ALFIE: Asistente Médico Modular Autónomo (Local en Apple Silicon)

ALFIE es un asistente de IA que funciona completamente en local, diseñado para correr en un Mac Mini con Apple Silicon (M4). El sistema es modular, autónomo y **no requiere conexión a la nube**, garantizando privacidad. Utiliza **Ollama** como backend para manejar varios modelos de lenguaje grandes (LLMs) optimizados para local, incluyendo *Mistral*, *Phi*, *Nous-Hermes* y un modelo personalizado llamado **"alfie"** entrenado en datos médicos psicológicos. ALFIE ofrece una interfaz gráfica moderna (Tauri + React/Svelte) con un diseño minimalista y personalizable, permitiendo interacción en tiempo real (respuestas por streaming), memoria persistente editable y opciones para controlar su comportamiento (modo aprendizaje, selección de modelo, tono de respuesta, etc.). A continuación, se detalla la arquitectura del proyecto, cada uno de sus módulos Python con código completo y explicación, la integración con la interfaz gráfica y las instrucciones para desplegar el sistema.

Arquitectura General del Proyecto

ALFIE se compone de dos partes principales: un **backend local en Python** y un **frontend de escritorio** construido con Tauri (HTML/CSS/JS). El backend maneja la lógica de la conversación, la gestión de memoria y la interacción con los modelos de IA a través de Ollama. El frontend proporciona la interfaz de usuario (ventana de chat, controles, etc.) que se comunica con el backend para enviar preguntas del usuario y mostrar las respuestas en tiempo real.

Componentes clave:

- **Backend (Python)**: Contiene módulos para configuración, memoria persistente, utilidades, gestión del chat y la interacción con modelos:
    - `config.py`: Configuración global (rutas, opciones por defecto, modelos disponibles, etc.).
    - `memory.py`: Manejo de memoria persistente (conocimientos guardados, contexto adicional).
    - `utils.py`: Funciones utilitarias (configuración de logging, manejo de comandos internos, etc.).
    - `chat_manager.py`: Lógica principal del chat (construcción de prompts, llamado al modelo Ollama, streaming de respuestas, gestión de estado del asistente).
    - `main.py`: Punto de entrada del backend; inicia el servidor local (FastAPI) y expone endpoints para que el frontend pueda interactuar (enviar mensajes, obtener respuestas streaming, modificar memoria, cambiar modelo/tono, etc.).
    - **Gestión de logs**: El backend registra eventos e interacciones en un archivo de log para debugging o auditoría (archivo de log rotativo en una carpeta `logs/`).
    - **Gestión de modelos**: El backend utiliza Ollama para ejecutar los distintos modelos (Mistral, Phi, Nous-Hermes, Alfie). Los nombres y parámetros de modelos están

centralizados en la configuración, y se pueden seleccionar dinámicamente durante la ejecución.

- **Frontend (Tauri + React/Svelte)**: Una aplicación de escritorio con:
  - Interfaz de chat limpia y minimalista, con burbujas de conversación (usuario y ALFIE).
  - **Campo de entrada** para las preguntas del usuario y botones para enviar.
  - **Área de memoria editable** donde se muestra y permite editar la memoria persistente de ALFIE (hechos o información que el asistente recuerda a largo plazo).
  - **Selector de modelo** para elegir qué modelo de lenguaje usar (por defecto el modelo "alfie"; opciones de Mistral, Phi, Nous-Hermes, etc.).
  - **Selector de tono/comportamiento** para ajustar el estilo de las respuestas (por ejemplo: neutral, empático, formal, casual, etc.).
  - **Modo aprendizaje ON/OFF**: un interruptor que permite activar o desactivar la persistencia de la conversación en la memoria (ON por defecto, para que ALFIE "aprenda" y recuerde).
  - **Botón "Forzar Nous-Hermes"**: opción rápida para que la siguiente respuesta use específicamente el modelo Nous-Hermes (independientemente del modelo seleccionado por defecto), ideal para obtener una respuesta más extensa o detallada de forma puntual.
  - La comunicación con el backend se realiza mediante llamadas HTTP locales o eventos en tiempo real (Server-Sent Events para streaming de tokens, asegurando que la respuesta se muestre progresivamente a medida que se genera).
- **Autonomía y modularidad**: ALFIE está preparado para funcionar de forma **autónoma** sin intervención de la nube, iniciándose automáticamente con el sistema (configurable como servicio de **LaunchAgent** en macOS). Su diseño modular facilita añadir futuras extensiones, como entrada/salida de voz, análisis de imágenes médicas, integración de sensores o plugins de herramientas (ej. realizar búsquedas locales, cálculos, etc.), mediante nuevos módulos que interactúen con el `chat_manager`. Los *comandos internos* (p. ej. `@recordar`, `@memoria`) permiten al usuario influir directamente en la memoria o comportamiento sin involucrar al modelo de lenguaje, añadiendo flexibilidad y capacidad de extensión.

A continuación, se detallan **todos los módulos del backend en Python** con su código completo y explicación, seguidos por la descripción de la interfaz Tauri y las instrucciones de despliegue.

Módulos del Backend (Python)

En esta sección se presentan uno a uno los archivos Python principales del backend de ALFIE, mostrando el código fuente y explicando su funcionalidad.

1. Módulo `config.py`

El módulo `config.py` define la **configuración global** y constantes utilizadas en todo el proyecto. Esto incluye rutas de archivos (por ejemplo, dónde guardar la memoria persistente y los logs), opciones por defecto para el modelo de lenguaje y tono de respuesta, y otros parámetros ajustables. Mantener esta configuración centralizada facilita la modificación de comportamientos sin tener que cambiar varias partes del código.

```
python
CopiarEditar
# config.py

import os
from pathlib import Path

# Directorio base del proyecto (asumiendo que config.py está en la carpeta backend)
BASE_DIR = Path(__file__).resolve().parent

# Carpeta para almacenar archivos de logs y memoria, crear si no existe
LOGS_DIR = BASE_DIR / "logs"
MODELS_DIR = BASE_DIR / "models"    # (opcional, si hubiera archivos de modelos locales)
MEMORY_FILE = BASE_DIR / "alfie_memory.txt"

os.makedirs(LOGS_DIR, exist_ok=True)

# Archivo de log
LOG_FILE = LOGS_DIR / "alfie.log"

# Configuración de modelos disponibles (nombre visible -> id de modelo Ollama)
MODELS = {
    "Alfie": "alfie",            # modelo personalizado Alfie (debe estar cargado en Ollama)
    "Mistral": "mistral",        # ejemplo: Mistral 7B (asegúrese de tenerlo en Ollama)
    "Phi": "phi-3.5",            # ejemplo: Phi 3.5 (Modelo ligero de Microsoft)
    "Nous-Hermes": "nous-hermes" # modelo Nous-Hermes (Llama2 13B fine-tune)
}
DEFAULT_MODEL = "Alfie"  # nombre por defecto (clave de MODELS)

# Tonos/comportamientos de respuesta disponibles
TONOS = {
```

```
    "Neutral": "neutral y objetivo",
    "Empático": "empático y solidario",
    "Formal": "formal y profesional",
    "Informal": "informal y cercano"
}
DEFAULT_TONE = "Neutral"  # tono por defecto

# Modo aprendizaje (memoria) activado por defecto
LEARNING_MODE_DEFAULT = True

# Parámetros de servidor backend
HOST = "127.0.0.1"
PORT = 8000

# Otros parámetros configurables
MAX_HISTORY_LENGTH = 10   # número máximo de intercambios
previos a mantener en contexto (para limitar tamaño del
prompt)
```

Explicación:

- Se determina el `BASE_DIR` a partir de la ubicación de `config.py`. A partir de ahí, se definen rutas para la carpeta de logs, carpeta de modelos (si se usan archivos locales) y el archivo de memoria persistente (`alfie_memory.txt`). Si la carpeta de logs no existe, se crea.

- `MODELS` es un diccionario que mapea nombres legibles (claves) a identificadores de modelos utilizados por Ollama. Estos nombres legibles aparecen en la interfaz (selector de modelo). **Importante:** Antes de usar ALFIE, los modelos correspondientes deben estar **descargados en Ollama** (por ejemplo, ejecutando `ollama pull nombre_modelo`). Los identificadores deben corresponder a los nombres conocidos por Ollama. En el ejemplo:

    - "alfie" sería un modelo personalizado agregado a Ollama (vía `ollama push alfie <path>` o similar).
    - "mistral" representa un modelo Mistral (asegúrese del nombre exacto en Ollama, podría ser `"mistral-7b"` o similar según la versión).
    - "phi-3.5" corresponde a una versión ligera del modelo Phi (p. ej., Phi 3.5 mini).
    - "nous-hermes" representa Nous-Hermes (Llama2 13B fine-tune por Nous Research).

- `DEFAULT_MODEL` es la clave por defecto (en este caso, "Alfie"). Esto indica que inicialmente ALFIE utilizará su modelo personalizado.

- `TONOS` define estilos de respuesta. Cada clave es el nombre del tono (usado en la interfaz) y el valor es la descripción que se integrará en el prompt del modelo. Por ejemplo, si el tono es "Empático", el prompt le indicará al modelo que responda de manera *"empática y solidaria"*. Se pueden añadir más estilos según necesidades.

- `DEFAULT_TONE` define el tono inicial predeterminado (Neutral).
- `LEARNING_MODE_DEFAULT` se establece a `True` para que el **modo aprendizaje (memoria activa)** esté encendido por defecto, es decir, el asistente recordará por omisión las interacciones y las integrará en su memoria persistente.
- Bajo "Parámetros de servidor backend" se configuran host y puerto donde correrá el servidor local (127.0.0.1:8000 por defecto). `MAX_HISTORY_LENGTH` limita cuántos turnos previos de conversación incluir en el contexto enviado al modelo, para evitar desbordar la ventana de contexto de este (valor ajustable; si se pone None o un número muy alto, incluirá todo el historial, pero con modelos pequeños es recomendable limitarlo).

2. Módulo `memory.py`

El módulo `memory.py` gestiona la **memoria persistente** de ALFIE. La memoria persistente es información que el asistente **recuerda entre sesiones** y que se puede editar manualmente. Por ejemplo, puede incluir detalles relevantes de conversaciones pasadas, datos sobre el perfil del usuario (nombre, alergias, etc.) o cualquier hecho que el usuario le haya pedido *recordar*. Esta memoria se almacena en disco (en un archivo de texto) para que no se pierda al reiniciar la aplicación.

```
python
CopiarEditar
# memory.py

import json
from typing import List
from config import MEMORY_FILE

class MemoryManager:
    """
    Gestiona la memoria persistente de ALFIE.
    La memoria se guarda en un archivo de texto
(MEMORY_FILE). Puede ser editada por el usuario.
    """
    def __init__(self):
        # Cargar memoria desde el archivo, o inicializar
vacía si no existe
        self.memory_content: str = ""
        if MEMORY_FILE.exists():
            try:
                self.memory_content =
MEMORY_FILE.read_text(encoding='utf-8').strip()
            except Exception as e:
                self.memory_content = ""
                print(f"Error cargando memoria: {e}")
        else:
            # Si no existe el archivo, crearlo vacío
```

```
            MEMORY_FILE.write_text("", encoding='utf-8')
            self.memory_content = ""

    def get_memory(self) -> str:
        """Devuelve la memoria persistente completa como
texto."""
        return self.memory_content

    def add_entry(self, new_info: str):
        """
        Agrega una nueva entrada de información a la memoria
persistente.
        La nueva información se añade al final, separada por
un salto de línea.
        """
        new_info = new_info.strip()
        if not new_info:
            return
        # Agregar como nueva línea a la memoria existente
        if self.memory_content:
            self.memory_content += "\n" + new_info
        else:
            self.memory_content = new_info
        # Guardar en archivo
        try:
            MEMORY_FILE.write_text(self.memory_content,
encoding='utf-8')
        except Exception as e:
            print(f"Error guardando en memoria: {e}")

    def update_memory(self, full_text: str):
        """
        Reemplaza el contenido completo de la memoria
persistente por el texto proporcionado.
        Se utiliza cuando el usuario edita manualmente la
memoria desde la interfaz.
        """
        self.memory_content = full_text.strip()
        try:
            MEMORY_FILE.write_text(self.memory_content,
encoding='utf-8')
        except Exception as e:
            print(f"Error actualizando memoria: {e}")

    def clear_memory(self):
```

```
        """Borra la memoria persistente (con precaución)."""
        self.memory_content = ""
        try:
            MEMORY_FILE.write_text("", encoding='utf-8')
        except Exception as e:
            print(f"Error al borrar la memoria: {e}")
```

Explicación:

- La clase `MemoryManager` encapsula la lógica de cargar, almacenar y modificar la memoria persistente.
- En el método `__init__`, se intenta cargar el contenido existente del archivo `MEMORY_FILE` (ruta definida en config). Si el archivo no existe, se crea uno vacío. La memoria se maneja como un bloque de texto (`self.memory_content`).
- `get_memory()` devuelve el texto completo de la memoria almacenada (usado para mostrarla en la interfaz o incluirla en el contexto del modelo).
- `add_entry(new_info)` añade una nueva línea de texto a la memoria:
  - Si `new_info` está vacío (cadena vacía tras strip), no hace nada.
  - Si ya existe algo de memoria, añade la nueva información empezando en una nueva línea; si estaba vacía, simplemente asigna el texto.
  - Inmediatamente escribe el contenido actualizado al archivo, para asegurar persistencia.
  - Este método se utiliza para el comando interno `@recordar` – cuando el usuario pide recordar algo, se añade a la memoria.
- `update_memory(full_text)` reemplaza todo el contenido de la memoria con el texto dado (por ejemplo, cuando el usuario edita manualmente la memoria en la interfaz y guarda cambios). Se hace strip para remover espacios extra y se escribe al archivo.
- `clear_memory()` vacía la memoria tanto en la instancia como en el archivo (podría vincularse a un comando interno tipo `@borrar_memoria` si se quisiera, aunque no se especificó como requisito, se incluye por completitud y precaución).

Esta implementación guarda la memoria como texto libre. El formato específico del contenido depende del uso: se sugiere almacenar hechos o notas separados por líneas (posiblemente enumeradas con guiones `-` o similar para legibilidad), pero se deja flexible. Por ejemplo, la memoria podría lucir así en el archivo:

```
css
CopiarEditar
El usuario se llama Juan Perez.
Alergia a penicilina.
Prefiere explicaciones breves.
```

El asistente puede incorporar este texto en sus respuestas para recordar esos detalles. Dado que la memoria es **editable**, el usuario puede corregir o eliminar información desde la interfaz.

3. Módulo `utils.py`

El módulo `utils.py` proporciona **funciones utilitarias** generales, como la configuración del logging y el manejo de comandos especiales. Separar estas utilidades ayuda a mantener el código ordenado y evita repetir lógica en varios lugares.

python
CopiarEditar

```
# utils.py

import logging
from logging.handlers import RotatingFileHandler
from config import LOG_FILE

def setup_logging():
    """
    Configura el sistema de logging para el asistente ALFIE.
    Crea un logger que escribe en consola y en archivo con
rotación.
    """
    logger = logging.getLogger("alfie")
    logger.setLevel(logging.INFO)
    # Formato de log: fecha/hora nivel mensaje
    formatter = logging.Formatter("%(asctime)s - %
(levelname)s - %(message)s")
    # Handler para archivo con rotación (1 MB por archivo, 3
backups)
    file_handler = RotatingFileHandler(LOG_FILE,
maxBytes=1_000_000, backupCount=3, encoding='utf-8')
    file_handler.setFormatter(formatter)
    file_handler.setLevel(logging.INFO)
    logger.addHandler(file_handler)
    # Handler para la consola (stdout)
    console_handler = logging.StreamHandler()
    console_handler.setFormatter(formatter)
    console_handler.setLevel(logging.INFO)
    logger.addHandler(console_handler)
    logger.info("Logger inicializado, guardando en %s",
LOG_FILE)
    return logger

def is_command(message: str) -> bool:
    """
```

```
    Determina si un mensaje de usuario es un comando interno
(empieza con '@').
    """
    return message.strip().startswith("@")

def parse_command(message: str) -> (str, str):
    """
    Parsea un mensaje de comando. Devuelve una tupla
(comando, contenido).
    Por ejemplo, '@recordar tomar medicina' -> ('recordar',
'tomar medicina')
    El comando se devuelve en minúsculas sin la '@'. El
contenido es lo que sigue.
    """
    text = message.strip()
    if not text.startswith("@"):
        return ("", "")
    # Remover '@' inicial y dividir en primera palabra
(comando) y resto (contenido)
    parts = text[1:].split(" ", 1)
    command = parts[0].lower() if parts else ""
    content = parts[1].strip() if len(parts) > 1 else ""
    return (command, content)
```

Explicación:

- `setup_logging()`: Configura el logger principal para ALFIE. Usa la ruta `LOG_FILE` definida en config. Se configura un handler de archivo con rotación (rotará el archivo de log cuando supere ~1MB, guardando hasta 3 archivos antiguos) para evitar crecimiento infinito del log. También se configura un handler de consola para ver mensajes en tiempo real si se ejecuta en modo desarrollador. El nivel de logging se establece en INFO para registrar eventos generales; se puede cambiar a DEBUG para depuración más detallada. Tras configurar, se registra un mensaje inicial indicando que el logger está listo.

- `is_command(message)`: Devuelve True si el mensaje del usuario parece ser un comando interno, definido aquí como cualquier entrada que comienza con el carácter `@`. Esto se usará para desviar la lógica en el chat: mensajes que son comandos no se mandarán al modelo de lenguaje sino que serán manejados internamente.

- `parse_command(message)`: Analiza un mensaje de comando y separa el nombre del comando de su contenido. Se quita el `@` inicial y luego:

    - `command`: la primera palabra tras `@`, en minúsculas (p. ej. "@Recordar algo" -> comando "recordar").

    - `content`: el resto del mensaje tras el comando, con espacios externos eliminados.

    - Si no hay contenido (ej. mensaje era solo "@memoria"), `content` será cadena vacía.

- Devuelve una tupla `(command, content)`.

Estos comandos internos permiten funcionalidad especial:

- **`@recordar <texto>`**: Indica que el asistente debe guardar `<texto>` en su memoria persistente. (Implementado más adelante en `chat_manager` usando `MemoryManager.add_entry`).
- **`@memoria`**: Solicita al asistente mostrar el contenido actual de su memoria persistente. (Se implementará para que el asistente devuelva ese texto al usuario, o se muestre en la interfaz).
- Podemos extender fácilmente con nuevos comandos siguiendo esta estructura:
  - Por ejemplo, `@modelo Nous-Hermes` podría usarse para cambiar el modelo activo desde la línea de chat (aunque la interfaz ya provee un selector, es otra vía).
  - `@tono Empático` para cambiar el tono vía chat.
  - `@borrar_memoria` para limpiar la memoria, etc.

Por ahora nos centramos en los comandos principales indicados.

4. Módulo `chat_manager.py`

Este es uno de los módulos centrales: `chat_manager.py` contiene la clase que **coordina la conversación**, integrando la memoria, la configuración (modelo y tono actuales) y llamando a los modelos de lenguaje mediante Ollama. También maneja la lógica de streaming de respuestas y los comandos internos. En esencia, este módulo decide *qué* se envía al modelo y *cómo* se procesa la respuesta para devolverse al usuario.

```
python
CopiarEditar
# chat_manager.py

from ollama import chat  # Librería de Ollama para invocar
modelos locales
from config import MODELS, DEFAULT_MODEL, TONOS,
DEFAULT_TONE, MAX_HISTORY_LENGTH
from memory import MemoryManager
from utils import is_command, parse_command
import logging

class ChatManager:
    def __init__(self):
        # Inicializar la memoria y la configuración inicial
        self.memory_manager = MemoryManager()
        self.current_model_name = DEFAULT_MODEL  # Nombre
clave del modelo en uso (e.g., "Alfie")
```

```
        self.current_tone = DEFAULT_TONE         # Tono/
comportamiento actual
        self.learning_mode = True                # Modo
aprendizaje (memoria) activado
        # Historial de conversación reciente (lista de dicts:
{"role": ..., "content": ...})
        self.chat_history = []
        self.logger = logging.getLogger("alfie")
        self.logger.info(f"ChatManager inicializado con
modelo {self.current_model_name} tono {self.current_tone}")

    def set_model(self, model_name: str):
        """Cambia el modelo actual (por nombre lógico, debe
existir en config.MODELS)."""
        if model_name in MODELS:
            self.current_model_name = model_name
            self.logger.info(f"Modelo cambiado a:
{model_name}")
        else:
            self.logger.warning(f"Modelo '{model_name}' no
reconocido. Manteniendo modelo actual.")

    def set_tone(self, tone_name: str):
        """Cambia el tono/comportamiento actual del
asistente."""
        if tone_name in TONOS:
            self.current_tone = tone_name
            self.logger.info(f"Tono de respuesta cambiado a:
{tone_name}")
        else:
            self.logger.warning(f"Tono '{tone_name}' no
reconocido. Manteniendo tono actual.")

    def set_learning_mode(self, mode: bool):
        """Activa o desactiva el modo aprendizaje (memoria
persistente)."""
        self.learning_mode = mode
        state = "activado" if mode else "desactivado"
        self.logger.info(f"Modo aprendizaje {state}.")

    def _build_prompt_messages(self, user_message: str) ->
list:
        """
        Construye la lista de mensajes (formato chat) a
enviar al modelo.
```

```
        Incluye:
          - Mensaje de sistema con personalidad y tono.
          - Mensaje de sistema con memoria persistente (si
existe contenido).
          - Historial de últimos turnos (truncado a
MAX_HISTORY_LENGTH).
          - Mensaje del usuario actual.
        """
        messages = []
        # 1. Mensaje de sistema con instrucciones de
personalidad y tono
        tone_description = TONOS.get(self.current_tone,
TONOS[DEFAULT_TONE])
        system_persona = (f"Eres ALFIE, un asistente médico
de IA. "
                          f"Responde en español de forma
clara y comprensible. "
                          f"Tu tono debe ser
{tone_description}. "
                          f"Proporciona información precisa y
útil, basándote en tu conocimiento médico. "
                          f"Si no sabes algo, admítelo en
lugar de inventar. "
                          f"Eres autónomo y funcionas
completamente en local.")
        messages.append({"role": "system", "content":
system_persona})
        # 2. Mensaje de sistema con memoria persistente
(contexto adicional)
        memory_text = self.memory_manager.get_memory()
        if memory_text:
            # Incluir la memoria como contexto conocido
            system_memory = ("Contexto: A continuación hay
información que recuerdas de interacciones previas "
                             "y hechos relevantes:\n" +
memory_text)
            messages.append({"role": "system", "content":
system_memory})
        # 3. Historial reciente de conversación
        if self.chat_history:
            # Tomar solo los últimos N turnos según
MAX_HISTORY_LENGTH
            history_to_use = self.chat_history[-
MAX_HISTORY_LENGTH*2:] if MAX_HISTORY_LENGTH else
self.chat_history
```

```
            # history_to_use contendrá alternando roles
usuario/assistant (2 mensajes por turno)
            for msg in history_to_use:
                messages.append(msg)
        # 4. Mensaje actual del usuario
        messages.append({"role": "user", "content":
user_message})
        return messages

    def handle_user_message(self, user_message: str):
        """
        Maneja un mensaje ingresado por el usuario.
        Si es un comando interno, lo ejecuta y produce la
respuesta apropiada directamente.
        Si es normal, construye el prompt y obtiene respuesta
del modelo.
        Devuelve un dict con posible 'response' (respuesta
completa) o 'stream' (iterable para streaming).
        """
        user_message = user_message.strip()
        if user_message == "":
            return {"response": ""}

        # Comando interno (no enviar al modelo)
        if is_command(user_message):
            command, content = parse_command(user_message)
            if command == "recordar":
                if content:
                    # Agregar a memoria
                    self.memory_manager.add_entry(content)
                    confirmation = "✅ He recordado esa
información."
                    self.logger.info(f"Comando @recordar
ejecutado: '{content}'")
                    # Opcionalmente, podemos añadir la
confirmación como respuesta de asistente en historial
                    self.chat_history.append({"role":
"assistant", "content": confirmation})
                    return {"response": confirmation}
                else:
                    return {"response": "⚠️ No se proporcionó
información para recordar."}
            elif command == "memoria":
                # Devolver contenido de memoria
```

```
                mem = self.memory_manager.get_memory()
                mem_text = mem if mem else "(Memoria vacía)"
                self.logger.info("Comando @memoria ejecutado:
mostrando memoria persistente.")
                # No agregamos esto al historial de prompts
del modelo, es una respuesta directa
                return {"response": f"📖 Memoria:
\n{mem_text}"}
            elif command == "modelo":
                # Permite cambiar modelo vía comando, e.g.
"@modelo Nous-Hermes"
                if content:
                    self.set_model(content.strip())
                    return {"response": f"🔄 Modelo cambiado
a {self.current_model_name}."}
                else:
                    return {"response": "⚠️ Especifique el
modelo a usar. Ej: @modelo Alfie"}
            elif command == "tono":
                # Permite cambiar tono vía comando, e.g.
"@tono Formal"
                if content:
                    self.set_tone(content.strip())
                    return {"response": f"🔄 Tono de
respuesta cambiado a {self.current_tone}."}
                else:
                    return {"response": "⚠️ Especifique el
tono deseado. Ej: @tono Empático"}
            elif command == "borrar_memoria":
                # Limpia la memoria persistente
                self.memory_manager.clear_memory()
                self.logger.warning("Memoria persistente
borrada via comando @borrar_memoria.")
                return {"response": "🗑️ Memoria persistente
borrada."}
            else:
                # Comando no reconocido
                return {"response": f"🤖 Comando '{command}'
no reconocido."}
        else:
            # Mensaje normal - obtener respuesta del modelo
            # Construir prompt con historial, memoria, etc.
```

```
            prompt_messages =
self._build_prompt_messages(user_message)
            # Llamar al modelo actual a través de Ollama
            model_id = MODELS.get(self.current_model_name,
MODELS[DEFAULT_MODEL])
            self.logger.info(f"Enviando prompt al modelo
'{self.current_model_name}'...")
            # Usar streaming para obtener respuesta parcial
            stream = chat(model=model_id,
messages=prompt_messages, stream=True)
            # Actualizar historial con el mensaje del usuario
            self.chat_history.append({"role": "user",
"content": user_message})
            # Preparar un generador para transmitir la
respuesta progresivamente
            def response_generator():
                response_text = ""
                for chunk in stream:
                    # Cada chunk es un dict con 'message':
{'role': 'assistant', 'content': '...'}
                    content_part = chunk['message']
['content']
                    if content_part:
                        response_text += content_part
                        yield content_part  # enviar el nuevo
fragmento al cliente
                # Una vez finalizada la respuesta completa:
                # Agregar al historial la respuesta final del
asistente
                if response_text:
                    self.chat_history.append({"role":
"assistant", "content": response_text})
                    # Si el modo aprendizaje está activado,
opcionalmente podemos guardar parte de esta info en memoria
persistente
                    # (Por ejemplo, podríamos extraer
conclusiones importantes y guardarlas, pero por simplicidad
no hacemos auto-record en este punto)
                self.logger.info("Respuesta completa generada
por el modelo.")
            return {"stream": response_generator()}
```

Explicación del `ChatManager`:

- `self.memory_manager`: instancia de `MemoryManager` para acceder a la memoria persistente.

- `self.current_model_name`: modelo en uso (nombre lógico como "Alfie"). Por defecto se inicia con `DEFAULT_MODEL` de config.

- `self.current_tone`: tono de respuesta actual (inicia con `DEFAULT_TONE`).

- `self.learning_mode`: indica si el asistente está en modo aprendizaje (True por defecto, ver `LEARNING_MODE_DEFAULT`). Si está activado, el asistente almacena información de las conversaciones en su memoria persistente cuando corresponda. *Nota:* En esta implementación simple, no estamos añadiendo automáticamente nada a la memoria persistente excepto cuando el usuario explícitamente use `@recordar`. Pero podríamos extender para que, con `learning_mode=True`, el asistente analice cada respuesta y decida qué recordar de la conversación, guardándolo con `memory_manager.add_entry`. Esto se deja como posible mejora (requiere procesamiento de lenguaje para extraer hechos).

- `self.chat_history`: lista que almacena el historial reciente de la conversación en formato de mensajes para el modelo (role/content). Usamos esto para proporcionar contexto de la conversación al modelo en cada nueva pregunta. Solo se guarda en memoria de la aplicación (no en disco), y se reinicia al reiniciar el backend (a menos que se decida persistir también el historial, lo cual no se hace aquí para mantener la memoria persistente manejable).

- Los métodos `set_model`, `set_tone`, `set_learning_mode` permiten actualizar la configuración dinámica durante la ejecución:

    - `set_model(model_name)`: Cambia el modelo activo si el nombre proporcionado existe en `MODELS`. Si no, registra una advertencia y deja el actual. Esto será llamado desde la API cuando el usuario seleccione un modelo diferente en la UI.

    - `set_tone(tone_name)`: Similar para el tono; cambia el estilo de respuesta del asistente si está en la lista `TONOS`. Un tono diferente afectará cómo se construye el mensaje de sistema (en `_build_prompt_messages`).

    - `set_learning_mode(mode)`: Activa o desactiva la recolección de conocimientos de la conversación. Aquí solo cambia la bandera y registra en log. Más adelante podríamos usar esta bandera para condicionar si agregamos automáticamente cosas a memoria o no.

- `_build_prompt_messages(user_message)`: Este método **arma la lista de mensajes** que se enviará al modelo cada vez que el usuario hace una pregunta. Siguiendo el formato estilo ChatGPT:

    - **Sistema (personalidad y tono)**: Un mensaje de rol "system" que establece quién es ALFIE, cómo debe responder, el tono de comunicación y algunas instrucciones (no alucinar, es local, etc.). En este mensaje se inserta la descripción del tono actual (`tone_description`) obtenida del diccionario `TONOS`. Por ejemplo, si el tono es Empático, el mensaje podría decir: "*Tu tono debe ser empático y solidario*.". También define que es un asistente médico en español, con respuestas claras, etc. Este es un **prompt fundamental** para guiar el comportamiento del modelo.

- **Sistema (memoria persistente)**: Si `memory_text` (lo que hay guardado en memoria) no está vacío, se añade otro mensaje de rol "system" que provee esa información como *contexto conocido*. Lo formulamos como "*Contexto: A continuación hay información que recuerdas...*" seguido del contenido de la memoria. Así, el modelo recibe esos datos como antecedentes que puede usar para responder. Ejemplo: "*Contexto: ... Alergia a penicilina.\nPrefiere explicaciones breves.*". Este bloque permite que el asistente incorpore hechos recordados a sus respuestas.

- **Historial de conversación**: Recorremos `self.chat_history` (que contiene mensajes anteriores de usuario y asistente). Para no sobrepasar la capacidad del modelo, tomamos solo las últimas `MAX_HISTORY_LENGTH`interacciones (cada interacción abarca 2 mensajes: usuario+asistente). Entonces `history_to_use` obtiene los últimos N*2 mensajes. Cada mensaje es ya un dict `{"role": ..., "content": ...}` y se añade en orden. Esto proporciona al modelo el contexto de la conversación reciente, para que la respuesta sea coherente con lo ya preguntado/respondido. Si `MAX_HISTORY_LENGTH` es None o suficientemente grande, en teoría incluiría todo el historial, pero normalmente limitamos este tamaño.

- **Mensaje actual del usuario**: Finalmente añadimos el mensaje nuevo del usuario como rol "user" con su contenido. Este es lo último que verá el modelo antes de generar su respuesta.

- El método devuelve la lista `messages` lista para ser pasada a Ollama.

- `handle_user_message(user_message)`: Es la función principal que el backend utiliza cuando llega una nueva entrada del usuario (desde la interfaz). Devuelve la respuesta apropiada en una de dos formas:

  - Un **comando interno** produce una respuesta inmediata (sin llamar al modelo) en la clave `'response'`.

  - Un mensaje normal produce un generador de respuesta en streaming en la clave `'stream'`.

  - Internamente:

    - Limpia espacios en el mensaje. Si por alguna razón está vacío, retorna respuesta vacía al cliente.

    - Usa `is_command` (de utils) para determinar si el mensaje comienza con '@'. Si es un comando:

      - Usa `parse_command` para obtener `(command, content)`.

      - Implementa distintos comandos:

        - **recordar**: Si hay contenido después del comando, lo añade a la memoria persistente con `memory_manager.add_entry(content)`, registra en log y devuelve una confirmación (con un check ✅).

También agrega esa confirmación como mensaje de asistente al historial chat (opcional, para que quede constancia en la conversación visible de que “recordó” algo, aunque este mensaje no se envía al modelo).

- Si el usuario escribió `@recordar` sin contenido, devuelve una alerta de que no se proporcionó información.

- **memoria**: Devuelve el contenido completo de la memoria persistente. Si está vacía, indica "(Memoria vacía)". Esto le permite al usuario ver rápidamente qué ha memorizado ALFIE. La respuesta se marca con un libro 📖 para indicar que es la memoria.

- **modelo**: Permite cambiar el modelo actual escribiendo, por ejemplo, `@modelo Nous-Hermes`. Llama a `set_model` con el contenido. Devuelve una respuesta confirmando el cambio (o indicando que especifique modelo si no puso cuál).

- **tono**: Similar, comando para cambiar el tono (ej: `@tono Formal`).

- **borrar_memoria**: Limpia la memoria persistente llamando a `memory_manager.clear_memory()`, y devuelve una respuesta indicando que se borró. (Este comando no se mencionó explícitamente en la solicitud, pero lo agregamos como mejora, dado que se habló de extensiones posibles).

- **desconocido**: Cualquier otro comando no reconocido devuelve un mensaje indicando que no se entiende ese comando.

- Todos estos casos de comando retornan un dict `{"response": "...texto..."}`. La clave `'response'` indica al controlador (que veremos en `main.py`) que debe enviar eso directamente al frontend sin pasar por el flujo de LLM.

- Si **no** es comando:

  - Llama a `_build_prompt_messages` para obtener la lista de mensajes (incluyendo contexto y personalidad).

  - Determina `model_id` obteniendo de `MODELS` (config) el identificador del modelo actual (`self.current_model_name`). Si por algún motivo la clave no existe (modelo inválido), usa el modelo por defecto.

  - Registra en log que va a enviar el prompt al modelo.

- Llama a `ollama.chat(...)` con `stream=True` para obtener un generador de fragmentos de respuesta. Esto utiliza la librería Python de Ollama para mandar la conversación al modelo local y recibir la salida progresivamente.
- Antes de iterar la respuesta, guarda el mensaje del usuario en el historial (`self.chat_history`).
- Define una función generadora interna `response_generator()`. Este generador va leyendo cada `chunk` del stream:
  - Cada `chunk` es un dict con una clave `'message'` conteniendo `'role': 'assistant'` y `'content': '...'` con la parte de texto generada. Ollama va entregando los tokens o fragmentos conforme se generan.
  - Extraemos `content_part` y si no está vacío, lo vamos añadiendo a `response_text`acumulado y lo `yield` al exterior. Al usar `yield`, estamos permitiendo enviar ese fragmento inmediatamente al cliente (frontend) para que lo muestre en tiempo real.
  - Cuando el bucle termina (el modelo terminó su respuesta):
    - Si `response_text` no es vacío, añadimos la respuesta completa como mensaje de asistente al `chat_history`. De este modo, el historial completo se mantiene para contexto en futuras preguntas.
    - Si el modo aprendizaje estuviera habilitado y quisiéramos hacer que ALFIE "aprenda" automáticamente, aquí podríamos analizar `response_text` para extraer algún hecho o conclusión y guardarlo con `memory_manager.add_entry`. (Esto no está implementado por defecto por simplicidad, pues requeriría LLM adicional o reglas para decidir qué recordar automáticamente).
    - Se registra en log que la respuesta completa se generó.
- Finalmente, la función `handle_user_message` retorna `{"stream": response_generator()}`. Quien llame a `handle_user_message` debe detectar esta clave `'stream'` y canalizar el generador hacia el cliente para que reciba los fragmentos en streaming.

Nota sobre streaming: Esta implementación se apoya en la funcionalidad de streaming de Ollama. En el frontend, se usarán *Server-Sent Events (SSE)* u otro mecanismo para recibir gradualmente

estos `content_part` y mostrarlos a medida que llegan, simulando la experiencia de escritura en tiempo real.

Nota de eficiencia: Limitar el historial a `MAX_HISTORY_LENGTH` interacciones y usar modelos locales relativamente pequeños (3B, 7B, 13B parámetros) ayuda a que la respuesta sea más rápida en hardware Apple Silicon. Además, Ollama está optimizado para Mac (usa Metal Performance Shaders en GPU/ANE) para acelerar la inferencia. ALFIE puede manejar conversaciones continuas manteniendo lo esencial del contexto sin exceder la ventana máxima de tokens del modelo.

5. Módulo `main.py`

El módulo `main.py` inicia el servidor backend y define las **rutas de la API local** que permiten al frontend interactuar con el ChatManager. Usaremos **FastAPI** (un framework web ligero y eficiente en Python) para exponer endpoints HTTP, incluyendo soporte para streaming mediante SSE (Server-Sent Events). El backend funcionará en localhost, evitando accesos externos por seguridad. Asimismo, aquí se inicializa el ChatManager global, el logger y se configura el arranque automático.

```
python
CopiarEditar
# main.py

import uvicorn
from fastapi import FastAPI, Request
from fastapi.responses import StreamingResponse, JSONResponse
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from chat_manager import ChatManager
from utils import setup_logging
from config import HOST, PORT, LEARNING_MODE_DEFAULT

# Inicializar logging
logger = setup_logging()

# Inicializar aplicación FastAPI
app = FastAPI(title="ALFIE Assistant Backend", version="1.0")

# Permitir CORS para el frontend local (Tauri file:// or
localhost origin)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # (En producción, limitar al origen
de la app Tauri)
    allow_methods=["*"],
    allow_headers=["*"],
)
```

```
# Instanciar el administrador de chat global
chat_manager = ChatManager()
chat_manager.set_learning_mode(LEARNING_MODE_DEFAULT)  #
configurar modo aprendizaje inicial

# Definir modelos Pydantic para peticiones (para validación
de JSON entrante)
class MessageRequest(BaseModel):
    message: str
    model: str = None   # modelo opcional para override
    tone: str = None    # tono opcional para override

class MemoryUpdateRequest(BaseModel):
    content: str

class ConfigUpdateRequest(BaseModel):
    model: str = None
    tone: str = None
    learning: bool = None

@app.post("/chat")
async def chat_endpoint(req: MessageRequest):
    """
    Endpoint para enviar un mensaje de usuario y obtener una
respuesta completa del asistente.
    Si se provee 'model' o 'tone' en la solicitud, se usa
para esta pregunta (no cambia la configuración global
permanentemente).
    """
    user_message = req.message
    # Opción de modelo/tono por override temporal
    original_model = chat_manager.current_model_name
    original_tone = chat_manager.current_tone
    if req.model:
        chat_manager.set_model(req.model)
    if req.tone:
        chat_manager.set_tone(req.tone)
    result = chat_manager.handle_user_message(user_message)
    # Restaurar modelo/tono originales si fueron temporales
(no permanente)
    chat_manager.set_model(original_model)
    chat_manager.set_tone(original_tone)

    # Si es respuesta directa (por comando interno o final)
    if "response" in result:
```

```
        return {"response": result["response"]}
    # Si es streaming, consumir todo el generador y juntar
respuesta (no streaming en este endpoint)
    elif "stream" in result:
        full_text = "".join([chunk for chunk in
result["stream"]])
        return {"response": full_text}
    else:
        # Caso inesperado
        return {"response": ""}

@app.get("/chat_stream")
async def chat_stream(request: Request):
    """
    Endpoint de streaming: recibe los parámetros via query
(message, opcional modelo y tono)
    y devuelve un StreamingResponse para enviar la respuesta
del modelo token a token.
    Usar EventSource (SSE) en el frontend para consumir.
    """
    # Extraer parametros de query
    user_message = request.query_params.get("message", "")
    model_override = request.query_params.get("model", None)
    tone_override = request.query_params.get("tone", None)
    if not user_message:
        return JSONResponse({"error": "No message provided"},
status_code=400)
    # Guardar configuraciones actuales y aplicar overrides
temporales si existen
    original_model = chat_manager.current_model_name
    original_tone = chat_manager.current_tone
    if model_override:
        chat_manager.set_model(model_override)
    if tone_override:
        chat_manager.set_tone(tone_override)
    result = chat_manager.handle_user_message(user_message)
    # Restaurar config original
    chat_manager.set_model(original_model)
    chat_manager.set_tone(original_tone)

    if "stream" in result:
        # Preparar respuesta en streaming (Server-Sent
Events)
        def event_generator():
            try:
```

```
                for chunk in result["stream"]:
                    # Formato SSE: "data: " + chunk + doble
salto de linea
                    yield f"data: {chunk}\n\n"
            except Exception as e:
                logger.error(f"Error en streaming: {e}")
            finally:
                # Señal de fin de stream
                yield "data: [DONE]\n\n"
        return StreamingResponse(event_generator(),
media_type="text/event-stream")
    else:
        # En caso de comando u otro que devolvió response
directa, mandarla inmediatamente
        resp_text = result.get("response", "")
        # Formato SSE para respuesta completa
        def single_event():
            yield f"data: {resp_text}\n\n"
            yield "data: [DONE]\n\n"
        return StreamingResponse(single_event(),
media_type="text/event-stream")

@app.get("/memory")
async def get_memory():
    """Devuelve el contenido de la memoria persistente."""
    mem = chat_manager.memory_manager.get_memory()
    return {"memory": mem}

@app.post("/memory")
async def update_memory(req: MemoryUpdateRequest):
    """Actualiza la memoria persistente con el contenido
proporcionado (reemplaza todo)."""
    chat_manager.memory_manager.update_memory(req.content)
    return {"status": "ok", "memory":
chat_manager.memory_manager.get_memory()}

@app.post("/config")
async def update_config(req: ConfigUpdateRequest):
    """
    Permite actualizar configuraciones: modelo, tono, y modo
aprendizaje.
    Cualquier campo puede omitirse si no se quiere cambiar.
    """
    if req.model:
        chat_manager.set_model(req.model)
```

```
    if req.tone:
        chat_manager.set_tone(req.tone)
    if req.learning is not None:
        chat_manager.set_learning_mode(req.learning)
    return {
        "model": chat_manager.current_model_name,
        "tone": chat_manager.current_tone,
        "learning": chat_manager.learning_mode
    }

# Punto de entrada para ejecutar el servidor
if __name__ == "__main__":
    logger.info(f"Iniciando servidor ALFIE en http://{HOST}:
{PORT}")
    uvicorn.run("main:app", host=HOST, port=PORT,
log_level="info")
```

Explicación de `main.py`:

- Primero se configura e inicia el logger llamando a `setup_logging()` de utils, para registrar desde el inicio.
- Se crea la aplicación FastAPI y se añade middleware CORS: aquí, por simplicidad, `allow_origins=["*"]`permite peticiones desde cualquier origen. Para mayor seguridad, se podría limitar al origen exacto que usará Tauri (por ejemplo, `http://localhost` o el esquema `tauri://localhost` que utiliza la app de escritorio, o incluso file://). Tauri generalmente carga la UI desde un servidor interno o desde files locales, por lo que es posible necesitar permitir `http://127.0.0.1` si se sirve en desarrollo, o en producción Tauri puede no necesitar CORS porque integra directamente. En cualquier caso, esto asegura que las llamadas desde la aplicación frontend sean aceptadas por el backend.
- Se instancia **una única** instancia de `ChatManager` (`chat_manager`), que mantendrá el estado mientras el backend esté corriendo.
- Se establecen clases de datos con Pydantic para validar entradas JSON:
    - `MessageRequest`: espera un campo `message` (texto del usuario) y opcionalmente `model` y `tone` para overrides instantáneos.
    - `MemoryUpdateRequest`: para actualizar la memoria, con un campo `content` que es el nuevo texto completo de memoria.
    - `ConfigUpdateRequest`: para cambiar configuraciones de modelo, tono o learning mode de manera más general.
- **Endpoint POST `/chat`**: Recibe un mensaje del usuario (JSON con `message`, opcional `model` y `tone`). Está pensado para solicitudes *no* streaming; es decir, espera a componer

toda la respuesta antes de devolverla (en formato JSON). Sin embargo, internamente utiliza igualmente `ChatManager.handle_user_message`. Funcionamiento:

- Lee `user_message`.
- Aplica overrides temporales: si el usuario incluyó un modelo o tono específico en esta petición, se usa `chat_manager.set_model` / `set_tone` para cambiarlo **antes** de manejar el mensaje, guardando los valores originales para restaurarlos después.
- Llama a `chat_manager.handle_user_message(user_message)`.
- Restaura el modelo y tono original del chat_manager (así el cambio era solo para este request, salvo que la intención sea persistirlo, pero en este endpoint asumimos override temporal; persistentes cambios preferimos hacerlos vía `/config` o comandos).
- Dependiendo del resultado:
  - Si `result` contiene `'response'`, simplemente devuelve ese texto en un JSON `{"response": "texto"}`. Esto cubre comandos internos o incluso respuestas completas no streaming.
  - Si `result` contiene `'stream'`, itera todos los chunks del generador y los concatena en `full_text`. Esto bloqueará hasta que termine la respuesta, pero luego devuelve la respuesta completa. (Básicamente haciendo streaming internamente pero sin mandarlo chunk por chunk al cliente). Esto está por si se usa este endpoint de manera sincrónica o para casos donde no se desea implementar SSE en el cliente.

- **Endpoint GET `/chat_stream`**: Este es el punto de entrada diseñado para streaming en tiempo real. Funciona mejor con SSE:
  - Recibe los parámetros por query (porque con SSE es más sencillo que enviar body JSON). Se espera `message` como parámetro de consulta, y opcionalmente `model` y `tone` como query params también para overrides.
  - Si no hay mensaje, devuelve un error 400 JSON.
  - Si hay, realiza la misma lógica de override temporal de modelo/tono, luego llama a `handle_user_message`.
  - Si obtiene un `stream` en la respuesta, define un generador `event_generator()` para empaquetar cada chunk en formato SSE:
    - Para cada `chunk` del stream, hace `yield f"data: {chunk}\n\n"`. La especificación SSE requiere la línea comience con "data: " y termine con doble newline para indicar un evento completo. Esto envía progresivamente los trozos al cliente.

        - Al final, envía un marcador `"[DONE]"` para indicar terminación (podría no ser estrictamente necesario, pero es útil para que el frontend sepa que acabó).
        - Devuelve un `StreamingResponse(event_generator(), media_type="text/event-stream")`. Con esto, el servidor empieza a enviar eventos SSE.
    - Si la respuesta vino como `'response'` (comando interno, etc.), de todos modos formatea uno o dos eventos SSE:
        - Envía el texto completo como un evento `data: respuesta\n\n` y luego un `[DONE]`. De esta forma, el frontend SSE también lo maneja de manera uniforme (aunque en este caso solo será un fragmento único).

- **Endpoint GET `/memory`**: Devuelve la memoria persistente actual en un JSON `{"memory": "<texto>"}`. Esto se usa para llenar la sección editable de memoria en la interfaz al iniciar o cuando se necesite refrescar.
- **Endpoint POST `/memory`**: Recibe un JSON con `content` (el nuevo texto completo de la memoria) y lo guarda usando `MemoryManager.update_memory`. Devuelve un JSON confirmando status y la memoria actualizada. El frontend llamará a este endpoint cuando el usuario edite y guarde la memoria manualmente.
- **Endpoint POST `/config`**: Permite actualizar configuraciones del asistente. Recibe un JSON donde cualquiera de las claves `model`, `tone`, `learning` puede aparecer:
    - `model`: cambia el modelo actual (persistente hasta que se cambie de nuevo).
    - `tone`: cambia el tono actual.
    - `learning`: booleando para activar/desactivar modo aprendizaje.
    - Solo se cambian los campos presentes (los `None` se ignoran). Devuelve el estado nuevo de todas las configuraciones relevantes.
    - Este endpoint puede no ser estrictamente necesario si todo se hace con comandos o UI directos, pero es útil tenerlo para que la UI pueda sincronizar estados o cambiar ajustes sin tener que enviar comandos de texto.
- Al final, en el bloque `if __name__ == "__main__":` se inicia el servidor utilizando Uvicorn (servidor ASGI). Esto permite ejecutar `python main.py` directamente para arrancar el backend en modo desarrollo. En un entorno de despliegue con Tauri, podría integrarse de forma distinta (por ejemplo, usando un sidecar o un subproceso), pero tener esta capacidad de ejecución directa facilita pruebas. `uvicorn.run("main:app", ...)` especifica la aplicación FastAPI a correr y el host/puerto (tomados de config).

Notas importantes:

- **Inicio automático (LaunchAgent)**: Dado que queremos que ALFIE se inicie al arrancar el sistema, podemos configurar en macOS un LaunchAgent que ejecute `main.py`. Sin

embargo, como alternativa, cuando se empaquete la aplicación Tauri, también se puede configurar el binario de backend como un *servicio* que arranca con la app. Exploraremos la configuración de LaunchAgent más adelante en la sección de despliegue.

- **Uso de la API**: La interfaz usará principalmente `/chat_stream` para obtener respuestas en tiempo real. Cuando el usuario envía un mensaje desde la UI, se abrirá una conexión SSE a este endpoint y se recibirá gradualmente la respuesta. Para comandos internos, dado que devolvemos el resultado inmediatamente también via SSE, la UI lo mostrará casi instantáneo.
- **CORS**: En la configuración actual, cualquier origen puede llamar. Dado que se trata de una app local, esto no presenta un riesgo inmediato, pero en producción se podría restringir. Tauri apps ejecutan en un webview con origen `app://` o similar, se pueden ajustar las reglas de CORS/seguridad según convenga.

Con esto, todos los módulos principales del backend están desarrollados. Ya tenemos el servicio capaz de:

- Aceptar peticiones de chat, con streaming o completas.
- Procesar comandos internos (@recordar, @memoria, etc.).
- Gestionar memoria persistente.
- Cambiar entre múltiples modelos locales con Ollama.
- Ajustar tono y comportamiento.
- Registrar logs para depuración.
- Mantener un contexto de conversación limitado para coherencia en diálogos continuos.

Interfaz Gráfica (Frontend Tauri + React/Svelte)

Con el backend en marcha, la siguiente pieza es la **interfaz de usuario**, que se construye con Tauri y tecnologías web (como React o Svelte, a elección del desarrollador). Tauri nos permite crear aplicaciones de escritorio ligeras usando HTML/JS, con un consumo de recursos mucho menor que Electron. Además, puede comunicarse con el backend Python local a través de HTTP o invocando comandos locales.

Diseño y funcionalidades de la UI

La interfaz de ALFIE está pensada para ser **minimalista, limpia y funcional**, enfocada en la conversación y el control del asistente. Algunos elementos clave del diseño:

- **Ventana principal de chat**: Ocupa la mayor parte de la pantalla, mostrando los mensajes de la conversación en forma de burbujas o bloques:
    - Mensajes del **usuario** alineados a la derecha (por ejemplo, con un fondo azul claro).
    - Respuestas de **ALFIE** alineadas a la izquierda (fondo gris o verde suave), tipo chat clásico.

    - Cada bloque de mensaje muestra el texto y opcionalmente pequeñas etiquetas (ej. "Usuario" o "ALFIE" arriba).
    - El scroll permite revisar la conversación.

- **Caja de entrada de texto**: Debajo del chat, un campo donde el usuario escribe su pregunta o comando. Al lado, un botón de enviar (por ejemplo un ícono de papel aeroplano).
- **Controles superiores o barra lateral**: Para opciones de configuración:
    - **Selector de Modelo**: Un menú desplegable que lista "Alfie (por defecto)", "Mistral", "Phi", "Nous-Hermes". Cambiar esta selección envía una petición al endpoint `/config` o usa el comando interno correspondiente para actualizar el modelo activo en el backend. Puede haber una indicación del modelo actual en uso.
    - **Selector de Tono**: Otro desplegable con "Neutral", "Empático", "Formal", "Informal". Al cambiar, también se actualiza el backend para que futuras respuestas adopten ese estilo. Esto puede cambiar dinámicamente incluso en medio de la conversación.
    - **Toggle Modo Aprendizaje**: Un interruptor (switch) marcado "Aprendizaje" o "Memorizar". Si está en ON (por defecto), ALFIE irá manteniendo memoria de la conversación y el usuario. Si se apaga, ALFIE no almacenará nuevos datos en su memoria persistente durante la conversación (y podría opcionalmente no usar la memoria en las respuestas, aunque en esta implementación igual la usa; se podría modificar para que OFF implique no incluir la memoria en el prompt). El estado de este switch se sincroniza con `chat_manager.learning_mode`.
    - **Botón "Forzar Nous-Hermes"**: Un botón quizás con texto "Boost" o "🔺 Nous-Hermes" que, al presionarlo, indica que la **próxima pregunta** que haga el usuario se contestará con el modelo Nous-Hermes independientemente del seleccionado actual. Esto se logra enviando la próxima pregunta con un parámetro especial (usando `model=Nous-Hermes` en la query SSE). Tras esa respuesta, el modelo por defecto permanece como estaba. Este botón puede usarse, por ejemplo, si el usuario quiere una respuesta más elaborada de vez en cuando usando el modelo más grande.
    - Estos controles pueden situarse en una pequeña barra encima del chat o en un panel lateral que se expande/colapsa para no distraer.
- **Panel de Memoria Persistente**: Un área (por ejemplo un botón "Memoria 📖" que abre un modal o panel) donde se muestra el contenido actual de la memoria persistente en un textarea multi-línea. El usuario puede leer todo lo que ALFIE ha guardado y también **editar** libremente este texto. Por ejemplo, puede corregir un dato o añadir manualmente información. Habrá un botón "Guardar" para enviar el contenido editado al endpoint `/memory` (POST), actualizando la memoria en el backend. También podría haber un botón "Borrar Memoria" para limpiar todo (que llamaría a `@borrar_memoria` o endpoint correspondiente).
- **Indicador de estado**: Como las respuestas son streaming, podría haber un pequeño indicador (como tres puntitos animados o un texto "Escribiendo...") mientras ALFIE está generando una respuesta. Esto se puede lograr detectando la conexión SSE abierta y cerrada.

- **Notificaciones visuales**: Cuando se usan comandos internos, las respuestas devueltas llevan emojis (✅, ⚠️, 📖, etc.) según el caso, dando pistas visuales (por ejemplo, memoria mostrada, información guardada, errores). La UI puede simplemente mostrar esos tal cual en el chat, o destacar de algún modo diferente (pero no es obligatorio).

- **Tema visual configurable**: Se puede permitir cambiar entre modo claro/oscuro, o colores, aunque no es un requisito, sería parte de "configurable". Con CSS moderno (por ejemplo TailwindCSS o simplemente variables CSS) esto es fácil de habilitar.

Comunicación Frontend-Backend

En Tauri, la comunicación con el backend Python local puede realizarse de varias formas:

- **Llamadas HTTP locales**: Dado que arrancamos un servidor FastAPI en `127.0.0.1:8000`, la aplicación web puede hacer peticiones fetch a `http://127.0.0.1:8000/...`. Para streaming, se utilizará la API de `EventSource`de JavaScript para conectarse a `/chat_stream`.

- **Invocar comandos Tauri**: Otra forma (especialmente si en lugar de FastAPI hubiéramos embebido Python) sería usar `window.__TAURI__.invoke` para llamar funciones. Sin embargo, dado que ya tenemos un servidor HTTP, es más estándar usar este.

- **WebSockets**: FastAPI también soporta WebSockets; se podría usar para streaming en vez de SSE. En este caso SSE es suficiente y más simple con EventSource.

Flujo típico de interacción:

1. El usuario escribe un mensaje "¿Qué puedo tomar para el dolor de cabeza?" y pulsa enviar.
2. La UI lee el texto. Antes de enviarlo, revisa si hay que forzar Nous-Hermes (por ejemplo, si pulsó el botón boost, puede añadir `&model=Nous-Hermes` en la query) y luego:
    - Crea un EventSource: `const evtSrc = new EventSource('http://127.0.0.1:8000/chat_stream?message=...&modelOverride=...');`
    - Añade manejadores:
      `js`
      CopiarEditar

```
evtSrc.onmessage = (event) => {
    if (event.data === "[DONE]") {
        evtSrc.close();
    } else {
        // Append event.data to the current response text being displayed
        updateAssistantMessage(event.data);
    }
```

    - `};`
    - `evtSrc.onerror = (error) => { /* manejar error, quizá reintentar o mostrar mensaje */ };`
    -

    - Inmediatamente en la UI, agrega el mensaje del usuario al chat con estilo de usuario.
    - Añade un nuevo mensaje de ALFIE al chat (estilo asistente) pero inicialmente vacío o con un indicador "Escribiendo...".
    - A medida que llegan datos en `onmessage`, va concatenándolos al contenido del último mensaje de ALFIE en pantalla, de manera que el usuario vea cómo se va escribiendo la respuesta.
    - Cuando recibe `[DONE]`, elimina el indicador de escritura o cierra la conexión.

3. Si el usuario envió un comando interno (por ej. `@memoria`), el backend no hace streaming sino que devuelve toda la respuesta de inmediato (por SSE igual se manda como un bloque seguido de `[DONE]`). La UI en este caso recibiría ese bloque, lo pone como mensaje de ALFIE (por ejemplo mostrando la lista de memoria). Como está todo junto, aparecerá de golpe.

4. Los controles de modelo/tono pueden hacer peticiones `fetch` al endpoint `/config` (POST) pasando el nuevo valor. Al completar, podrían actualizar alguna indicación (aunque como el estado ya se cambió en backend y en la UI misma al seleccionar, normalmente basta con la UI local).

5. El panel de memoria al guardar hará un `fetch('/memory', {method:'POST', body: JSON.stringify({content: newText})})`. Tras éxito, se podría refrescar la vista de memoria o simplemente cerrar el modal. La próxima vez que el usuario consulte `@memoria` o se genere una respuesta, esa memoria actualizada estará en uso.

Integración con Tauri:

- En el proyecto Tauri (generalmente la carpeta `src-tauri` contiene `tauri.conf.json` y archivos Rust), se debe indicar que el binario backend (si queremos distribuirlo con la app) es un *sidecar*. Esto para que Tauri lo incluya en el bundle y pueda lanzarlo. Por ejemplo, en `tauri.conf.json` se agregaría:
  `json`
  CopiarEditar

  ```
  {
  ```

- ```
    "tauri": {
      "bundle": {
        "externalBin": [
          "path/to/python/backend_executable"
  ```
  ```

```
          ]
        },
        "allowlist": {
          "shell": {
            "sidecar": true
          }
        }
      }
    }
```

    - Otra opción es compilar el Python a ejecutable con PyInstaller o similar, y referenciarlo.
    - Como nosotros podemos instalar el backend separadamente, incluso se puede asumir que el servicio estará corriendo (vía LaunchAgent) antes de abrir la app, en cuyo caso no es necesario que Tauri lo lance.
- Durante el desarrollo, se puede correr `npm run tauri dev` para levantar la UI y a la vez tener el backend corriendo con `python main.py`. En producción, la app Tauri final deberá asegurarse de que el backend esté activo:
    - O bien incluyéndolo y lanzándolo en background al abrir la app (sidecar).
    - O confiando en que está como servicio autoiniciado. Una ventaja de LaunchAgent es que el backend ya estaría activo esperando, así la UI se conecta instantáneamente.
    - Incluso se podrían implementar verificaciones: si la UI no consigue conectarse al backend en `localhost:8000`, podría intentar lanzar el sidecar o mostrar un mensaje de error "Backend no disponible".

Estilos y Librerías UI:

- Se puede usar React con componentes funcionales y hooks para manejar el estado de la conversación, o Svelte con su reactividad simplificada. Cualquiera permite crear la interfaz descrita.
- Para un diseño atractivo pero minimalista, se puede usar Tailwind CSS (muy conveniente en Tauri) o Bulma, o simplemente escribir CSS personalizado enfocándose en la simplicidad:
    - Colores suaves, tipografía legible (ej. fuentes sans-serif modernas).
    - Modo oscuro con fondo gris oscuro y texto claro, modo claro con fondo muy claro y texto oscuro.
    - Burbujas de chat con border-radius, quizás una sombra ligera.
    - Scrollbar personalizada para que no sea muy ancha, etc.
    - Los controles (select, botones) con estilo nativo o custom.

Ejemplo de estructura de frontend (React):

```
php
CopiarEditar
alfie-frontend/
├── src/
│   ├── App.jsx        # Componente principal
│   ├── components/
│   │   ├── Chat.jsx   # Muestra la lista de mensajes
│   │   ├── InputBar.jsx # Campo de texto + botón enviar
│   │   ├── Controls.jsx # Modelo selector, tono selector,
etc.
│   │   └── MemoryModal.jsx # Modal para editar memoria
│   ├── styles/
│   │   └── App.css    # Estilos generales
│   └── ... (archivos de config)
├── public/
│   └── index.html
└── src-tauri/
    ├── tauri.conf.json
    └── Cargo.toml (y código Rust de bootstrap, minimal)
```

En este ejemplo, `App.jsx` mantiene el estado de la conversación (una lista de mensajes), el estado de los controles (modelo seleccionado, tono, etc., que puede inicializarse llamando a `/config` GET si hubiera, o con valores por defecto `DEFAULT_MODEL`, etc.). `Chat.jsx` se encarga de renderizar las burbujas. `InputBar.jsx` contiene el textarea y el botón enviar, y gestiona crear la EventSource al enviar, y pasar los datos entrantes a `App.jsx` para actualizar el último mensaje. `Controls.jsx` tendría select inputs vinculados a funciones que hagan fetch al backend (o usar `invoke` con comandos, pero aquí usamos HTTP) para actualizar la config. `MemoryModal.jsx` al abrir hace fetch `/memory` para obtener contenido, lo muestra en un `<textarea>`, y al dar guardar, llama al POST `/memory` con el texto.

Todas las comunicaciones son locales, rápidas y no hay latencia de red real, por lo que la experiencia debe ser fluida. Además, los modelos corriendo en Apple Silicon (especialmente un Mac Mini M2/M3) pueden responder en segundos para preguntas típicas, dependiendo del tamaño del modelo:

- Modelos pequeños (Phi 3.5B) pueden ser casi instantáneos para respuestas cortas.
- Modelos medianos (Mistral 7B, Alfie si es ~7B) tardarán unos pocos segundos por párrafo de respuesta.
- Nous-Hermes 13B tardará más (varios segundos para respuestas largas), por eso tener la opción de usarlo solo cuando se necesite más potencia evita ralentizar cada interacción.

Seguridad y privacidad

Como ALFIE corre en local, **no envía ninguna información a internet**. Toda la conversación, incluidos datos médicos sensibles, permanecen en la máquina del usuario. Esto es crítico para

entornos médicos o de salud donde la privacidad es esencial. Aún así, es importante señalarle al usuario que **ALFIE no sustituye consejo médico profesional**; es un asistente que brinda información en base a sus datos entrenados, los cuales pueden estar actualizados hasta cierto punto. (Sería recomendable incorporar en el prompt de sistema alguna indicación para que el asistente sugiera consultar a un médico en dudas críticas, etc., pero eso puede manejarse en la afinación del modelo o prompts).

Inicialización Automática en macOS (LaunchAgent)

Para que ALFIE esté siempre disponible, podemos configurar el backend para que se inicie automáticamente al iniciar sesión en macOS. Esto se logra creando un **LaunchAgent**. Un LaunchAgent es un archivo de configuración `.plist`que indica al sistema que lance un determinado programa en background en el arranque del usuario.

Supongamos que el proyecto se despliega en `/Applications/ALFIE` (o en el home `~/ALFIE`). Crearemos un archivo de LaunchAgent en `~/Library/LaunchAgents/com.alfie.assistant.plist` con contenido similar a:

xml
CopiarEditar

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
   "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <!-- Nombre único del LaunchAgent -->
    <key>Label</key>
    <string>com.alfie.assistant</string>

    <!-- Programa a ejecutar (Python + script main.py) -->
    <key>ProgramArguments</key>
    <array>
      <string>/usr/local/bin/python3</string>
      <string>/Users/miusuario/ALFIE/backend/main.py</string>
    </array>

    <!-- Iniciar al cargar (login) -->
    <key>RunAtLoad</key>
    <true/>

    <!-- Reiniciar automáticamente si finaliza
inesperadamente -->
    <key>KeepAlive</key>
    <true/>

    <!-- Directorio de trabajo (por si el script asume rutas
relativas) -->
```

```
    <key>WorkingDirectory</key>
    <string>/Users/miusuario/ALFIE/backend</string>

    <!-- Opcional: Rutas para stdout y stderr (logs) -->
    <key>StandardOutPath</key>
    <string>/Users/miusuario/ALFIE/logs/alfie_agent.out</
string>
    <key>StandardErrorPath</key>
    <string>/Users/miusuario/ALFIE/logs/alfie_agent.err</
string>
  </dict>
</plist>
```

Instrucciones para activar el LaunchAgent:

1. Guardar el contenido anterior en el archivo `com.alfie.assistant.plist` dentro de `~/Library/LaunchAgents/`.
2. Abrir Terminal y ejecutar:
   bash
   CopiarEditar

   ```
   launchctl load ~/Library/LaunchAgents/
   com.alfie.assistant.plist
   ```
3.

   Esto carga el agente. A partir de ahora, cada vez que el usuario inicie sesión, macOS ejecutará el comando indicado (en este caso, el backend de ALFIE).
4. Verificar que el servicio esté corriendo:
   bash
   CopiarEditar

   ```
   launchctl list | grep alfie
   ```
5.

   Debe listar `com.alfie.assistant` como cargado. Si no aparece, revisar si hay errores en la configuración (los logs de error especificados pueden ayudar).

Con esta configuración, el backend siempre estará activo en segundo plano. Cuando el usuario abra la aplicación ALFIE (frontend Tauri), podrá conectarse al backend de inmediato. Incluso es posible interactuar con ALFIE vía otros medios (por ejemplo, en el futuro se podría tener un pequeño cliente CLI o comandos de voz que se comuniquen con este mismo backend).

Nota: Para desactivar el LaunchAgent, se puede descargar con `launchctl unload ...` o eliminar el plist. Si se actualiza el backend (por ejemplo, una nueva versión del código), conviene reiniciar el servicio (`launchctl stop com.alfie.assistant` y `launchctl start com.alfie.assistant` o simplemente logout/login de nuevo).

Extensiones Futuras y Consideraciones Finales

El proyecto ALFIE, tal como se ha desarrollado, es ya un asistente médico funcional local. Sin embargo, su arquitectura modular prepara el camino para **múltiples extensiones futuras**:

- **Entrada/Salida de Voz**: Integrar un módulo de voz permitiría hablar con ALFIE. Por ejemplo, usando bibliotecas locales:
    - *Reconocimiento de voz (Speech-to-Text)*: en Apple Silicon se podría usar la API de reconocimiento de voz de macOS o modelos offline como Coqui STT para convertir la voz del usuario en texto, el cual se enviaría al backend igual que si se hubiera escrito.
    - *Síntesis de voz (Text-to-Speech)*: para que ALFIE lea sus respuestas. macOS ofrece voces integradas (AVSpeechSynthesizer) o se podrían usar modelos TTS offline. ALFIE podría tener un botón "🎤" para escuchar la pregunta y luego leer la respuesta en voz alta.
    - Estos componentes podrían comunicarse con el mismo ChatManager (por ejemplo, la app Tauri podría manejar audio, o un módulo Python podría suscribirse).
- **Soporte de Imágenes**: Como asistente médico, podría ser útil analizar imágenes (por ejemplo, informes médicos escaneados, radiografías, etc.). ALFIE podría incorporar en el futuro un módulo de visión:
    - Esto podría ser vía un modelo multimodal (como *LLaVA* o *MiniGPT-4* adaptado) integrado localmente, o herramientas de análisis específicas.
    - La infraestructura actual no se ocupa de imágenes, pero se podría extender la interfaz para adjuntar una imagen y enviar un comando `@imagen ruta` o similar que el backend reconozca y procese con un modelo de visión, devolviendo luego una descripción o diagnóstico asistido.
- **Integración de sensores/IoT**: En un entorno doméstico, ALFIE podría interactuar con dispositivos (por ejemplo, obtener la lectura de un glucómetro conectado, o la temperatura ambiente). Con su naturaleza autónoma, podría haber un módulo que recoja datos de ciertos sensores disponibles vía API locales y los incluya en el contexto (por ejemplo, en la memoria persistente con actualizaciones constantes: "Temperatura corporal: 37.1°C").
- **Plugins de Herramientas**: Similar a cómo ChatGPT Plugins o LangChain funcionan, ALFIE podría tener comandos para utilizar herramientas:
    - Por ejemplo `@buscar <término>` que haga una búsqueda rápida en una base de conocimientos médica local (o en internet si en el futuro se permitiera momentáneamente conexión, aunque la filosofía actual es offline).

    - `@calcular <expresión>` para cálculos matemáticos (p. ej., dosificación basada en peso).

    - Estas herramientas se pueden implementar detectando ciertos patrones en `handle_user_message` antes de enviar al LLM, usando Python para obtener el resultado, y luego formando una respuesta. La estructura de comandos internos ya facilita esto.

- **Mejora de aprendizaje automático**: Si bien ALFIE no envía datos a la nube, podría guardar las conversaciones (con consentimiento) de forma que un desarrollador mejore el modelo 'alfie' periódicamente con fine-tuning en base a preguntas reales de usuarios (haciendo un bucle de mejora local). También se podría implementar un modo "entrenamiento" local si el hardware lo permite, pero eso es más complejo. Por ahora, *modo aprendizaje* se refiere a la memoria de la conversación.

- **Multi-usuario o chat múltiple**: Actualmente, se asume un usuario principal (memoria única). En el futuro, la UI podría permitir manejar diferentes perfiles o sesiones, cada una con su memoria (por ejemplo, perfil para distintos pacientes, si un médico lo usara con varios pacientes). Bastaría con extender `MemoryManager` para manejar diferentes archivos de memoria según un ID de sesión, y permitir en la UI elegir o iniciar una nueva sesión de chat.

- **Seguridad adicional**: Si ALFIE se usa en un entorno compartido, se podría poner un *PIN* o contraseña para acceder a la memoria o ciertas funciones (ya que es local, cualquiera con acceso al equipo podría ver la memoria). Implementar cifrado de la memoria en disco o requerir autenticación para abrir la app son consideraciones posibles.

Finalmente, la **despliegue** del sistema ALFIE sería así organizado:

```
bash
CopiarEditar
ALFIE/
├── backend/                  # Código backend Python
│   ├── main.py
│   ├── chat_manager.py
│   ├── memory.py
│   ├── config.py
│   ├── utils.py
│   └── __pycache__/ ...      # (archivos compilados, ignorar)
│
├── models/                   # (opcional) Archivos de modelos
locales si se usan fuera de Ollama
│   └── ...
│
├── logs/
│   ├── alfie.log             # Registro de eventos
│   ├── alfie_agent.out       # Salida standard del servicio
(si LaunchAgent la redirige aquí)
```

```
│   └── alfie_agent.err      # Errores del servicio (por
LaunchAgent)
│
├── alfie_memory.txt         # Archivo de memoria persistente
del asistente
│
├── frontend/                # Código fuente de la interfaz
(React/Svelte)
│   ├── src/
│   ├── public/
│   └── package.json etc.
│
└── src-tauri/               # Configuración Tauri y código
Rust de arranque
    ├── tauri.conf.json
    └── Cargo.toml, main.rs, etc.
```

Instrucciones de despliegue resumidas:

1. **Instalar dependencias base**:
   - Python 3.9+ (idealmente 3.10 o superior) y pip.
   - Node.js (para construir el frontend Tauri) y Rust (Tauri requiere el toolchain de Rust instalado).
   - Ollama CLI para MacOS, descargar desde la web oficial o via Homebrew (`brew install ollama`).
2. **Instalar modelos en Ollama**: En Terminal, ejecutar por lo menos:
   - `ollama pull alfie` (asumiendo que el modelo personalizado alfie ha sido publicado o existe localmente; si es local, usar `ollama create` o `ollama run` con el modelo custom).
   - `ollama pull mistral` (o nombre exacto de modelo Mistral, p. ej. `mistral-7b-v0.1`).
   - `ollama pull phi-3.5` (si ese es el identificador).
   - `ollama pull nous-hermes` (puede ser `nous-hermes-13b` según versión).
   - *Nota:* Cada modelo puede tardar en descargarse, y ocupará espacio en disco. Asegurarse de tener espacio suficiente.
3. **Configurar entorno Python**: Crear un entorno virtual (recomendado) y `pip install fastapi uvicorn[standard] ollama`. Esto instala FastAPI, Uvicorn (servidor ASGI) y la librería de Ollama para Python.

4. **Probar backend**: Ejecutar `python backend/main.py`. Verificar en la consola los logs (debería indicar que el servidor arranca en 127.0.0.1:8000). Probar enviando una request sencilla: por ejemplo con curl:
   `bash`
   CopiarEditar

   ```
   curl -X POST -H "Content-Type: application/json" -d
   '{"message": "Hola"}' http://127.0.0.1:8000/chat
   ```

5.

   Debería devolver un JSON con una respuesta (posiblemente un saludo de ALFIE). Esto confirma que el backend y el modelo por defecto funcionan.
6. **Configurar frontend**:
   - Entrar en `frontend/`, instalar dependencias (`npm install`).
   - Ajustar si es necesario en el código JS la URL del backend (en desarrollo puede ser `http://localhost:8000`, en producción Tauri se puede mantener igual ya que es local).
   - `npm run tauri dev` para levantar la app en modo desarrollo. Se debería abrir la ventana con la interfaz. Probar a escribir algo y recibir respuesta.
   - Ajustar estilos o funcionalidad si hay fallos. Asegurarse que los selectores de modelo/tono funcionan (inspeccionando el log backend se ve cambios de modelo/tono).
   - Cuando todo esté correcto, `npm run tauri build` para generar el binario de la aplicación (en `src-tauri/target`).
   - Confirmar que en la configuración Tauri de bundling se incluyó el sidecar del backend si se desea (o planear instalarlo por separado con LaunchAgent).
7. **Implementar LaunchAgent (opcional)**: Como indicado, crear el `.plist` en LaunchAgents. Esto es opcional si preferimos que el backend se lance junto con la app Tauri. Sin embargo, autoiniciar tiene la ventaja de que ALFIE puede eventualmente ejecutarse incluso sin abrir la interfaz (por ejemplo, podría tener atajos globales o activar al hablar "Oye ALFIE" si se implementa voz, etc.).
8. **Iniciar ALFIE**: Si se usa LaunchAgent, reiniciar sesión para que arranque, o lanzar manualmente con `launchctl start com.alfie.assistant`. Luego abrir la aplicación ALFIE (la UI). Debería conectar y estar lista para usar. Si no se usa LaunchAgent, simplemente abrir la app Tauri hará que arranque el backend (si está configurado como sidecar).
9. **Disfrutar de ALFIE**: Ya en funcionamiento, se puede conversar con el asistente. Probar características:

- Preguntar algo médico: ALFIE (modelo Alfie) responderá.
- Escribir `@memoria` para ver memoria (debería estar inicialmente vacía o con algún valor si se puso).
- Escribir `@recordar El usuario tiene 30 años.` y luego `@memoria` para ver que se añadió.
- Cambiar tono a Informal y preguntar algo para notar la diferencia en estilo.
- Probar el botón "Forzar Nous-Hermes": tras activarlo, la siguiente pregunta debería tomar más tiempo pero ser respondida más elaboradamente (y quizás con distinto estilo ya que el modelo es diferente). Luego, automáticamente volverá a usar Alfie (o el modelo que estuviera por defecto).
- Verificar los logs en `logs/alfie.log` si ocurre algo inesperado.

Con todos estos pasos, ALFIE queda desplegado de forma **autónoma**, **local** y **optimizada** para Apple Silicon, listo para asistir en dudas médicas de manera privada y configurable. Su arquitectura modular permite ir incrementando sus capacidades. Por ejemplo, para añadir voz, podríamos complementar el frontend con modales de grabación y reproducir audio (usando la API de sistema de Tauri para audio, o integrando un motor TTS). Para imágenes, añadir un botón de adjuntar imagen y un endpoint en backend que acepte archivos (FastAPI soporta envío de archivos fácilmente) y pasarlo a un modelo de visión.

ALFIE sirve como base para un asistente personalizable. Al ser local, el usuario tiene control total sobre sus datos. Además, correr en un Mac Mini u otro Apple Silicon aprovecha el hardware optimizado para ML. Este proyecto equilibra la **privacidad** con la **funcionalidad** de un asistente inteligente, y sienta las bases para características avanzadas en versiones futuras.