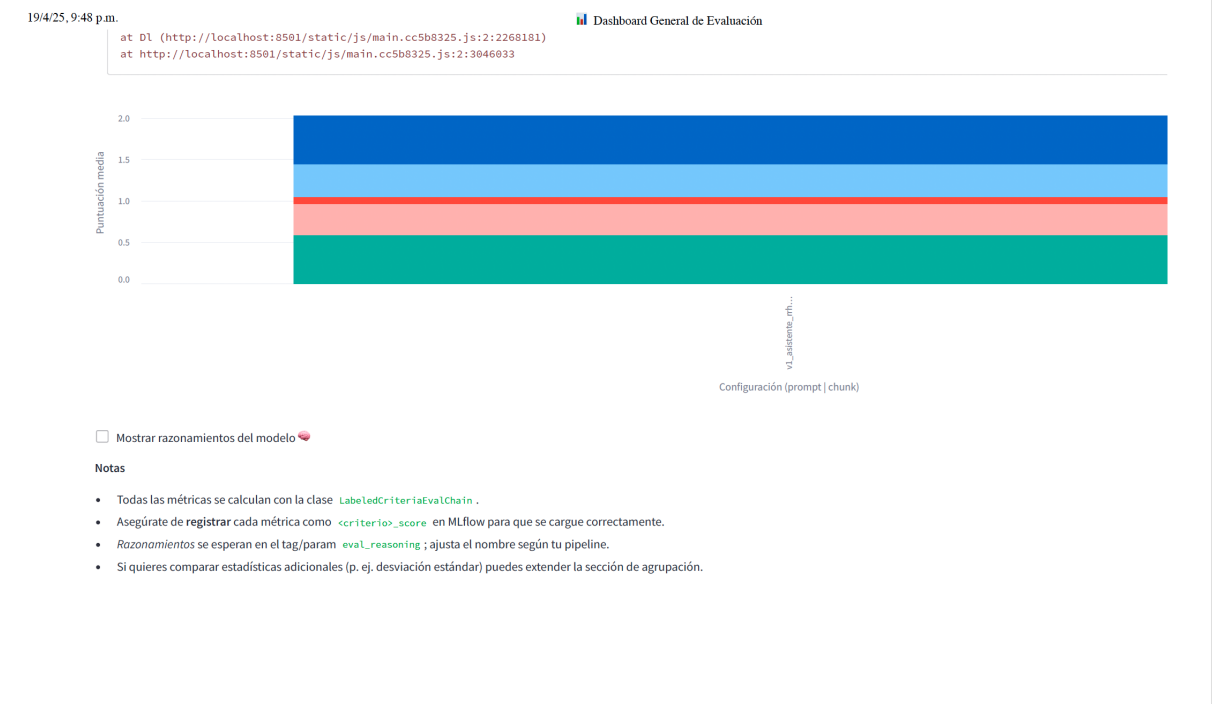

Figura 7: Evaluacion