Remembrances: Diseño e Implementación de un Servidor de Memoria Persistente en Golang

I. El Plan Arquitectónico de Remembrances-MCP

Esta sección establece la visión de alto nivel para el proyecto Remembrances-MCP, delineando la estructura del sistema, el flujo de datos y los principios de diseño fundamentales que garantizan su robustez, modularidad y rendimiento. El sistema está diseñado no solo como un servidor para el Protocolo de Contexto de Modelo (MCP), sino como una plataforma de memoria inteligente y autónoma, inspirada en la sofisticada arquitectura de capas del proyecto Mem0.[1]

1.1. Arquitectura de Sistema de Alto Nivel

El diseño de Remembrances-MCP se fundamenta en una arquitectura de microservicios desacoplada, compuesta por cinco componentes primarios, cada uno con una responsabilidad claramente definida. Esta separación de incumbencias es crucial para la mantenibilidad, escalabilidad y flexibilidad del sistema.

1. **Capa de Transporte (Transport Layer):** Este es el punto de entrada para toda la comunicación externa. Su función es abstraer los detalles del protocolo de comunicación subyacente. Maneja las interacciones específicas del protocolo, ya sea MCP sobre E/S estándar (stdio), MCP sobre Eventos Enviados por el Servidor (SSE), o una API REST sobre HTTP. Esta capa es responsable de recibir las solicitudes en bruto, decodificarlas y pasarlas al componente apropiado.
2. **Núcleo MCP (MCP Core):** Construido sobre la librería mcp-go [3], este componente gestiona el ciclo de vida del servidor MCP. Sus responsabilidades incluyen la inicialización del servidor, el registro de las tools (herramientas) que se expondrán al agente de IA (como Claude o Cursor) [4] y el despacho de las solicitudes

callTool a los manejadores (handlers) correspondientes. Actúa como el orquestador principal para todas las interacciones basadas en MCP.

3. **Subsistema de Memoria (El "Cerebro"):** Este es el corazón lógico del sistema, implementado como un paquete de Go independiente y desacoplado. Aquí reside la lógica de negocio principal que emula el funcionamiento de Mem0. Es invocado tanto por el Núcleo MCP como por los manejadores de la API REST. Su responsabilidad es procesar las solicitudes de adición y búsqueda de memoria, orquestando la interacción con los LLMs y la capa de almacenamiento para implementar una memoria inteligente y multicapa.
4. **Capa de Almacenamiento (El "Banco de Memoria"):** Esta capa proporciona una interfaz de persistencia unificada, implementada exclusivamente con el SDK surrealdb.go.[6] Abstrae las operaciones de base de datos, ya sea contra una instancia de SurrealDB embebida en el propio binario o una instancia remota configurada a través de variables de entorno.[7] Su diseño se aprovecha de la naturaleza multi-modelo de SurrealDB para gestionar el almacenamiento clave-valor, vectorial y de grafos dentro de un único motor de base de datos.
5. **Gestión de Configuración y Ciclo de Vida:** Este componente, residente en el paquete main de la aplicación, es el punto de partida. Es responsable de analizar los flags de la línea de comandos [8], inicializar todos los demás componentes basándose en la configuración proporcionada y gestionar el ciclo de vida de la aplicación, incluyendo el inicio y la detención ordenada de los servidores y servicios.

Conceptualmente, una solicitud fluye a través de estas capas de manera secuencial. Por ejemplo, una llamada de herramienta MCP es recibida por la Capa de Transporte, procesada por el Núcleo MCP, que a su vez invoca la lógica en el Subsistema de Memoria. Este subsistema utiliza la Capa de Almacenamiento para persistir o recuperar datos, todo ello configurado y orquestado por el Gestor de Ciclo de Vida. Este diseño modular asegura que la lógica de memoria central pueda evolucionar independientemente de cómo se acceda a ella.

1.2. Flujo de Datos y Lógica: Un Recorrido Detallado

Para comprender la interacción dinámica entre los componentes, es útil analizar los flujos de datos para las operaciones de escritura (ingesta) y lectura (recuperación).

Flujo de Ingesta (Ej: Llamada a la herramienta add_memory)

1. **Recepción de la Solicitud:** Un cliente MCP, como el escritorio de Claude [4], envía una solicitud
   callTool para la herramienta add_memory a través del transporte configurado (p. ej., stdio).
2. **Procesamiento MCP:** La Capa de Transporte recibe los datos brutos y los pasa a la instancia del servidor mcp-go. El Núcleo MCP identifica la herramienta solicitada y despacha la llamada al manejador addMemoryHandler registrado.[3]
3. **Invocación del Subsistema de Memoria:** El manejador extrae los argumentos de la solicitud, como el contenido textual y el user_id, e invoca el método MemorySubsystem.Add(content, userId).
4. **Procesamiento Inteligente:** Dentro del Subsistema de Memoria, el "Procesador de Ingesta Inteligente" entra en acción. Utiliza langchaingo para realizar una llamada a un LLM (configurado para usar Ollama o una API compatible con OpenAI). El prompt está diseñado para emular la lógica de extracción de hechos de Mem0 [2], solicitando al LLM que identifique y extraiga hechos discretos, entidades y sus relaciones del texto de entrada, devolviéndolos en un formato estructurado como JSON.
5. **Estrategia de Almacenamiento:** Basándose en la respuesta del LLM, el procesador determina la estrategia de almacenamiento óptima para cada pieza de información. Un hecho simple como "al usuario no le gusta el queso" se destina al almacén clave-valor. El texto original y su embedding vectorial se preparan para el almacén semántico. Las relaciones identificadas, como "(Usuario:Alex) -> [escribió] -> (Documento:Plan)", se destinan al almacén de grafos.
6. **Persistencia de Datos:** El procesador invoca los métodos correspondientes en la Capa de Almacenamiento, como storage.SaveFact(), storage.IndexVector() o storage.CreateRelationship().
7. **Ejecución en la Base de Datos:** La Capa de Almacenamiento traduce estas llamadas en consultas SurrealQL específicas y las ejecuta utilizando el SDK surrealdb.go.[7]
8. **Respuesta:** Una vez que los datos se han persistido con éxito, una respuesta de confirmación se propaga hacia arriba a través de la cadena de llamadas, llegando finalmente al cliente MCP.

Flujo de Recuperación (Ej: Llamada a la herramienta search_memory)

El flujo de recuperación sigue una ruta similar, pero con una lógica de consulta híbrida:

1. La solicitud search_memory llega al manejador searchMemoryHandler.
2. El manejador invoca MemorySubsystem.Search(query, userId).
3. El Subsistema de Memoria primero genera un embedding vectorial de la consulta del usuario utilizando el servicio de embeddings.
4. A continuación, realiza una búsqueda híbrida y multifacética en la Capa de Almacenamiento:
    - **Búsqueda Vectorial:** Realiza una búsqueda de similitud semántica en la tabla vector_memories para encontrar los fragmentos de memoria más relevantes.
    - **Búsqueda en Grafo:** Extrae entidades clave de la consulta y de los resultados vectoriales, y las utiliza para realizar travesías en el grafo, descubriendo relaciones y contextos que no son evidentes semánticamente.
    - **Búsqueda Clave-Valor:** Realiza búsquedas directas de hechos específicos que puedan ser relevantes para la consulta.
5. **Síntesis de Resultados:** Los resultados de las tres búsquedas se recopilan y se pasan a un LLM con un prompt de síntesis. Este LLM genera una respuesta coherente y contextualizada que combina la información recuperada.
6. **Retorno de la Respuesta:** La respuesta sintetizada se devuelve al cliente MCP como texto plano.

1.3. Principios Fundamentales de Diseño

El diseño de Remembrances-MCP se rige por un conjunto de principios clave para asegurar su calidad y viabilidad a largo plazo.

- **Modularidad y Desacoplamiento:** El Subsistema de Memoria es el núcleo y está diseñado como un paquete Go con una interfaz bien definida. Esto permite que las capas de transporte (MCP, REST) y los clientes puedan ser intercambiados, actualizados o extendidos sin afectar la lógica de negocio central. Esta separación es fundamental para la reutilización y las pruebas unitarias.

- **Persistencia Unificada:** El uso de SurrealDB como único motor de persistencia es una decisión arquitectónica estratégica. Aprovechando sus capacidades multi-modelo, se evita la complejidad operativa de gestionar, sincronizar y mantener consistentes bases de datos separadas para clave-valor, vectores y grafos.[6] Esto simplifica drásticamente el despliegue y el mantenimiento.
- **Comportamiento Dirigido por Configuración:** El modo operativo completo de la aplicación (transporte, habilitación de APIs, uso de la base de conocimiento) se determina en el momento del arranque a través de flags de línea de comandos y variables de entorno. Esto confiere al sistema una alta adaptabilidad para diversos escenarios de despliegue, desde un entorno de desarrollo local hasta un servicio de producción a gran escala.
- **Servicios sin Estado, Base de Datos con Estado:** Los componentes de la aplicación Go están diseñados para ser sin estado (stateless). Toda la persistencia y el estado de la memoria a largo plazo son gestionados íntegramente por el motor de SurrealDB. Esto facilita la escalabilidad horizontal de la aplicación Go, ya que cualquier instancia puede atender cualquier solicitud sin necesidad de sincronización de estado entre ellas.

La combinación de una API REST opcional con el servidor MCP transforma el sistema en algo más que una simple herramienta para un agente de IA. Se convierte en un microservicio de memoria "headless" (sin cabeza) y accesible por red. Mientras que MCP está diseñado para la interacción directa con anfitriones como Claude [5], la API REST [15] permite que cualquier servicio de backend, script de automatización o aplicación externa aproveche el mismo almacén de memoria sofisticado. Esto amplía drásticamente la utilidad del sistema más allá del ecosistema MCP, posicionándolo como un componente de infraestructura de IA reutilizable. Por lo tanto, el Subsistema de Memoria debe ser tratado como un producto de primera clase, con una API bien documentada y potencialmente versionada.

Además, la elección de SurrealDB en modo *embebido* (--dbpath) permite crear una herramienta de IA completamente autocontenida en un único binario. Esta es una ventaja de despliegue monumental, especialmente para el desarrollo local o para casos de uso portátiles. El binario compilado de Go y su directorio de datos asociado constituyen todo el sistema, eliminando dependencias externas como contenedores Docker o servidores de bases de datos separados.[7] Este modelo de "dependencia cero" es un beneficio significativo derivado de la pila tecnológica elegida y debe ser destacado como una característica clave.

II. El Subsistema de Memoria Multicapa: El Motor Central

Esta sección constituye el núcleo técnico del informe, detallando la implementación de la arquitectura de memoria inteligente de Mem0 sobre la base de datos multi-modelo SurrealDB. Este subsistema es el responsable de la ingesta, el procesamiento, el almacenamiento y la recuperación de la información de manera contextual y eficiente.

2.1. Deconstruyendo el Modelo Mem0: Un Marco Conceptual

Un análisis de la arquitectura de Mem0, basado en su documentación y trabajos de investigación [2], revela una estrategia de almacenamiento híbrida y multicapa. Remembrances-MCP replicará este modelo conceptual utilizando SurrealDB como backend unificado.

- **Nivel 1: El Almacén de Hechos (Clave-Valor):** Este nivel está diseñado para almacenar información discreta, atómica y directamente recuperable. Es ideal para hechos y preferencias que no requieren interpretación semántica.
  - **Ejemplos:** "El nombre del usuario es Alex", "El usuario no le gusta el queso", "La preferencia de idioma del usuario es español".
  - **Implementación en SurrealDB:** Se utilizará una tabla kv_memories donde el id del registro puede ser una clave compuesta (p. ej., user:alex:preference:food) para una recuperación rápida y directa. Las operaciones se realizarán mediante surrealdb.Create y surrealdb.Select con IDs de registro específicos.[7]
- **Nivel 2: El Almacén Semántico (Vectorial/RAG):** Este nivel es para memorias que requieren una comprensión semántica del contenido. El texto original de interacciones importantes se convierte en un embedding vectorial y se almacena para búsquedas de similitud.
  - **Ejemplos:** Una consulta sobre "ideas para una cena saludable" debería recuperar una memoria de que "el usuario es vegetariano", aunque las palabras no coincidan. El sistema busca por significado, no por palabras clave.
  - **Implementación en SurrealDB:** Se usará una tabla vector_memories con un campo embedding de tipo array<float>. Se definirá un índice vectorial

especializado (p. ej., MTREE o HNSW) para acelerar las búsquedas de similitud.[20]

- **Nivel 3: El Almacén Relacional (Grafo):** Este nivel modela las conexiones y relaciones *entre* entidades y eventos a lo largo del tiempo. Es fundamental para el razonamiento complejo y las consultas de múltiples saltos (multi-hop).
    - **Ejemplos:** Modelar que (Usuario:Alex) --> (Documento:PlanDeProyecto) -[EN_FECHA]-> (Fecha:2024-08-15). Esto permite responder preguntas como "¿Qué documentos escribió Alex la semana pasada?".
    - **Implementación en SurrealDB:** Se definirán tablas para los nodos (p. ej., entities, users) y para las aristas (p. ej., wrote, mentioned_in). Las aristas en SurrealDB son tablas que contienen campos in y out obligatorios que apuntan a los nodos conectados.[14]

2.2. El Procesador de "Ingesta Inteligente"

Este componente es la implementación en Go de la lógica impulsada por LLM que decide qué almacenar y dónde. Es la función central del Subsistema de Memoria y replica el ciclo de extracción y actualización de Mem0.[11]

- **Paso 1: Extracción de Hechos y Entidades:** El procesador recibe el texto de la conversación en bruto. Utiliza langchaingo para invocar a un LLM (Ollama o compatible con OpenAI). El prompt está cuidadosamente diseñado, similar al FACT_RETRIEVAL_PROMPT de Mem0 [10], para instruir al LLM a extraer hechos clave, entidades nombradas y las relaciones entre ellas, devolviendo el resultado en un formato JSON estructurado y predecible.
- **Paso 2: Consolidación de Memoria:** Por cada hecho extraído en el paso anterior, el procesador realiza una búsqueda semántica en el almacén vectorial (vector_memories) para encontrar memorias existentes que sean similares en significado. Este paso es crucial para evitar la duplicación y para entender si la nueva información es realmente nueva, un complemento o una contradicción.
- **Paso 3: Determinación de la Acción (Lógica de "Llamada a Herramienta"):** Este es el núcleo de la "reconciliación de memoria". Basándose en el hecho extraído y en las memorias similares recuperadas, se realiza otra llamada al LLM. Esta vez, el LLM actúa como un "evaluador", decidiendo qué acción tomar sobre la base de conocimiento, tal como se describe en la investigación de Mem0.[35] Las posibles acciones son:
    - ADD: La información es completamente nueva y no existe una memoria

semánticamente equivalente.
    - UPDATE: La información complementa o refina una memoria existente (p. ej., añade detalles a un plan de viaje previamente mencionado).
    - DELETE: La información contradice directamente una memoria existente (p. ej., "Ahora me gusta el queso" contradice "No me gusta el queso").
    - NOOP: La información ya está presente y no se requiere ninguna acción.
- **Paso 4: Despacho al Almacenamiento:** Según la acción determinada y la naturaleza de los datos, el procesador invoca los métodos apropiados de la Capa de Almacenamiento.
    - Hechos simples y discretos se guardan en la tabla kv_memories.
    - El texto original de la interacción, junto con su embedding, se almacena en vector_memories para futuras búsquedas RAG.
    - Las relaciones identificadas entre entidades se utilizan para crear aristas en el grafo mediante sentencias RELATE de SurrealQL.

Este proceso de reconciliación de memoria es una desviación significativa de los sistemas RAG simples que solo añaden datos. Al implementar la lógica de UPDATE y DELETE basada en la evaluación del LLM, el sistema se vuelve auto-curativo, evitando la acumulación de información obsoleta o incorrecta. Esta es una característica crítica para la fiabilidad a largo plazo de un agente de IA. [35]

2.3. SurrealDB como Backend de Memoria Unificado

La elección de SurrealDB permite una implementación elegante de esta arquitectura de tres niveles en un único sistema de base de datos.

- **Inicialización:** La lógica de conexión a la base de datos es flexible. Primero, verificará la presencia de variables de entorno como SURREALDB_URL, SURREALDB_USER y SURREALDB_PASS para conectarse a una instancia remota. Si no se encuentran, recurrirá al flag --dbpath para iniciar una instancia embebida utilizando una URL de conexión de tipo file://.[7] Esto proporciona una gran flexibilidad para el despliegue.
- **Definiciones de Esquema y Tablas:** A continuación se presenta el conjunto completo de sentencias SurrealQL para definir el esquema de la base de datos. Este esquema es el plano para la capa de persistencia.

Componente	Declaración SurrealQL	Propósito
Tabla KV	DEFINE TABLE kv_memories SCHEMAFULL; DEFINE FIELD user_id ON kv_memories TYPE string; DEFINE FIELD key ON kv_memories TYPE string; DEFINE FIELD value ON kv_memories TYPE any; DEFINE FIELD created_at ON kv_memories TYPE datetime VALUE time::now(); DEFINE FIELD updated_at ON kv_memories TYPE datetime VALUE time::now() ON UPDATE time::now();	Almacena hechos discretos. id es la clave principal (p. ej., kv_memories:('user_id:pref_color')).
Tabla Vectorial	DEFINE TABLE vector_memories SCHEMAFULL; DEFINE FIELD user_id ON vector_memories TYPE string; DEFINE FIELD content ON vector_memories TYPE string; DEFINE FIELD embedding ON vector_memories TYPE array<float>; DEFINE FIELD metadata ON vector_memories TYPE object;	Almacena texto y sus embeddings para búsqueda semántica (RAG).
Índice Vectorial	DEFINE INDEX idx_embedding ON vector_memories FIELDS embedding MTREE DIMENSION 768 DIST COSINE;	Acelera la búsqueda de similitud. La DIMENSION debe coincidir con el modelo de embedding utilizado.[20]
Tabla de Nodos	DEFINE TABLE entities SCHEMAFULL; DEFINE FIELD name ON entities TYPE string; DEFINE FIELD type ON entities TYPE string;	Tabla genérica para nodos del grafo (p. ej., personas, lugares, conceptos).
Tabla de Aristas	DEFINE TABLE wrote SCHEMALESS; DEFINE FIELD in ON wrote TYPE record(entities); DEFINE FIELD	Representa una relación "escribió" entre dos entidades. Las aristas son

	out ON wrote TYPE record(entities); DEFINE FIELD timestamp ON wrote TYPE datetime;	tablas con campos in y out.[21]

- **Implementación de Operaciones:**
  - **Clave-Valor:** Las operaciones se implementan con los métodos de alto nivel del SDK, como surrealdb.Create(db, "kv_memories", data) y surrealdb.Select(db, recordID).[7]
  - **Vectorial (RAG):** Dado que el SDK de Go para SurrealDB no tiene una abstracción de alto nivel para la búsqueda vectorial (a diferencia de la librería de Python [23]), las búsquedas se deben construir manualmente con consultas SurrealQL en bruto. Esto se realiza con db.Query o db.RawQuery.[12] La consulta típica será:
    Code snippet
    ```
    SELECT id, content, vector::similarity::cosine(embedding, $query_embedding) AS similarity
    FROM vector_memories
    WHERE user_id = $user_id AND embedding <|k|> $query_embedding;
    ```

    Esta consulta utiliza el operador KNN (<|k|>) para encontrar los k vecinos más cercanos y la función vector::similarity::cosine para calcular la puntuación de similitud.[20]
  - **Grafo:** La creación de relaciones se realiza con sentencias RELATE ejecutadas a través de db.Query. Por ejemplo:
    Code snippet
    ```
    RELATE $user_id->wrote->$document_id CONTENT { timestamp: time::now() };
    ```

    La consulta de relaciones utiliza la sintaxis de travesía de grafos de SurrealQL: SELECT ->wrote->entities FROM entities:alex;.[13]

La ausencia de un SurrealDBVectorStore nativo en langchaingo presenta un desafío que se convierte en una oportunidad. Obliga a una implementación manual, pero esto permite construir una estrategia de recuperación híbrida y altamente optimizada. Se pueden combinar búsquedas vectoriales, travesías de grafos y búsquedas de clave-valor en una única y compleja consulta SurrealQL, algo que una abstracción genérica de almacén vectorial podría no soportar. Esto permite la creación de una función HybridSearch personalizada que encapsule esta lógica avanzada, superando

las capacidades de una simple búsqueda RAG.

III. Implementación del Servidor MCP y Herramientas

Esta sección detalla la construcción de la interfaz principal de la aplicación: el servidor MCP. Se aborda la inicialización del proyecto, la definición de las herramientas que el agente de IA podrá utilizar y la implementación de los manejadores que conectan estas herramientas con el Subsistema de Memoria.

3.1. Inicialización y Configuración del Proyecto

El punto de entrada de la aplicación (main.go) es responsable de configurar y poner en marcha todos los componentes del sistema basándose en los argumentos de la línea de comandos.

- **Estructura del Proyecto:** Se inicia con la creación de un nuevo módulo de Go: go mod init Remembrances-MCP.
- **Análisis de Flags:** Se utiliza el paquete estándar flag de Go para definir y analizar todos los parámetros de configuración.[8] Esto proporciona una forma robusta y convencional de controlar el comportamiento de la aplicación desde la línea de comandos.

A continuación se presenta la tabla de referencia de configuración, que sirve como guía principal para operadores y desarrolladores sobre cómo ejecutar y configurar la aplicación.

Flag	Variable de Entorno	Tipo	Descripción	Valor por Defecto
--sse	GOMEM_SSE	Booleano	Habilita el transporte mediante Server-Sent Events (SSE) en	false

			lugar de stdio.	
--rest-api-serve	GOMEM_REST_API_SERVE	Booleano	Habilita y expone el servidor de API REST en el puerto 8080.	false
--knowledge-base	GOMEM_KB_PATH	String	Ruta al directorio raíz de la base de conocimiento en formato Markdown. Activa la funcionalidad de KB.	"" (desactivado)
--dbpath	GOMEM_DB_PATH	String	Ruta al archivo/directorio para la base de datos SurrealDB embebida. Se ignora si se usan variables de conexión a DB remota.	"./gomem.db"
(ninguno)	SURREALDB_URL	String	URL de conexión para una instancia remota de SurrealDB (p. ej., ws://localhost:8000).	(ninguno)
(ninguno)	SURREALDB_USER	String	Nombre de usuario para la autenticación en SurrealDB.	(ninguno)
(ninguno)	SURREALDB_PASS	String	Contraseña para la autenticación	(ninguno)

			en SurrealDB.	
(ninguno)	OLLAMA_URL	String	URL del servidor Ollama para la generación de embeddings (p. ej., http://localhost:11434).	(ninguno)
(ninguno)	OPENAI_API_KEY	String	Clave de API para OpenAI o servicios compatibles.	(ninguno)
(ninguno)	OPENAI_API_BASE	String	URL base para APIs compatibles con OpenAI (p. ej., para usar LM Studio).	(ninguno)

El proceso en main.go sería:

1. Definir todas las variables para los flags.
2. Llamar a flag.Parse() para poblar estas variables desde los argumentos de la línea de comandos.
3. Leer las variables de entorno para sobreescribir o complementar la configuración.
4. Inicializar los servicios (base de datos, embedder, subsistema de memoria, etc.) con la configuración final.

3.2. Definición de Herramientas MCP

Las herramientas MCP (tools) son las funciones que el agente de IA (el LLM) puede "llamar". Se definen utilizando la función mcp.NewTool de la librería mcp-go, que genera un esquema que el LLM puede entender e invocar.[3]

- **Herramienta add_memory:**

```Go
addMemoryTool := mcp.NewTool(
    "add_memory",
    mcp.WithDescription("Añade una nueva pieza de información, turno de conversación o hecho a la memoria a largo plazo."),
    mcp.WithString(
        "content",
        mcp.Required(),
        mcp.Description("El contenido textual de la memoria a añadir."),
    ),
    mcp.WithString(
        "user_id",
        mcp.Required(),
        mcp.Description("El identificador único para el usuario asociado con esta memoria."),
    ),
)
```

- **Herramienta search_memory:**
```Go
searchMemoryTool := mcp.NewTool(
    "search_memory",
    mcp.WithDescription("Busca en la memoria información relevante para una consulta, proporcionando contexto para la conversación actual."),
    mcp.WithString(
        "query",
        mcp.Required(),
        mcp.Description("La consulta en lenguaje natural para buscar en la memoria."),
    ),
    mcp.WithString(
        "user_id",
        mcp.Required(),
        mcp.Description("El contexto de usuario para la búsqueda."),
    ),
)
```

- **Herramienta update_memory:**
```Go
updateMemoryTool := mcp.NewTool(
    "update_memory",
```

```
    mcp.WithDescription("Actualiza una memoria existente con nueva información."),
    mcp.WithString(
        "memory_id",
        mcp.Required(),
        mcp.Description("El ID único de la memoria a actualizar."),
    ),
    mcp.WithString(
        "content",
        mcp.Required(),
        mcp.Description("El nuevo contenido textual para la memoria."),
    ),
    mcp.WithString(
        "user_id",
        mcp.Required(),
        mcp.Description("El identificador de usuario para asegurar los permisos correctos."),
    ),
)
```

- **Herramienta delete_memory:**

```
Go
deleteMemoryTool := mcp.NewTool(
    "delete_memory",
    mcp.WithDescription("Elimina una memoria específica del sistema."),
    mcp.WithString(
        "memory_id",
        mcp.Required(),
        mcp.Description("El ID único de la memoria a eliminar."),
    ),
    mcp.WithString(
        "user_id",
        mcp.Required(),
        mcp.Description("El identificador de usuario para asegurar los permisos correctos."),
    ),
)
```

- **Herramientas de la Base de Conocimiento (si está habilitada):**

```
Go
// Para consulta (RAG)
queryKBTool := mcp.NewTool(
```

```
    "query_knowledge_base",
    mcp.WithDescription("Consulta la base de conocimiento estática del proyecto en formato Markdown para obtener información técnica, documentación o contexto del proyecto."),
    mcp.WithString(
        "query",
        mcp.Required(),
        mcp.Description("La consulta para buscar en la base de conocimiento."),
    ),
)

// Para escritura y actualización
updateKBTool := mcp.NewTool(
    "update_knowledge_base",
    mcp.WithDescription("Crea o actualiza un documento en la base de conocimiento del proyecto. El sistema lo re-indexará automáticamente."),
    mcp.WithString(
        "file_path",
        mcp.Required(),
        mcp.Description("La ruta relativa del archivo.md a crear o actualizar (p. ej., 'docs/nueva_funcionalidad.md')."),
    ),
    mcp.WithString(
        "content",
        mcp.Required(),
        mcp.Description("El contenido completo en formato Markdown para el archivo."),
    ),
)
```

3.3. Implementación de los Manejadores de Herramientas MCP

Cada herramienta definida necesita una función manejadora (handler) en Go que ejecute la lógica real. Estos manejadores actúan como puentes entre el protocolo MCP y el Subsistema de Memoria.

1. **Creación del Servidor:** Primero, se crea una instancia del servidor MCP:
   Go

```
s := server.NewMCPServer(
    "Remembrances-MCP Server",
    "1.0.0",
    server.WithToolCapabilities(true),
)
```

2. **Registro de Herramientas:** Luego, se asocia cada herramienta con su manejador:
   Go
   ```
   s.AddTool(addMemoryTool, addMemoryHandler)
   s.AddTool(searchMemoryTool, searchMemoryHandler)
   s.AddTool(updateMemoryTool, updateMemoryHandler)
   s.AddTool(deleteMemoryTool, deleteMemoryHandler)

   if knowledgeBaseEnabled {
       s.AddTool(queryKBTool, queryKnowledgeBaseHandler)
       s.AddTool(updateKBTool, updateKnowledgeBaseHandler)
   }
   ```

3. **Implementación de los Manejadores:**
   - **addMemoryHandler:**
     Go
     ```
     func addMemoryHandler(ctx context.Context, request mcp.CallToolRequest)
     (*mcp.CallToolResult, error) {
         content, err := request.RequireString("content")
         if err!= nil { return mcp.NewToolResultError(err.Error()), nil }
         userID, err := request.RequireString("user_id")
         if err!= nil { return mcp.NewToolResultError(err.Error()), nil }

         // memorySubsystem es la instancia inicializada del cerebro
         err = memorySubsystem.Add(ctx, content, userID)
         if err!= nil {
             return mcp.NewToolResultError("Failed to add memory: " + err.Error()), nil
         }

         return mcp.NewToolResultText("Memory added successfully."), nil
     }
     ```

   - **searchMemoryHandler:**

```
Go
func searchMemoryHandler(ctx context.Context, request mcp.CallToolRequest)
(*mcp.CallToolResult, error) {
    query, err := request.RequireString("query")
    if err!= nil { /*... */ }
    userID, err := request.RequireString("user_id")
    if err!= nil { /*... */ }

    result, err := memorySubsystem.Search(ctx, query, userID)
    if err!= nil { /*... */ }

    return mcp.NewToolResultText(result), nil
}
```

- Los manejadores para update, delete y las herramientas de la base de conocimiento seguirán un patrón similar, extrayendo los argumentos y llamando a los métodos correspondientes en sus respectivos subsistemas.

3.4. Inicio del Servidor

Finalmente, en main.go, una estructura de control basada en el flag --sse decidirá qué transporte iniciar.

```
Go

// Después de inicializar todos los servicios y registrar los handlers...

if *sse {
    // Iniciar servidor con Server-Sent Events
    log.Println("Starting MCP server with SSE transport on :8080")
    if err := server.ServeSSE(s, ":8080"); err!= nil {
        log.Fatalf("SSE Server error: %v\n", err)
    }
} else {
```

```
    // Iniciar servidor con stdio por defecto
    log.Println("Starting MCP server with stdio transport")
    if err := server.ServeStdio(s); err!= nil {
        log.Fatalf("Stdio Server error: %v\n", err)
    }
}
```

Si el flag de la API REST también está activo, se iniciará el servidor HTTP en una goroutine separada para que no bloquee el servidor MCP.

IV. Características Avanzadas: Base de Conocimiento y APIs

Esta sección cubre la implementación de las funcionalidades opcionales que extienden las capacidades del servidor más allá de la memoria conversacional, convirtiéndolo en una herramienta de contexto de proyecto más completa y versátil.

4.1. La Base de Conocimiento en Markdown (--knowledge-base)

Esta característica permite a Remembrances-MCP ingerir y servir como una base de conocimiento dinámica a partir de una colección de archivos Markdown (.md). Esto es ideal para proporcionar al agente de IA un contexto profundo sobre un proyecto específico, como documentación técnica, decisiones de arquitectura (ADRs) o guías de usuario, y permitirle actualizar ese conocimiento.

- **Lógica de Activación:** En main.go, si el valor del flag *kbPath no es una cadena vacía, se inicializa el servicio de la Base de Conocimiento. Se valida que la ruta proporcionada exista y sea un directorio.
- **Vigilancia de Archivos (File Watching):** Para mantener la base de conocimiento actualizada en tiempo real sin necesidad de reiniciar el servidor, se utilizará un vigilante de archivos. La librería github.com/fsnotify/fsnotify es una opción robusta y estándar en el ecosistema Go. Se configurará un vigilante para monitorear recursivamente el directorio *kbPath en busca de eventos de creación, modificación y eliminación de archivos con la extensión .md. Librerías como github.com/radovskyb/watcher son alternativas que funcionan mediante

sondeo y pueden ser más consistentes en sistemas de archivos de red.

- **Pipeline de Procesamiento:**
  1. **Disparo (Trigger):** Cuando el vigilante de archivos detecta un cambio (un archivo .md es creado o modificado), se dispara una función de procesamiento.
  2. **Carga del Documento:** La función lee el contenido del archivo Markdown. Se utiliza el paquete documentloaders de langchaingo para cargar el texto: documentloaders.NewText(reader).[26]
  3. **División del Texto (Chunking):** El texto cargado se divide en fragmentos (chunks) más pequeños y semánticamente coherentes. Para esto, se usará textsplitter.NewMarkdownTextSplitter de langchaingo.[27] Este divisor es superior a los divisores basados en caracteres o tokens, ya que entiende la estructura de Markdown y puede dividir el texto por encabezados (#, ##), párrafos, listas o bloques de código. Esto preserva el contexto local, lo cual es crucial para la calidad de los resultados de RAG.
  4. **Generación de Embeddings:** Cada fragmento de texto se pasa al Servicio de Embeddings para generar su representación vectorial.
  5. **Almacenamiento:** El fragmento de texto, su embedding vectorial y metadatos relevantes (como la ruta del archivo de origen y el encabezado de la sección) se almacenan en una tabla dedicada en SurrealDB, por ejemplo, knowledge_base. Esta tabla tendrá su propio índice vectorial para búsquedas eficientes. Si un archivo es modificado, sus entradas anteriores en la base de datos se eliminan y se reemplazan con las nuevas. Si un archivo es eliminado, todas sus entradas asociadas se purgan.
- **Consulta y Escritura:**
  - La herramienta MCP query_knowledge_base y su correspondiente endpoint REST realizarán una búsqueda vectorial exclusivamente contra esta tabla knowledge_base.
  - La nueva herramienta update_knowledge_base (y su endpoint REST) escribirá o sobrescribirá un archivo .md en el directorio vigilado. El vigilante de archivos se encargará automáticamente del proceso de re-indexación, manteniendo la lógica simple y desacoplada.

La implementación de esta característica da lugar a un flujo de trabajo muy potente, que se puede describir como "GitOps para el contexto de la IA". La documentación del proyecto, mantenida en un repositorio Git como archivos Markdown, puede ser la fuente directa de conocimiento del agente. Cuando un desarrollador actualiza la documentación y la empuja al repositorio, un pipeline de CI/CD puede desplegar los nuevos archivos en el directorio vigilado por Remembrances-MCP. El servidor

detectará los cambios automáticamente y actualizará su base de conocimiento. De este modo, el agente de IA siempre tendrá acceso a la información más reciente del proyecto, sincronizando de forma continua el conocimiento humano y el de la máquina.

4.2. La Capa de API REST (--rest-api-serve)

La API REST convierte a Remembrances-MCP en un microservicio de memoria independiente, accesible desde cualquier aplicación o servicio que pueda realizar solicitudes HTTP, desacoplándolo completamente del ecosistema MCP.

- **Lógica de Activación:** Si el flag *restApi es true, se lanza una nueva goroutine en main.go para iniciar un servidor HTTP utilizando el paquete estándar net/http.[15] Esto asegura que el servidor REST se ejecute concurrentemente con el servidor MCP sin bloquearlo.
- **Configuración del Enrutador (Router):** Se utilizará http.HandleFunc para mapear las rutas de la URL a las funciones manejadoras correspondientes. Para una API más compleja, se podría considerar una librería de enrutamiento ligera como gorilla/mux o chi, pero para los requisitos actuales, net/http es suficiente.
- **Especificación de los Endpoints de la API:** La siguiente tabla detalla el contrato de la API, sirviendo como documentación fundamental para cualquier desarrollador que desee integrarse con el sistema.

Endpoint	Método HTTP	Descripción	Cuerpo de la Solicitud (JSON)	Respuesta Exitosa (Código y Cuerpo)
/api/v1/memory	POST	Añade una nueva memoria al sistema.	{"content": "...", "user_id": "..."}	202 Accepted - {"status": "memory received for processing"}
/api/v1/memory/search	POST	Busca en la memoria conversacional.	{"query": "...", "user_id": "..."}	200 OK - {"results": "..."}
/api/v1/memory/{	PUT	Actualiza una memoria	{"content": "..."}	200 OK - {"status":

id}		existente.		"memory updated"}
/api/v1/memory/{id}	DELETE	Elimina una memoria específica.	(ninguno)	204 No Content
/api/v1/kb/query	POST	Consulta la base de conocimiento estática. (Solo si --knowledge-base está activo).	{"query": "..."}	200 OK - {"results": [{"content": "...", "source": "...", "score": 0.9}]}
/api/v1/kb/document	PUT	Crea o actualiza un documento en la KB. (Solo si --knowledge-base está activo).	{"file_path": "...", "content": "..."}	202 Accepted - {"status": "document update accepted"}
/api/v1/health	GET	Comprueba el estado del servicio.	(ninguno)	200 OK - {"status": "ok", "database": "connected"}

- **Manejo de Solicitudes y Respuestas:** Los manejadores serán responsables de:
  1. Decodificar las solicitudes JSON entrantes en structs de Go.
  2. Invocar los métodos correspondientes del Subsistema de Memoria o del servicio de la Base de Conocimiento.
  3. Codificar las respuestas (o errores) en formato JSON y escribirlas en el http.ResponseWriter, estableciendo el código de estado HTTP y el Content-Type adecuados.

V. El Servicio de Embeddings

Este componente es crítico, ya que es el responsable de convertir el texto en representaciones numéricas (vectores) que los algoritmos de búsqueda semántica pueden procesar. Su diseño se centra en la flexibilidad y la modularidad para soportar diferentes proveedores de embeddings.

5.1. Una Abstracción para los Embeddings

Para desacoplar la lógica de negocio del proveedor de embeddings específico, se define una interfaz de Go. Esta abstracción permite cambiar entre Ollama, OpenAI o cualquier otro proveedor futuro con un cambio mínimo en la configuración, sin alterar el código que utiliza el servicio.

Go

```
package embedder

import (
    "context"
)

// Embedder define la interfaz para cualquier servicio que pueda crear
// embeddings vectoriales a partir de texto.
type Embedder interface {
    // EmbedDocuments crea embeddings para un lote de textos.
    EmbedDocuments(ctx context.Context, textsstring) (float32, error)

    // EmbedQuery crea un embedding para un único texto (una consulta).
    EmbedQuery(ctx context.Context, text string) (float32, error)

    // Dimension devuelve la dimensionalidad de los vectores generados.
    // Es crucial para configurar dinámicamente los índices vectoriales.
    Dimension() int
}
```

Esta interfaz se inspira directamente en la de langchaingo [28], pero con la adición explícita del método

Dimension(). Este método es fundamental para que el sistema pueda configurar dinámicamente el índice vectorial en SurrealDB con la dimensionalidad correcta del

modelo de embedding que se esté utilizando, evitando así errores de configuración manual.

5.2. Implementación para Ollama

Se creará una implementación concreta para Ollama, que permite el uso de modelos de embedding de código abierto ejecutándose localmente.

- **Estructura:** Se definirá un struct OllamaEmbedder que implemente la interfaz Embedder.
- **Cliente langchaingo:** Utilizará el cliente de Ollama proporcionado por langchaingo, específicamente del paquete llms/ollama. La investigación confirma que langchaingo tiene soporte directo para Ollama.[29]
- **Constructor:** La función NewOllamaEmbedder recibirá como parámetros la URL del servidor Ollama y el nombre del modelo a utilizar (p. ej., "nomic-embed-text", "mxbai-embed-large"). Estos valores se obtendrán de variables de entorno como OLLAMA_URL y OLLAMA_EMBEDDING_MODEL.
- **Implementación de Métodos:** Los métodos de la interfaz (EmbedDocuments, EmbedQuery, Dimension) envolverán las llamadas a las funciones correspondientes del cliente de langchaingo. El valor de Dimension puede ser pre-configurado o, si es posible, consultado al modelo.

5.3. Implementación para APIs Compatibles con OpenAI

Esta implementación permitirá la conexión con los servicios de OpenAI o cualquier otro proveedor que ofrezca una API compatible (como LM Studio, Together AI, etc.).

- **Estructura:** Se definirá un struct OpenAIEmbedder que también implemente la interfaz Embedder.
- **Cliente langchaingo:** Utilizará el cliente de OpenAI de langchaingo del paquete llms/openai.[32]
- **Constructor:** La función NewOpenAIEmbedder se configurará a través de variables de entorno estándar como OPENAI_API_KEY, OPENAI_API_BASE (crucial para la compatibilidad con otros proveedores) y OPENAI_EMBEDDING_MODEL.[33]
- **Implementación de Métodos:** Los métodos envolverán las funciones del

paquete embeddings de langchaingo, como embeddings.NewEmbedder.[28] El constructor del cliente de OpenAI en
langchaingo permite especificar la BaseURL, lo que facilita apuntar a endpoints que no son de OpenAI pero que mantienen la compatibilidad de la API.

5.4. Instanciación Dinámica

Para seleccionar qué Embedder utilizar en tiempo de ejecución, se creará una función de fábrica (factory function) en el paquete de configuración o en main.go. Esta función inspeccionará las variables de entorno y decidirá qué implementación instanciar. Por ejemplo:

Go

```
func NewEmbedderFromConfig(cfg *Config) (embedder.Embedder, error) {
    if cfg.OllamaURL!= "" {
        log.Println("Initializing Ollama embedder...")
        return embedder.NewOllamaEmbedder(cfg.OllamaURL, cfg.OllamaModel)
    }
    if cfg.OpenAIAPIKey!= "" {
        log.Println("Initializing OpenAI-compatible embedder...")
        return embedder.NewOpenAIEmbedder(cfg.OpenAIAPIKey, cfg.OpenAPIBaseURL,
cfg.OpenAIModel)
    }
    return nil, errors.New("no embedder configuration found")
}
```

Este enfoque asegura que el resto de la aplicación interactúe únicamente con la interfaz Embedder, permaneciendo agnóstica a la implementación concreta.

VI. Estructura del Proyecto, Despliegue y Documentación

Esta sección final proporciona el andamiaje necesario para convertir el código en un proyecto de software mantenible, desplegable y fácil de usar para otros desarrolladores.

6.1. Diseño Recomendado del Proyecto

Una estructura de directorios clara y convencional es fundamental para la navegabilidad y mantenibilidad del código en un proyecto de Go. Se propone la siguiente estructura:

```
/Remembrances-MCP
├── /cmd/Remembrances-MCP/         # Punto de entrada principal de la aplicación
│   └── main.go
├── /internal/              # Lógica de aplicación privada, no exportable
│   ├── /config/            # Carga y análisis de la configuración
│   ├── /memory/            # El Subsistema de Memoria central ("el cerebro")
│   ├── /storage/           # La capa de persistencia con SurrealDB
│   ├── /transport/         # Manejadores para REST y otros transportes no MCP
│   └── /kb/                # Lógica de la base de conocimiento (vigilancia de archivos, etc.)
├── /pkg/                   # Librerías públicas (si las hubiera)
│   ├── /mcp_tools/         # Definiciones de las herramientas MCP
│   └── /embedder/          # La interfaz del embedder y sus implementaciones
├── go.mod
├── go.sum
└── README.md
```

Esta estructura separa claramente el punto de entrada (cmd), la lógica de negocio interna (internal) y las librerías potencialmente reutilizables (pkg), siguiendo las mejores prácticas de la comunidad Go.

6.2. Instrucciones de Compilación y Ejecución

Una guía paso a paso para que un desarrollador pueda poner en marcha el proyecto:

1. **Prerrequisitos:**
   - Instalación de Go (versión 1.18+).
   - (Opcional) Instalación de Ollama y descarga de un modelo de embedding (p. ej., ollama pull nomic-embed-text).
   - (Opcional) Una clave de API de OpenAI o servicio compatible.
2. **Clonación e Instalación:**
   Bash
   ```
   git clone <url_del_repositorio>
   cd Remembrances-MCP
   go mod tidy
   ```
3. **Configuración:**
   - Exportar las variables de entorno necesarias (p. ej., export OPENAI_API_KEY="...").
4. **Compilación:**
   Bash
   ```
   go build -o Remembrances-MCP./cmd/Remembrances-MCP
   ```
5. **Ejecución:** Se proporcionarán ejemplos para cada modo de operación:
   - **Modo Stdio por defecto:** ./Remembrances-MCP
   - **Modo SSE:** ./Remembrances-MCP --sse
   - **Con API REST:** ./Remembrances-MCP --rest-api-serve
   - **Con Base de Conocimiento:** ./Remembrances-MCP --knowledge-base=./docs
   - **Combinado:** ./Remembrances-MCP --rest-api-serve --knowledge-base=./docs

6.3. El Archivo README.md

Se generará un archivo README.md completo y listo para usar, que servirá como la

puerta de entrada al proyecto. Incluirá:

- **Remembrances-MCP:** Un resumen del proyecto, su propósito y su arquitectura inspirada en Mem0.
- **Características Principales:** Una lista con viñetas de las funcionalidades clave (memoria multicapa, RAG, grafo, base de conocimiento en Markdown, API REST, etc.).
- **Prerrequisitos:** Software necesario para compilar y ejecutar el proyecto.
- **Instalación y Compilación:** Las instrucciones detalladas de la sección anterior.
- **Uso:** Una explicación exhaustiva de todos los flags de línea de comandos y variables de entorno, posiblemente incluyendo la tabla de configuración de la sección 3.1.
- **Referencia de la API:** Un resumen de los endpoints de la API REST, con ejemplos de curl, haciendo referencia a la tabla de especificación de la sección 4.2.
- **Ejemplos de Uso:** Comandos concretos para ejecutar el servidor en diferentes modos y cómo interactuar con él (p. ej., configuración de Claude Desktop [4] o Cursor [5]).
- **Arquitectura:** Un breve resumen de la arquitectura de componentes.

6.4. Buenas Prácticas de Documentación en Go

Para asegurar la calidad y mantenibilidad del código, se seguirán las convenciones de documentación de Go.

- **Comentarios de Documentación:** Cada función, tipo y paquete exportado irá precedido de un comentario de documentación (// o /*... */).
- **Explicar el "Porqué":** Los comentarios se centrarán en explicar la intención y el razonamiento detrás del código (el "porqué"), no simplemente en describir lo que hace (el "qué"), que ya debería ser evidente por el propio código.
- **Uso de go doc:** Se incluirán instrucciones sobre cómo los desarrolladores pueden utilizar la herramienta go doc para ver la documentación directamente desde la terminal (p. ej., go doc./internal/memory), lo que fomenta una exploración y comprensión más profundas del código base.

Conclusiones

El proyecto Remembrances-MCP representa una síntesis ambiciosa y potente de varias tecnologías de vanguardia en el campo de la inteligencia artificial y el desarrollo de software. Al replicar la sofisticada arquitectura de memoria multicapa de Mem0 dentro del ecosistema de alto rendimiento de Golang, y al unificar la persistencia a través de la base de datos multi-modelo SurrealDB, el sistema resultante ofrece una combinación única de inteligencia, flexibilidad y eficiencia.

Las conclusiones clave de este diseño arquitectónico son:

1. **La Memoria como Microservicio:** La inclusión de una API REST y la capacidad de ejecutarse como un servicio independiente transforman a Remembrances-MCP de una simple herramienta de agente a un microservicio de memoria centralizado. Esto permite que una única instancia sirva de "memoria a largo plazo" para múltiples agentes, aplicaciones y sistemas dentro de una organización, promoviendo la consistencia del contexto y reduciendo la duplicación de esfuerzos.
2. **La Unificación Simplifica la Complejidad:** La decisión estratégica de utilizar SurrealDB para los tres niveles de memoria (clave-valor, vectorial y grafo) es fundamental. Elimina la sobrecarga operativa y la complejidad de la sincronización de datos que surgiría al utilizar tres bases de datos especializadas distintas. Esto no solo simplifica el despliegue, sino que también abre la puerta a consultas híbridas potentes que atraviesan los diferentes modelos de datos en una sola operación.
3. **La Portabilidad es una Característica Clave:** La combinación de un binario único compilado en Go y la opción de una base de datos embebida da como resultado una solución de "dependencia cero". Esta portabilidad extrema es invaluable para el desarrollo rápido, los despliegues en entornos de borde (edge) y la facilidad de distribución, diferenciando al proyecto de soluciones más pesadas que dependen de contenedores o múltiples servicios externos.
4. **Sinergia entre Conocimiento Dinámico y Estático:** La integración de una memoria conversacional dinámica con una base de conocimiento estática basada en archivos Markdown crea un sistema de contexto holístico. El agente de IA no solo recuerda sus interacciones pasadas (memoria), sino que también tiene acceso a la "verdad fundamental" del proyecto (conocimiento). El flujo de trabajo "GitOps para el contexto de la IA" que emerge de esta sinergia es un patrón poderoso para mantener a los agentes de IA alineados con el conocimiento humano en evolución.