Integrantes : Bryan Piña

Juan Castro

SE DESARROLLO USANDO LA VERSION DE PYTHON Python 3.11.9

Este es el informe general de desarrollo, donde iremos anotando nuestras experiencias al momento de desarrollar **LA EVALUACION**

Primero identificamos una empresa, en este caso usaremos **Everlast Chile** ya que conozco sobre cosas de deporte en general.

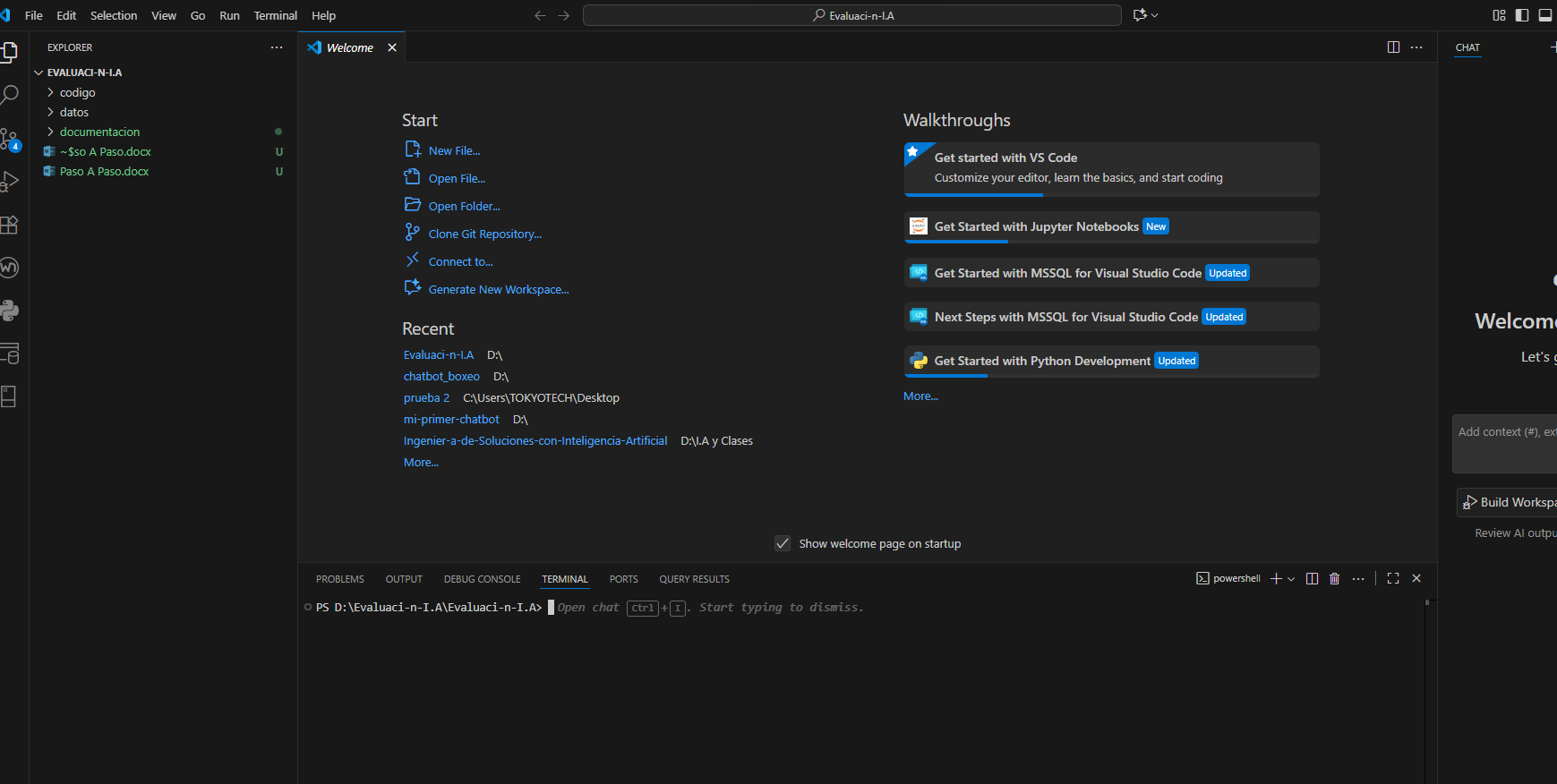
**Justificación:** Es una marca real, conocida y simple. Su catálogo de productos (guantes de boxeo, sacos, ropa) es específico y la información está públicamente disponible, y aunque no podamos acceder a ella, crearemos la información sobre los artículos para que la i.a la ocupe al momento de responder y no “alucine”, esto para un sistema RAG, ya que podemos simular que usamos su data interna.

Creamos 3 carpetas, en código todo el código de Python, en documentación, como el objetivo, la elección de la empresa y demás, en datos, pondremos archivos de texto con la información de la empresa, toda la información como de los artículos, las políticas y demás, para que nos sirva de simulación.

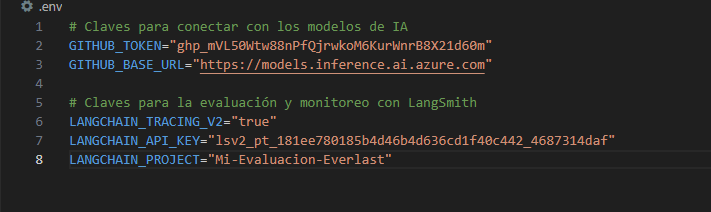
**AHORA COMENZAREMOS CON EL DESARROLLO DE CODIGO E IMPORTACIÓN DE HERRAMIENTAS**

**Partiremos creando el entorno virtual, y el archivo .env que nos sirve para mantener las llaves de acceso seguras.**

**Añadimos el archivo .env.**

****

****

****

**Adjuntamos nuestra key de github, y la api key de langsmith que nos servirá para el tema de las métricas.**

**Creamos el entorno virtual para que no se combinen las librerías y no haya problemas, asi podremos trabajar sin miedo y cómodamente.**

****

**Activamos el entorno.**

****

**Instalaremos las dependencias necesarias para poder trabajar, instalando las librerías necesarias para trabajar en este proyecto.**

****

**También instalaremos la de lansgmith que a decir verdad, no sé si se instaló con la anterior entrada de comandos, creo que no :**

****

**También instalamos pip install langchain-community.**

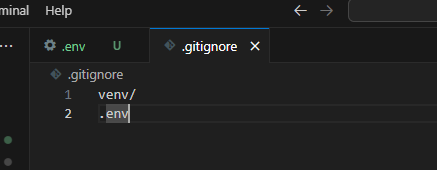
**Instalaremos nuevas dependencias para hacer las métricas con langchain A FUTURO:**

**pip install pandas plotly scikit-learn**

**resumen: pip install openai langchain langchain-community langchain-openai python-dotenv streamlit faiss-cpu pandas scikit-learn plotly langsmith**

**Hasta ahora con esta base, haremos nuestro primer commit.**

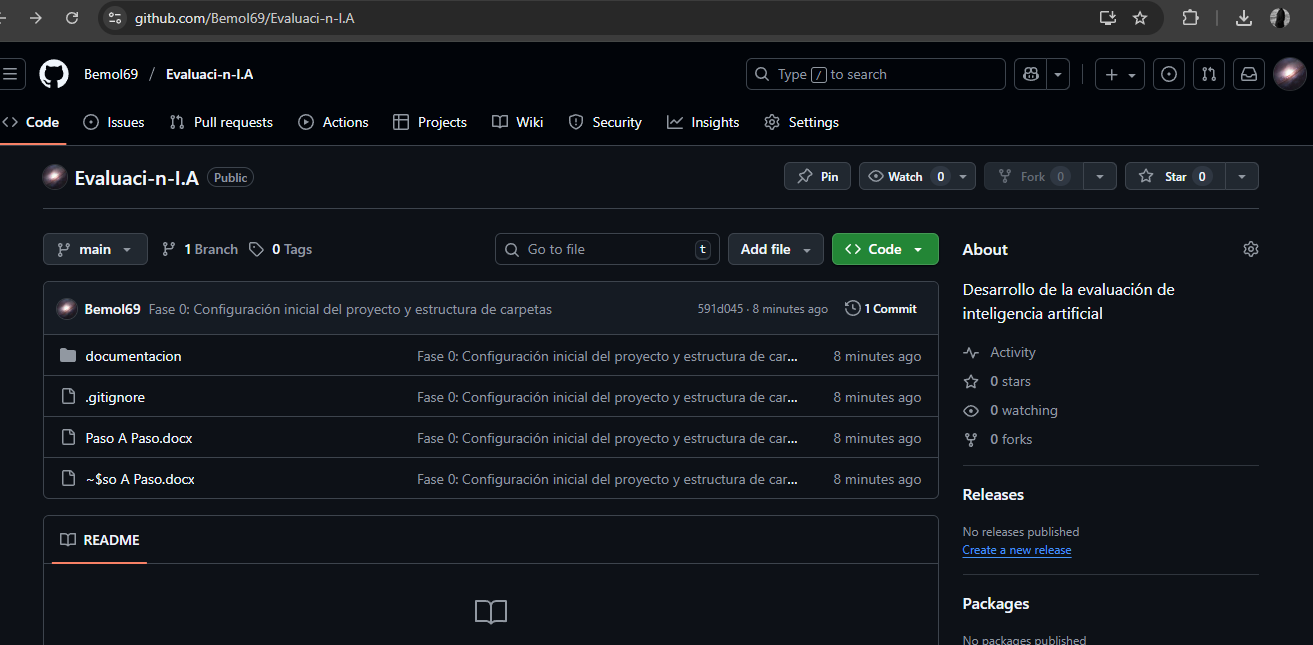
**Antes, como seguridad, crearemos un archivo .gitignore poniendo los archivos de venv y ,env de entorno para que no haga commit de eso, como regla de seguridad.**

****

**git add .**

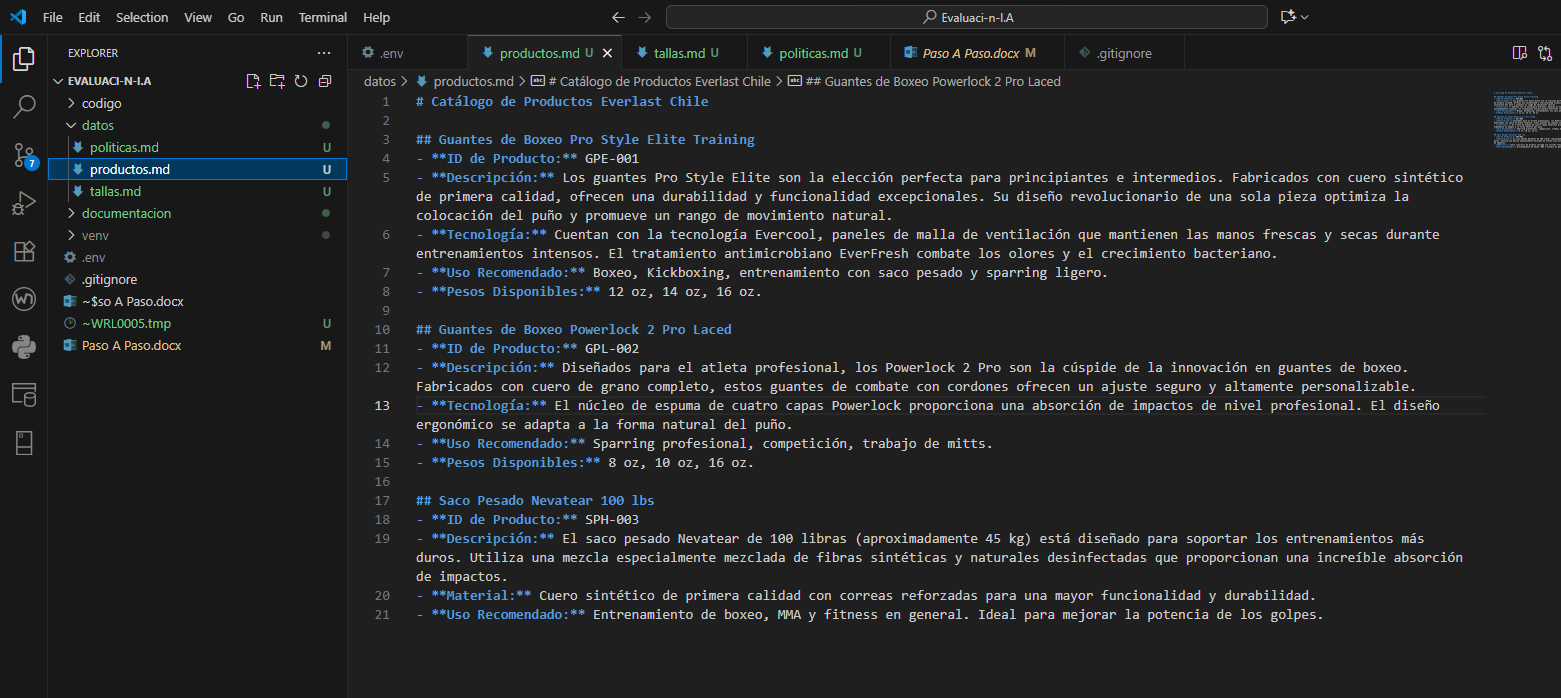
**git commit -m "Fase 0: Configuración inicial del proyecto y estructura de carpetas"**

**git push (a mi repositorio)**

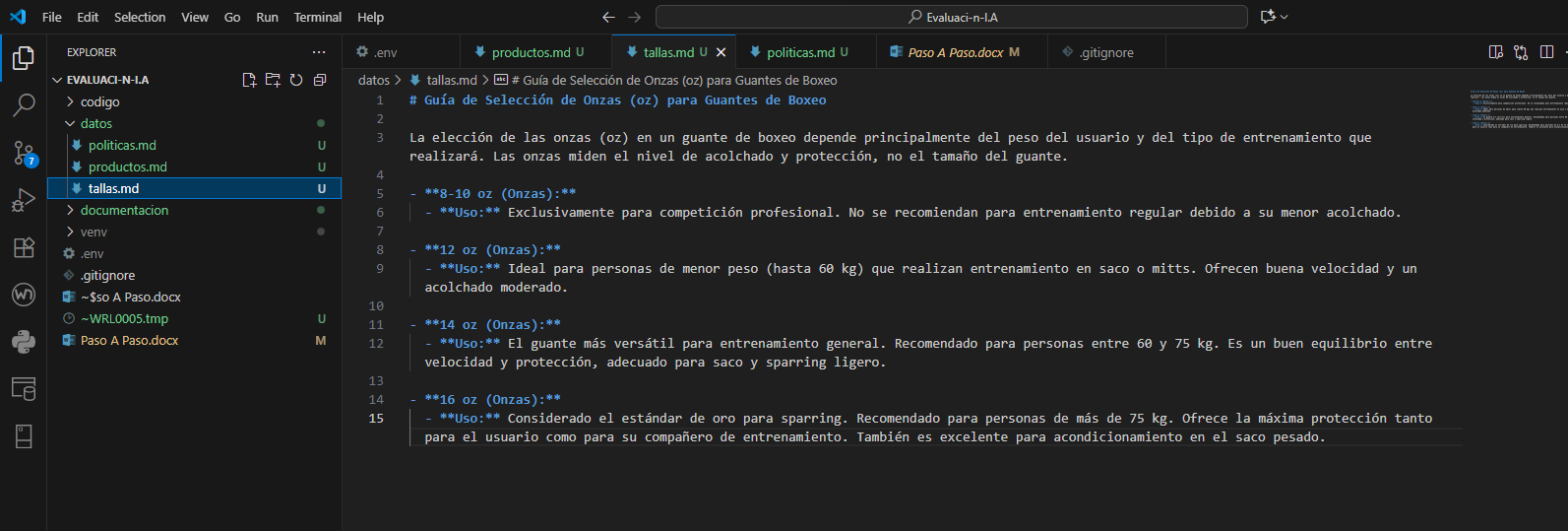
****

**Ahora lo siguiente que haremos es proporcionarle información a nuestra i.a, esto para que funcione el sistema RAG, crearemos la “base de conocimiento”, esto nos servirá para que nuestro asistente virtual de respuestas precisas y no alucine, partiremos creando 3 archivos dentro de la carpeta de datos, estos archivos serán:**

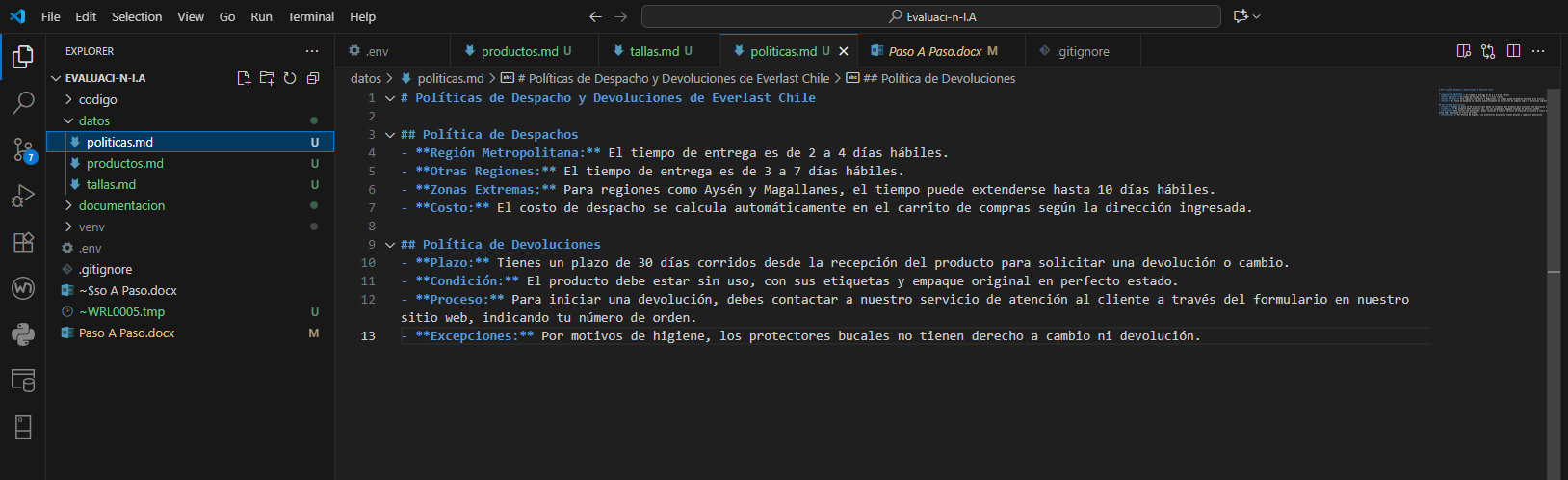
1. **productos.md (Este archivo contendrá la descripción de algunos productos clave de Everlast.)**

****

1. **tallas.md (Este documento ayudará al chatbot a responder una de las preguntas más frecuentes)**

****

1. **politicas.md (información de devoluciones y despachos.)**

****

**CREAREMOS OTRO COMMIT PARA SUBIR LA BASE DE CONOCIMIENTO A GITHUB**

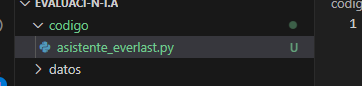
**Git add. Git commit, etc.**

**Ahora agregaremos la i.a con la que vamos a trabajar, en este caso ocuparemos el modelo gpt4o**

**Ocuparemos esta key de gpt4 : ghp\_acjSYV9PKW8GSwWkE7N02sq3zxxxxxxxxxxxxxxxxxxxxx**

**Comenzaremos a generar el código en Python para desarrollar el chatbot:**

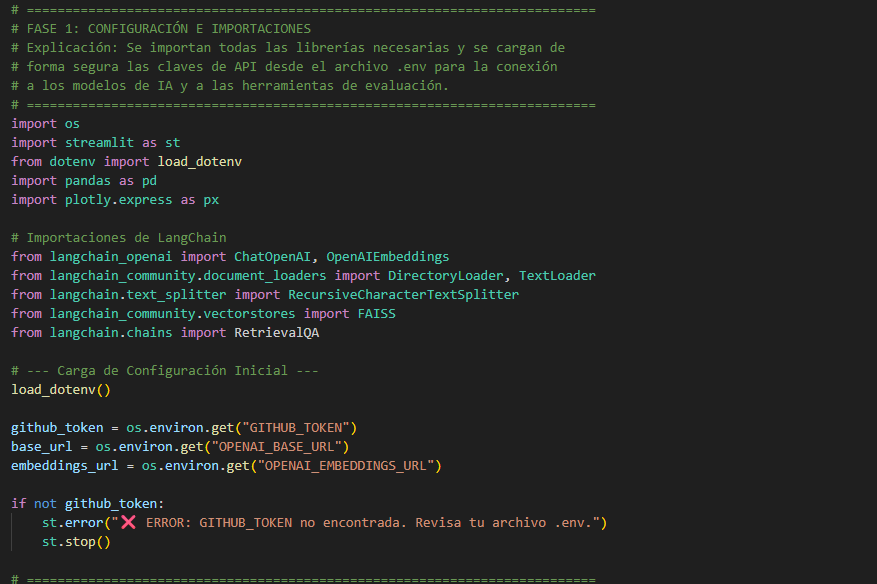
**Dentro de la carpeta código, crearemos el asistente asistente\_everlast.py**

****

**Luego ocuparemos un código y este fue dividido en fases para ser mas fácil de reconocer sus partes y de que sirven.**

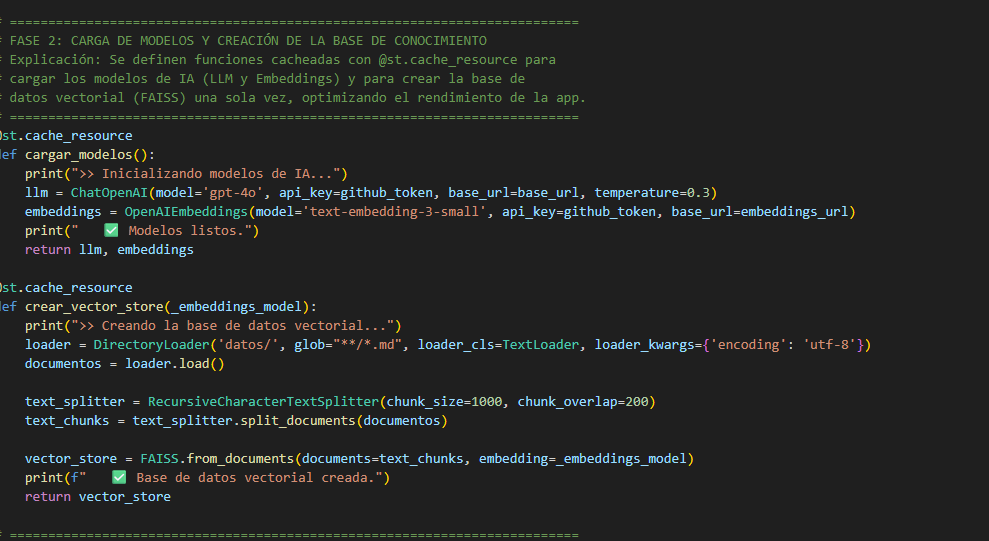
**Fase 1**

**El punto de partida de nuestra aplicación es la configuración del entorno y la importación de las librerías necesarias. En esta fase, utilizamos la librería os y dotenv para cargar de forma segura las claves de API desde un archivo .env, evitando exponerlas en el código fuente, lo cual es una práctica de seguridad fundamental. A continuación, importamos los componentes específicos de Streamlit para la interfaz de usuario y de LangChain, que forman el núcleo de nuestra arquitectura de IA, incluyendo los modelos de chat, embeddings, cargadores de documentos y la cadena de RAG (RetrievalQA)**

****

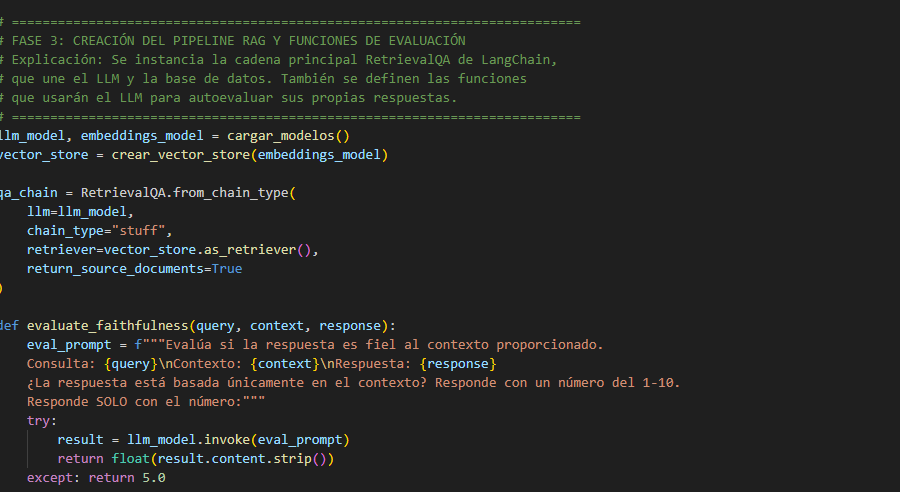
**Fase 2**

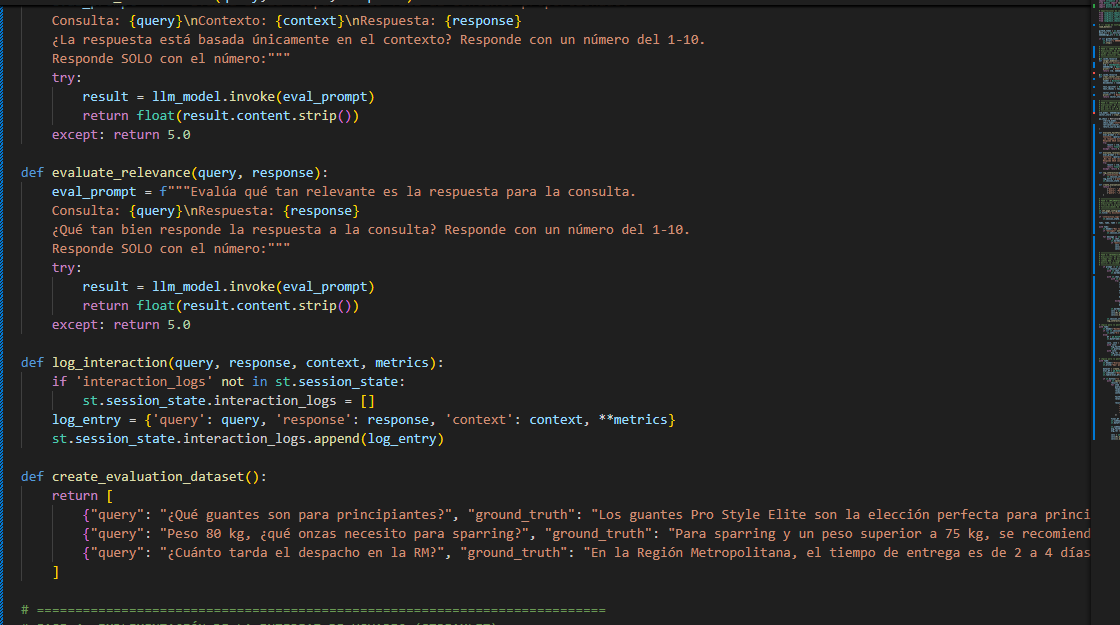
**Esta fase es crucial ya que preparamos los componentes pesados de la aplicación. Definimos dos funciones principales, cargar\_modelos y crear\_vector\_store, y utilizamos el decorador @st.cache\_resource de Streamlit. Este decorador es vital para la eficiencia, ya que asegura que los modelos de IA y la base de datos vectorial se carguen en memoria una sola vez al iniciar la aplicación, en lugar de recargarse con cada interacción del usuario. La primera función inicializa el LLM (gpt-4o) para la generación de respuestas y el modelo de Embeddings para la conversión de texto a vectores. La segunda función carga nuestros documentos (.md), los divide en chunks (trozos) y los vectoriza usando FAISS, creando así el 'cerebro' externo que nuestro chatbot consultará**

****

**Fase 3**

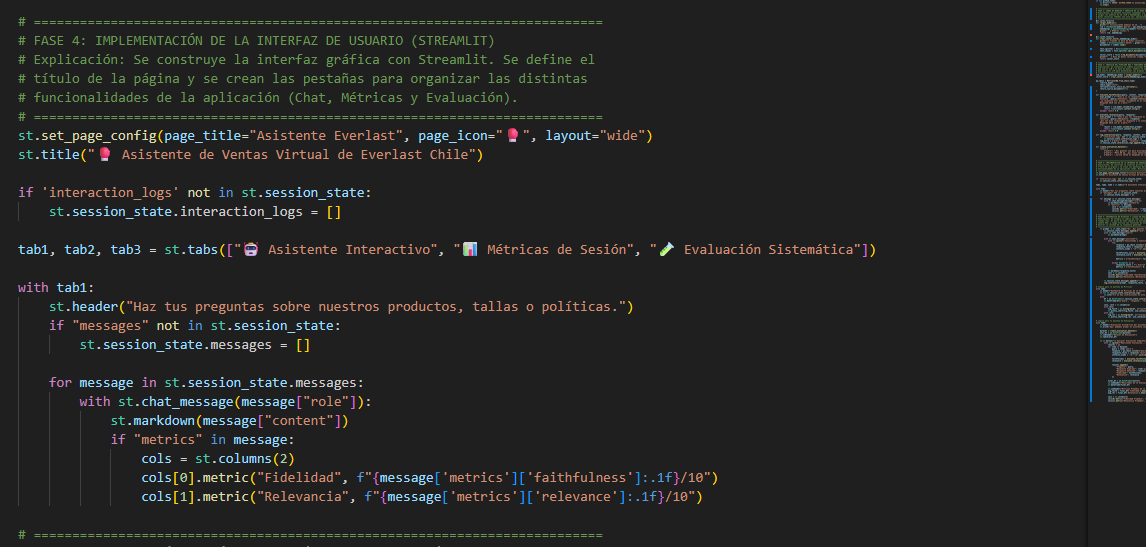
**En esta fase construimos el motor lógico de la aplicación. Primero, instanciamos los modelos y la base de datos vectorial llamando a las funciones cacheadas. Luego, creamos la cadena RetrievalQA de LangChain, que orquesta el flujo RAG completo. La configuramos con return\_source\_documents=True, un parámetro esencial que nos permite acceder a los chunks exactos que el sistema recuperó para generar una respuesta, lo cual es fundamental para la evaluación de la 'Fidelidad'. Adicionalmente, definimos las funciones de evaluación (evaluate\_faithfulness, evaluate\_relevance, etc.), que utilizan el propio LLM como un juez para calificar la calidad de las respuestas del chatbot en base a métricas predefinidas.**

****



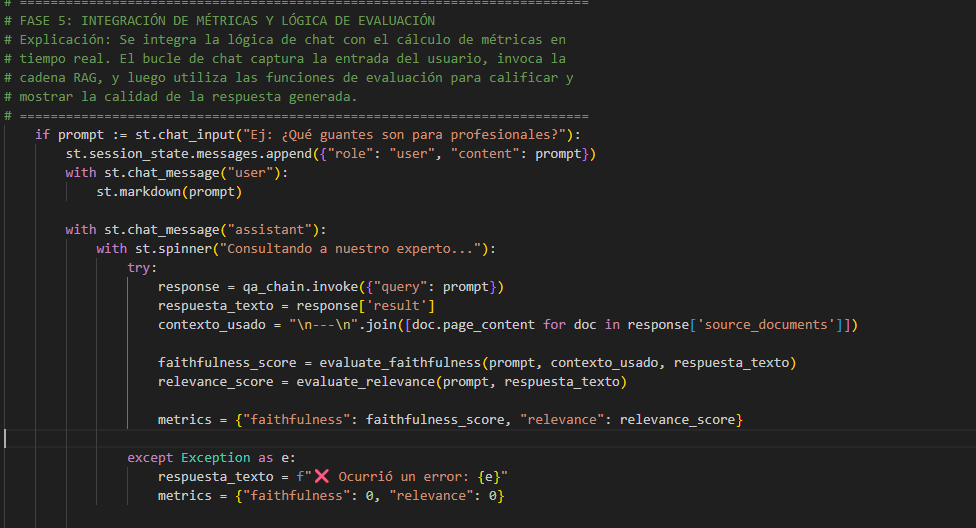
**Fase 4**

**Esta fase se centra en la experiencia del usuario (UX). Utilizamos los comandos de Streamlit para construir la interfaz gráfica. st.set\_page\_config y st.title definen la apariencia general de la página. El comando st.tabs es clave para organizar la aplicación en secciones lógicas: un chat interactivo, un dashboard de métricas y un panel de evaluación sistemática. La lógica del chat se maneja con st.session\_state para preservar el historial de la conversación, y los componentes st.chat\_message y st.chat\_input crean la familiar interfaz de chat que el usuario final utilizará para interactuar con el asistente**

****

**Y FASE 5 FINAL**

La fase final integra la lógica de evaluación directamente en la interfaz. Dentro del bucle del chat, tras invocar la cadena qa\_chain para obtener una respuesta, extraemos los documentos fuente utilizados. Luego, llamamos a nuestras funciones evaluate\_faithfulness y evaluate\_relevance para calificar la respuesta en tiempo real. Estos puntajes se muestran inmediatamente al usuario mediante st.metric, proporcionando una retroalimentación instantánea sobre la calidad del sistema. La interacción completa (pregunta, respuesta, contexto y métricas) se guarda en el log de la sesión. Finalmente, se implementa la lógica para las otras pestañas, que leen este log para generar gráficos con Plotly y ejecutar la evaluación sistemática sobre un dataset predefinido, completando así el ciclo de desarrollo y evaluación



Por ultimo añadiremos mas productos para el trabajo porque actualmente solo contamos con 3 productos con sus respectivos atributos, haremos un commit final

SE CORRE CON ESTE CODIGO

**streamlit run codigo/asistente\_everlast.py**