Historias de usuario

1. Carga de Documentos

- **Como usuario**, quiero subir documentos en formato PDF de máximo 5MB, para que la aplicación pueda analizarlos.
  Criterios:
  1. El usuario puede subir archivos en formato PDF de 5MB.
- **Como usuario,** quiero que antes de realizar preguntas al LLM en el chat, se exija que este subido un documento.
  Criterios:
  1. La aplicación bloquea la funcionalidad de preguntas si no hay un documento cargado.
  2. Se muestra un mensaje indicando que se debe subir un documento antes de realizar preguntas.
- **Como usuario**, quiero recibir un mensaje en el chat cuando mi documento se haya procesado correctamente, para saber cuándo puedo revisarlo.
  Criterios:
  1. Se muestra un mensaje en el chat cuando el documento ha sido procesado exitosamente.
  2. Si ocurre un error en el procesamiento, se notifica al usuario.
- **Como usuario,** quiero poder revisar un solo documento en cada chat que tenga.
  Criterios:
  1. Cada chat está asociado a un único documento.
  2. No es posible subir más de un documento por chat.

2. Procesamiento de Texto con IA

- **Como usuario**, quiero hacer preguntas sobre el contenido de un documento y recibir respuestas precisas, para aclarar dudas de manera eficiente.
  Criterios:
  1. El usuario puede realizar preguntas sobre el documento cargado.
  2. La IA responde utilizando la información del documento.

3. Interfaz Web y API

- **Como usuario**, quiero acceder a la aplicación desde una interfaz web, para gestionar mis chats.
  Criterios:
  1. La aplicación es accesible desde un navegador web.
  2. El usuario puede iniciar sesión y ver sus chats en la interfaz.

4. Gestión de Usuarios y Sesiones

- **Como usuario**, quiero registrarme e iniciar sesión en la aplicación, lo cual me permita acceder a los chats.
  Criterios:
  1. El usuario puede registrarse con un usuario y contraseña.
  2. El sistema valida las credenciales al iniciar sesión.
- **Como usuario**, quiero cerrar sesión en cualquier momento, para proteger mi privacidad.
  Criterios:
  1. Se proporciona un botón de "Cerrar sesión".
- **Como usuario**, quiero ver mis chats, para consultar información de diferentes documentos.
  Criterios:
  1. Se muestra una lista de chats activos en la interfaz.
  2. Cada chat al que le puedo preguntar está asociado a un documento.
- **Como usuario,** quiero poder crear tantos chats como yo considere necesarios.
  Criterios:
  1. No hay un límite en la cantidad de chats que el usuario puede crear.
- **Como usuario,** quiero poder consultar las respuesta y preguntas realizada de los chats mientras tenga abierta la sesión (sin persistencia).
  Criterios:
  1. Las preguntas y respuestas permanecen visibles mientras la sesión esté activa.
- **Como usuario,** quiero poder borrar los chats que considere que ya cumplieron su función de consulta.
  Criterios:
  1. Se proporciona una opción para eliminar chats individualmente.

5. Despliegue y Escalabilidad

- **Como administrador**, quiero que la aplicación se ejecute en contenedores con Docker y Docker Compose, para facilitar el despliegue y la escalabilidad.
  Criterios:
  1. La aplicación se ejecuta en un contenedores Docker.
  2. Se proporciona un archivo docker-compose.yml para la orquestación de los servicios.

Arquitectura de la Aplicación

La arquitectura de la aplicación está compuesta por 10 servicios, descritos a continuación:

1. **Frontend:** Es la capa de experiencia de usuario que gestiona la interacción del cliente con la interfaz gráfica. Permite acciones como iniciar sesión, registrarse y utilizar un chat interactivo, mediante el cual los usuarios pueden cargar documentos y realizar búsquedas específicas de información utilizando un modelo de inteligencia artificial.

2. **Embedding (1 y 2):** Esta capa genera los embeddings, convirtiendo el texto en vectores y llama a los servicios correspondientes una vez ya tenga los resultados convertidos en vectores.

    - **Embedding 1:** Produce embeddings de los chunks y los almacena en la base de datos vectorial.

    - **Embedding 2:** Convierte las preguntas en vectores y los envía al retriever para su procesamiento.

    - **Todos los vectores** son de un tamaño fijo, igual 384 posiciones.

3. **Chunking:** Este servicio divide el texto en chunks o fragmentos, estructurando la información antes de enviarla a Embedding 1 para generar embeddings y almacenarlos en la base de datos vectorial

4. **Base de datos relacional:** Asegura la persistencia de la información del cliente, como los datos de registro y sus chats asociados.

5. **Base de datos vectorial:** Se encarga de almacenar los documentos cargados por los usuarios, el ID del chat, los embeddings y el texto asociado a cada vector.

6. **Retriever:** Realiza una similitud de coseno entre el prompt (consulta del cliente) y los textos almacenados durante la sesión (un solo chat especifico), devolviendo los fragmentos del documento que mejor se ajusta a la pregunta formulada.

7. **Augment:** Estructura la consulta que se enviará al modelo, recibe la respuesta del modelo y la remite, junto con la consulta original, al servicio del backend directamente.

8. **Backend:** Actúa como orquestador, conectando el frontend con las capas de datos y procesos, y asegurando la comunicación entre los servicios.

9. **Ollama:** Procesa las búsquedas solicitadas por el cliente en los documentos y proporciona una respuesta que se envía al servicio de Augment.

Todos los servicios están desplegados dentro de una misma red denominada "project_network". Lo cual refuerza la seguridad, ya que la comunicación queda limitada a los microservicios que están dentro de la red.

La intención de utilizar microservicios es aislar la lógica de negocio, lo que permitirá, en futuras implementaciones en la nube, monitorear los recursos de la aplicación e identificar con precisión dónde está la falla en caso de que se presente. Por otro lado, también es importante tener en cuenta escalabilidad a la hora de desplegar los microservicios, es por ello que las políticas de auto-escalado deben ser definidas para cada uno de los microservicios, según la posible carga de trabajo que vaya a soportar. Lo anterior, garantiza que cada componente de la aplicación cuente con la capacidad adecuada para satisfacer la demanda.

Con base en esto, se tomaron decisiones como la gestión de los embeddings, donde, aunque el servicio sea el mismo, se implementó en dos microservicios distintos. Esto evita problemas de escalabilidad entre los dos flujos (uno de la carga de documentos y el otro de las preguntas sobre el mismo), permitiendo ajustarlos de manera independiente según la demanda.

El flujo de la aplicación es el siguiente: un usuario inicia sesión en el frontend, abre un chat y, antes de poder realizar preguntas, debe subir un documento. Una vez cargado, el documento es dividido en fragmentos (chunking), procesado para generar sus embeddings y almacenado en la base de datos junto con el ID del chat. Finalmente, el usuario puede comenzar a hacer consultas.

Por otro lado, cuando el usuario realiza una consulta, la pregunta se envía al backend, donde es procesada y vectorizada mediante el servicio de embeddings. Una vez generado el vector, este se utiliza para buscar información relevante en el retriever, identificando los chunks mas relevantes con respecto a la pregunta. Luego, el contexto recuperado y la pregunta se combinan en el módulo de augmentation, que los envía al modelo de lenguaje (LLM). El LLM genera una respuesta, la cual es devuelta al módulo de augmentation y finalmente con esto se tiene la respuesta para el front end.

Hablando sobre el diagrama de despliegue, la mayoría de los microservicios están asociados a un volumen. Esto permite la persistencia de logs y registros de la aplicación, asegurando que la información se mantenga disponible incluso si el contenedor deja de ejecutarse.

Como se mencionó anteriormente, para garantizar la escalabilidad del servicio, es necesario definir políticas de autoescalado. Estas políticas son estándar y preliminares, sirviendo como un punto de partida para el despliegue. Sin embargo, aún no se han realizado pruebas de carga, por lo que podrán ajustarse según los resultados obtenidos.

Politicas de autoescalado :

- Escalabilidad horizontal por sobrepasar un umbral del 80% en la CPU
- Escalabilidad horizontal por sobrepasar un umbral del 80% en memoria

- Los recursos de los microservicios no concernientes al modelo llm deben tener 512M de ram y 1024M de CPU
- Los recursos del micro deben ser de 512M de ram y 1024M de GPU para el microservicio del modelo ollama

Limitaciones de la aplicación:

- Los chat soportan más de un documento, pero está actualmente limitado a uno solo en la interfaz web.
- El tipo de documento debe ser PDF de máximo 5 MB.
- El modelo puede alucinar con las respuestas que regresa.
- No se persiste el historial de chats una vez se cierra la interfaz web, se cierra o expira la sesión.

A continuación, se ilustra los diagramas de arquitectura y despliegue planteados para esta solución:

Diagrama de Arquitectura

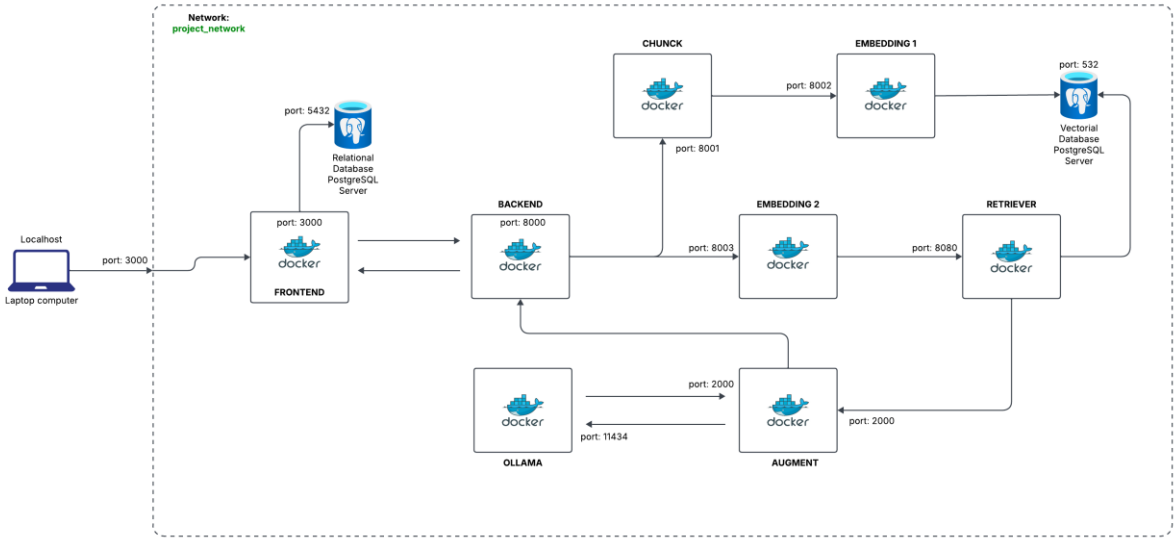
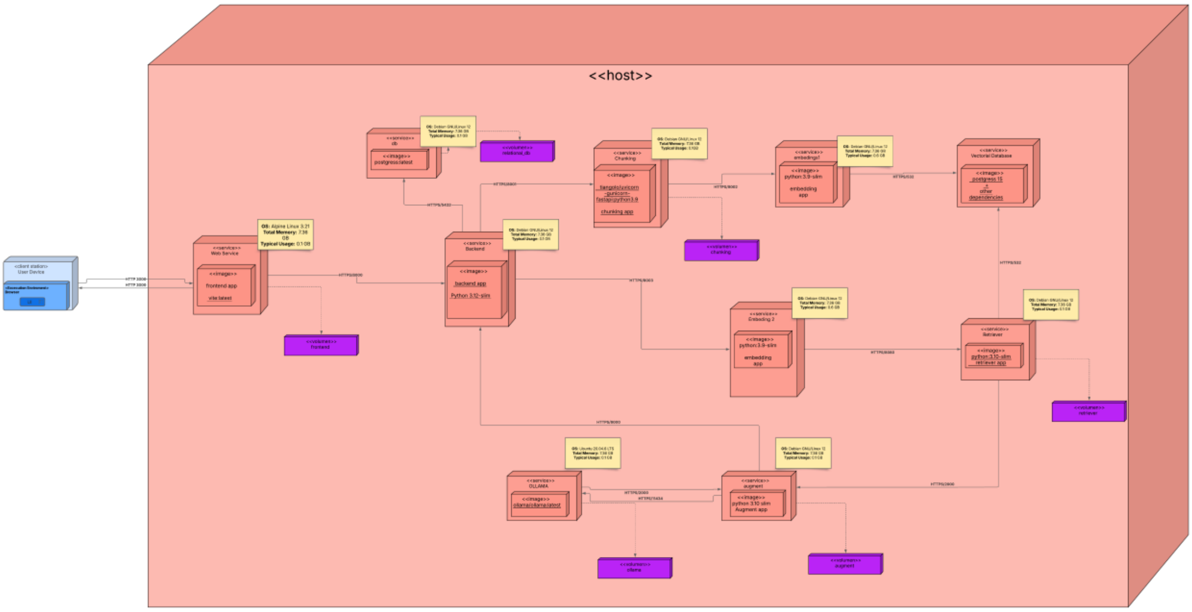


Diagrama de Despliege:

(Una versión)



<<host>>
<client station>
User Device
Browser
UI
HTTP 3000
HTTP 3000
OS: Alpine Linux 3.21
Total Memory: 7.38 GB
Typical Usage: 0.1 GB
<<service>>
Web Service
<<image>>
frontend app
vite:latest
<<volumen>>
frontend
<<service>>
db
<<image>>
postgress:latest
<<volumen>>
relational_db
<<service>>
Backend
<<image>>
backend app
Python 3.12-slim
<<service>>
Chunking
<<image>>
tiangolo/uvicorn
-gunicorn-
fastapi:python3.9
chunking app
<<volumen>>
chunking
<<service>>
embedings1
<<image>>
python:3.9-slim
embedding
app
<<service>>
Vectorial Database
<<image>>
postgress 15
+
other
dependencies
<<service>>
Embeding 2
<<image>>
python:3.9-slim
embedding
app
<<service>>
Retriever
<<image>>
python:3.10-slim
retriever app
<<volumen>>
retriever
<<service>>
OLLAMA
<<image>>
ollama/ollama:latest
<<volumen>>
ollama
<<service>>
augment
<<image>>
python 3.10 slim
Augment app
<<volumen>>
augment