

Implementación de Memoria Persistente en Agentes de IA

Una Arquitectura Práctica basada en Bases de Datos Vectoriales

Equipo de Desarrollo de Agentes Autónomos

13 de febrero de 2026

Índice

1. Introducción	2
2. Arquitectura del Sistema	2
3. Requisitos Previos	2
3.1. Configuración del Entorno	2
4. Implementación Técnica (Capa de Ejecución)	2
4.1. Guardado de Memoria (<code>save_memory.py</code>)	2
4.2. Consulta Semántica (<code>query_memory.py</code>)	3
4.3. Gestión de Recuerdos (<code>list_memories.py</code>)	3
5. Integración (Capa de Directivas)	4
5.1. Directiva de Guardado (<code>save_memory.yaml</code>)	4
5.2. Directiva de Consulta (<code>query_memory.yaml</code>)	4
6. Caso de Uso Práctico: Resolución de Errores Recurrentes	4
6.1. Escenario	4
6.2. Flujo de Trabajo	5
7. Conclusión	5

1. Introducción

Los Modelos de Lenguaje Grande (LLMs) poseen una limitación fundamental: su 'memoria' es efímera y está restringida por la ventana de contexto. Una vez que una sesión termina o el contexto se llena, el conocimiento adquirido durante la interacción se pierde.

Este documento detalla una solución arquitectónica para dotar a los agentes de IA de una **memoria a largo plazo** (Long-Term Memory). Utilizando una base de datos vectorial local (*ChromaDB*) y scripts deterministas en Python, logramos que el agente almacene aprendizajes, soluciones a errores y preferencias del usuario, recuperándolos semánticamente en futuras sesiones.

2. Arquitectura del Sistema

El sistema sigue una arquitectura de tres capas diseñada para separar la intención probabilística (LLM) de la ejecución determinista (Código).

- **Capa 1: Directivas (YAML):** Definen el 'qué hacer'. Son los procedimientos operativos estándar.
- **Capa 2: Orquestación (LLM):** El cerebro que decide cuándo guardar o consultar la memoria.
- **Capa 3: Ejecución (Python):** Scripts que interactúan físicamente con la base de datos.

3. Requisitos Previos

Para implementar este sistema, se requiere un entorno aislado para manejar las dependencias de Python.

3.1. Configuración del Entorno

```
1 # 1. Crear el entorno
2 conda create --name agent_env python=3.10 -y
3
4 # 2. Activar el entorno
5 conda activate agent_env
6
7 # 3. Instalar dependencias
8 pip install chromadb
```

Listing 1: Creación del entorno con Conda

4. Implementación Técnica (Capa de Ejecución)

4.1. Guardado de Memoria (save_memory.py)

Este script se encarga de vectorizar el texto y almacenarlo en *ChromaDB*. Añade metadatos cruciales como la categoría y la marca de tiempo para facilitar la gestión futura.

```

1 import chromadb
2 import uuid
3 import datetime
4
5 # Inicialización persistente
6 client = chromadb.PersistentClient(path='.tmp/chroma_db')
7 collection = client.get_or_create_collection(name='agent_memory')
8
9 # Generación de metadatos
10 memory_id = str(uuid.uuid4())
11 timestamp = datetime.datetime.now().isoformat()
12 metadata = {
13     'category': args.category, # Ej: 'error_fix', 'user_preference'
14     'timestamp': timestamp,
15     'source': 'user_input',
16 }
17
18 # Upsert (Insertar o Actualizar)
19 collection.add(
20     documents=[args.text],
21     metadatas=[metadata],
22     ids=[memory_id]
23 )

```

Listing 2: Fragmento clave de save_memory.py

4.2. Consulta Semántica (query_memory.py)

A diferencia de una búsqueda por palabras clave (Ctrl+F), este script busca por *similaridad semántica*. Si buscamos 'problema de módulos', el sistema encontrará registros sobre 'ModuleNotFound' aunque las palabras no sean idénticas.

```

1 results = collection.query(
2     query_texts=[args.query],
3     n_results=3 # Top 3 resultados más relevantes
4 )
5
6 # El resultado incluye la 'distancia', donde menor valor
7 # implica mayor similitud semántica.

```

Listing 3: Lógica de búsqueda semántica

4.3. Gestión de Recuerdos (list_memories.py)

Para auditoría y mantenimiento, es necesario poder listar los recuerdos cronológicamente sin realizar búsquedas vectoriales. Dado que ChromaDB no ordena nativamente por metadatos en la consulta básica, lo hacemos en Python.

```

1 data = collection.get() # Obtener todo
2
3 # Estructurar datos
4 memories = []
5 for i in range(len(data['ids'])):
6     memories.append({
7         'id': data['ids'][i],
8         'content': data['documents'][i],
9         'timestamp': data['metadatas'][i].get('timestamp', '')

```

```

10    })
11
12 # Ordenar por fecha descendente (más reciente primero)
13 memories.sort(key=lambda x: x['timestamp'], reverse=True)

```

Listing 4: Listado cronológico

5. Integración (Capa de Directivas)

Las directivas YAML exponen estas capacidades al orquestador.

5.1. Directiva de Guardado (save_memory.yaml)

```

1 goal: 'Guardar un fragmento de información en la memoria a largo plazo.'
2 required_inputs:
3   - name: 'content'
4     description: 'La información textual a recordar.'
5 optional_inputs:
6   - name: 'category'
7     default: 'general'
8 steps:
9   - step: 1
10     script_to_invoke: 'execution/save_memory.py'
11     inputs:
12       - name: '--text'
13         value: '{{content}}'
14       - name: '--category'
15         value: '{{category}}',

```

5.2. Directiva de Consulta (query_memory.yaml)

```

1 goal: 'Recuperar información relevante basada en una consulta semántica.
2
2 required_inputs:
3   - name: 'query'
4     description: 'El contexto sobre el cual se busca información.'
5 steps:
6   - step: 1
7     script_to_invoke: 'execution/query_memory.py'
8     inputs:
9       - name: '--query'
10         value: '{{query}}',

```

6. Caso de Uso Práctico: Resolución de Errores Recurrentes

6.1. Escenario

Un desarrollador encuentra un error complejo durante la compilación y encuentra la solución tras horas de investigación.

6.2. Flujo de Trabajo

1. **Incidente:** El usuario reporta: 'El error 'ModuleNotFound' se soluciona activando el entorno conda'.
2. **Acción del Agente:** El orquestador detecta un aprendizaje útil e invoca `save_memory.yaml`.
 - `content`: 'El error 'ModuleNotFound' se soluciona activando el entorno conda'
 - `category`: 'error_fix'
3. **Persistencia:** El script guarda el vector en `.tmp/chroma_db`.
4. **Recuperación (Meses después):** El usuario pregunta: '¿Por qué me dice que faltan librerías?'.
5. **Consulta:** El orquestador invoca `query_memory.yaml` con la query 'faltan librerías'.
6. **Resultado:** ChromaDB devuelve el recuerdo sobre 'ModuleNotFound' debido a la alta similitud semántica, permitiendo al agente dar la solución correcta inmediatamente.

7. Conclusión

La implementación de una memoria persistente mediante bases de datos vectoriales transforma a los agentes de IA de simples procesadores de texto a sistemas capaces de aprendizaje incremental. Esta arquitectura modular asegura que el conocimiento del equipo se preserve y sea accesible, aumentando la eficiencia operativa y reduciendo la redundancia en la resolución de problemas.