Facsímil 05

Agentes y orquestación

Capítulo 04

Contexto, memoria, compaction y handoff

686f6c61 · 2026

Capítulo 04: Contexto, memoria, compaction y handoff

El momento en que el agente empieza a olvidar

Hay una escena que aparece en casi todos los proyectos con agentes. Empiezas una tarea larga: revisar un repositorio, preparar un informe, analizar varias fuentes o resolver una incidencia de producto. Al principio el agente parece situado. Recuerda el objetivo, sabe qué herramientas ha usado y mantiene el hilo.

Después de muchos pasos, algo cambia. Vuelve a pedir una información ya consultada, confunde una decisión provisional con una decisión cerrada, mezcla preferencias personales con reglas del proyecto o propone repetir una operación que ya falló. No necesariamente porque el modelo sea peor. Muchas veces el sistema le está dando un contexto desordenado, demasiado largo o incompleto.

Este capítulo trata de esa zona gris que en ingeniería se suele nombrar mal: contexto, memoria, compaction y handoff. Si el capítulo 03 explicó cómo un agente actúa mediante tools, aquí veremos cómo sabe **qué debe tener presente** para actuar bien durante más de una llamada.

Qué no deberíamos llamar memoria

No deberíamos llamar memoria a “meter todo el chat otra vez”. Eso es historial, no memoria. Puede funcionar durante un rato, pero crece, cuesta, ralentiza y aumenta la probabilidad de que el modelo atienda a información vieja, contradictoria o poco relevante.

Tampoco deberíamos llamar memoria a “un resumen bonito”. Una narración agradable puede perder los detalles que permiten continuar: IDs, rutas de archivos, decisiones, errores ya probados, permisos concedidos, límites, pruebas ejecutadas y siguiente paso exacto. En agentes, una compaction mala puede sonar elegante y ser inútil.

Y no deberíamos llamar memoria a una carpeta de notas sin curación. Un vault de Obsidian, un workspace de Notion o una base documental pueden ser una fuente magnífica, pero solo si el sistema sabe seleccionar, citar, actualizar, retirar y contextualizar. Un almacén lleno no equivale a una memoria útil.

La definición útil

Para este facsímil, **contexto** es lo que el modelo ve ahora. **Memoria** es lo que el sistema guarda fuera de la llamada y puede recuperar después. **Compaction** es la reescritura controlada del historial para que quepa lo importante. **Handoff** es el paquete de continuidad que permite seguir trabajando sin releerlo todo.

La diferencia importa porque el modelo no recuerda como una persona. En una llamada concreta, procesa una secuencia de tokens. Si algo no está en esa secuencia, no lo puede usar directamente. Si está, pero rodeado de ruido, quizá lo use mal. Si está comprimido de forma ambigua, quizá pierda la razón por la que era importante.

Andrej Karpathy popularizó en 2025 la idea de que estamos pasando de “escribir prompts” a diseñar contextos enteros. En su charla *Software Is Changing (Again)* describe los LLMs como una nueva clase de ordenador programable mediante lenguaje natural, y advierte que los productos maduros se parecen menos a autonomía total y más a sistemas de autonomía parcial bien guiados.[1] Lance Martin resume esa intuición como *context engineering*: llenar la ventana de contexto con la información justa para el siguiente paso, no con toda la información disponible.[2]

La versión práctica para nosotros:

> Un agente no “recuerda” por voluntad propia. Un sistema de agentes construye, recupera, compacta y entrega contexto.

La anatomía formal del contexto

El contexto de una llamada no es «todo el historial»: es una composición ordenada de segmentos, cada uno con un papel. Eso no es una fórmula de la literatura, es una lista de ingeniería, así que la escribimos como tabla, no como ecuación:

SEGMENTO	QUÉ APORTA	EJEMPLO
Instrucciones I	Reglas estables del sistema.	Tono, límites, criterios del proyecto.
Objetivo G_t	La meta actual.	«Revisar el capítulo 04 y dejarlo publicable».
Estado S_t	Qué se hizo, qué falta, qué está bloqueado.	Pasos hechos y pendientes.
Historial H_t	Mensajes o eventos útiles, no todos.	Últimas decisiones relevantes.
Recuperación R_t	Fragmentos de RAG, notas, documentación.	Norma de matrícula citada.
Memoria M_t	Preferencias y hechos consolidados.	«El facsímil usa blanco, negro y grises».
Artefactos A_t	Referencias a objetos.	Rutas, hashes, IDs de tickets, trazas.
Tools T_t	Capacidades disponibles y contratos.	Schemas, permisos, costes.

Lo único que sí es una restricción formal es el presupuesto: el contexto no puede superar la ventana efectiva menos el margen de salida y de tools.

$$\text{tokens}(C_t) \leq B_t$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
$\text{tokens}(C_t)$	Tokens ocupados por el contexto.	46.000 tokens.
B_t	Presupuesto máximo para la llamada.	Ventana efectiva menos margen de salida y tools.

En palabras: puedes ordenar y priorizar los segmentos como quieras, pero su suma en tokens tiene un techo duro; por eso hay que elegir qué entra.

Los tipos de memoria no los inventamos: vienen de la ciencia cognitiva

La memoria de un agente se separa en capas, y esa taxonomía no es nuestra: es la de las arquitecturas cognitivas para agentes de lenguaje (CoALA), que adopta la distinción clásica de la psicología entre memoria de trabajo, episódica, semántica y procedimental.[34]

TIPO (COALA)	QUÉ GUARDA	EJEMPLO EN EL FACSÍMIL
Working (contexto)	Lo activo en la llamada actual.	El objetivo y el estado de este capítulo.
Episódica	Eventos concretos ocurridos.	«El 26 de mayo se decidió no mostrar rangos internos del taller».
Semántica	Hechos estables.	«El facsímil usa blanco, negro y grises».
Procedimental	Reglas de cómo se trabaja.	«Cada capítulo lleva Mermaid y `Dónde solía tropezar yo`».

Recuperar por utilidad: el score de los Generative Agents

Recuperar memorias por mera cercanía semántica trae recuerdos irrelevantes. La solución de referencia es la de los *Generative Agents*, que puntúa cada memoria combinando tres señales: relevancia para la consulta, recencia (con decaimiento temporal) e importancia.[5]

$$\text{score}(m) = \alpha_{\text{rel}}\ \text{rel}(m, q) + \alpha_{\text{rec}}\ \text{rec}(m, t) + \alpha_{\text{imp}}\ \text{imp}(m)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
$\text{rel}(m, q)$	Relevancia (similitud) con la consulta.	Alta si la memoria habla de estructura de capítulos.
$\text{rec}(m, t)$	Recencia con decaimiento exponencial.	Una decisión de ayer pesa más que una de hace un año.
$\text{imp}(m)$	Importancia asignada a la memoria.	Una regla del plan del libro pesa más que un comentario.
$\alpha_{\text{rel}}, \alpha_{\text{rec}}, \alpha_{\text{imp}}$	Pesos de cada señal.	Ajustables por producto y riesgo.

En palabras: una memoria se recupera no porque «se parezca», sino porque es relevante, reciente e importante a la vez. En producción se le añaden filtros de vigencia y coste, pero el núcleo es este score de Park y colaboradores (2023).

Los Generative Agents añaden un paso más, la *reflexión*: cuando la importancia acumulada supera un umbral, el agente resume sus memorias recientes en conclusiones de más alto nivel y las guarda como nuevas memorias (Park y otros, 2023). Es la versión automática de «sacar lecciones»: en lugar de recordar cien eventos, el agente recuerda la conclusión que se deduce de ellos, y esa conclusión vuelve a competir en el score como una memoria más.

Compaction: una función de resumen que debe ser suficiente

Cuando el historial crece, hay que comprimirlo. Formalmente, la compaction es una función de resumen recursivo que mapea el historial acumulado a un handoff compacto.[6]

$$K : C_{1:t} \longrightarrow h_t$$

El resumen no puede ser cualquiera: debe ser **suficiente** para continuar, es decir, preservar lo necesario para decidir igual que con el historial completo. En la práctica, eso son cinco invariantes:

$$\text{suficiente}(h_t) \iff O \wedge D \wedge P \wedge E \wedge N$$

SÍMBOLO	QUÉ DEBE CONSERVAR EL HANDOFF	EJEMPLO
O	Objetivo actual.	Qué se intenta terminar.
D	Decisiones tomadas.	Qué se eligió y por qué.
P	Permisos y límites.	Qué requiere confirmación.
E	Evidencia y artefactos.	URLs, rutas, pruebas, resultados.
N	Siguiente paso.	La acción con la que continuar.

En palabras: un handoff es válido si quien lo recibe (otra sesión, otro agente, otra persona) puede continuar sin releer todo. Un resumen que ahorra tokens pero pierde uno de esos cinco elementos destruye la continuidad.

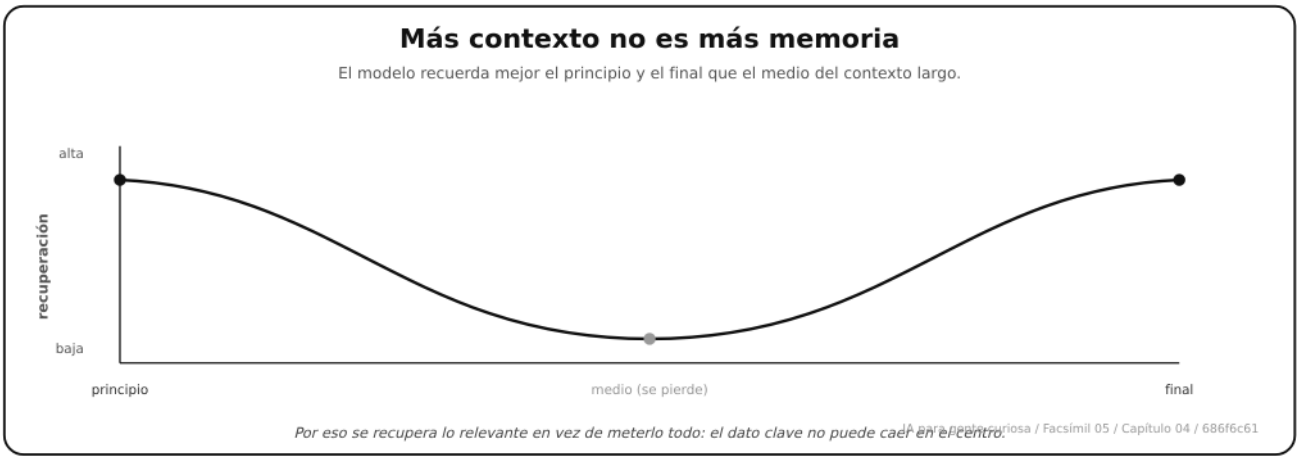
Fecha de corte del estado del arte

Fecha de corte: 10 de junio de 2026.
Fuentes consultadas ese día: charla de Andrej Karpathy sobre Software 3.0; texto de Lance Martin sobre context engineering para agentes; documentación oficial de OpenAI Agents SDK sobre sesiones y compaction; documentación oficial de Anthropic sobre memoria de Claude Code y ventanas de contexto; documentación de Google ADK Memory; LangGraph Persistence; LlamaIndex Memory; documentación de Obsidian sobre grafos y Bases; documentación de Zep y Mem0; Letta/MemGPT; artículos académicos sobre RAG, memoria de agentes y uso de contextos largos.

Lo estable es el mecanismo: el contexto es finito, la recuperación debe ser selectiva, la memoria debe tener ciclo de vida, las compactions deben conservar estado operativo y los handoffs deben ser verificables.

Lo cambiante son los nombres de SDK, límites exactos de ventana, APIs de sesión, formatos de memoria, herramientas comerciales, capacidades de contexto largo, precios y condiciones de proveedor.

Por qué contexto largo no resuelve la memoria



Una ventana de contexto más grande ayuda. Permite meter más documentos, más historial, más resultados de tools y más instrucciones. Pero no convierte automáticamente una conversación larga en una memoria fiable.

El paper *Lost in the Middle* muestra que incluso modelos con contextos largos pueden usar peor la información cuando el dato relevante aparece en posiciones intermedias del contexto. El resultado importante para ingeniería no es “los contextos largos no sirven”, sino “más contexto no significa más control”.[7]

RAG nació precisamente para combinar memoria paramétrica del modelo con memoria no paramétrica recuperada en tiempo de inferencia. Lewis y colaboradores lo plantearon para tareas intensivas en conocimiento: recuperar pasajes explícitos y condicionar la generación con ellos.[8] En agentes, la misma idea se amplía: no solo recuperamos documentos, también reglas, estado, eventos, decisiones y artefactos.

El diseño profesional no pregunta “¿cuánto cabe?”. Pregunta:

PREGUNTA	POR QUÉ IMPORTA
¿Qué información necesita el siguiente paso?	Evita llenar contexto con material interesante pero inútil.
¿Qué información debe salir del contexto activo?	Reduce ruido y coste.
¿Qué debe guardarse como memoria duradera?	Evita repetir aprendizaje entre sesiones.
¿Qué debe mantenerse solo como estado temporal?	Evita convertir cada detalle en recuerdo permanente.
¿Qué debe citarse por referencia, no copiarse entero?	Permite reabrir artefactos sin consumir tokens.
¿Qué memoria puede estar obsoleta?	Impide que una decisión vieja mande sobre una nueva.

La lección de Karpathy: programar el entorno del modelo

La aportación útil de Karpathy para este capítulo no es una receta concreta ni una herramienta comercial. Es el cambio de foco: si el LLM es una especie de ordenador programable con lenguaje natural, entonces el contexto se parece a su memoria de trabajo. El trabajo del ingeniero deja de ser solo “redactar bien la petición” y pasa a ser “construir el entorno de ejecución que el modelo va a ver”.

Eso incluye:

PIEZA DEL ENTORNO	QUÉ DECIDE	EJEMPLO
Instrucciones	Cómo debe comportarse el sistema.	Reglas editoriales del facsímil.
Estado	Qué se sabe ahora mismo.	Capítulo actual, cambios hechos, pruebas pendientes.
Tools	Qué acciones puede proponer.	Buscar referencias, validar SVG, ejecutar build.
Evidencia	En qué se apoya.	Documentación oficial, papers, trazas, capturas.
Memoria	Qué debe recordar entre sesiones.	Preferencias duraderas del proyecto.
Handoff	Cómo se continúa si cambia la sesión.	Resumen operativo con próximos pasos.

La frase “context engineering” puede sonar a etiqueta de moda, pero apunta a un problema real: en sistemas con agentes, el fallo muchas veces no está en el modelo aislado, sino en el contexto que le llega. Contexto viejo, contexto duplicado, contexto contradictorio, contexto sin prioridad o contexto sin evidencia.

Un ejemplo cercano: si el sistema va a seguir este facsímil, no basta con decir “escribe el capítulo 04”. Debe ver que cada capítulo necesita Mermaid, SVG sobrio, fuentes, fecha de corte, fórmulas si aplica, `Dónde solía tropezar yo` antes de `Antes de pasar página`, rutas enlazables y ausencia de lenguaje provisional. Eso no es una frase decorativa: es contexto operativo.

Obsidian: memoria humana, no memoria automática

Obsidian es interesante para este capítulo porque representa muy bien una idea: una memoria útil no es una base de datos enorme, sino una red mantenible de notas, enlaces y propiedades. Obsidian trabaja sobre un *vault*, una carpeta local donde las notas son archivos Markdown. Sus grafos muestran relaciones entre notas; el grafo global enseña todo el vault y el grafo local muestra lo conectado con la nota activa.[9] Sus Bases permiten consultar archivos y propiedades, incluyendo enlaces, etiquetas, tamaño, ruta, fecha de modificación y propiedades YAML.[10]

Pero Obsidian no convierte por sí mismo un agente en alguien con memoria. Sirve como **fuente estructurable**. Para que un agente lo use bien, el vault necesita disciplina:

DECISIÓN EN EL VAULT	POR QUÉ AYUDA AL AGENTE
Una idea por nota cuando sea posible.	Recupera conceptos concretos sin traer capítulos enteros.
Títulos descriptivos.	Mejora búsqueda léxica, enlaces y lectura humana.
Propiedades YAML.	Permite filtrar por tema, fecha, estado, fuente o confianza.
Enlaces bidireccionales.	Explicita relaciones entre conceptos.
Notas de decisión.	Conserva por qué se eligió una opción.
Extractos con fuente.	Evita que el agente cite de memoria.
Fecha de actualización.	Permite detectar conocimiento viejo.
Separación personal/proyecto/cliente.	Evita mezclar ámbitos.

La traducción a un sistema de agentes sería esta:

EN OBSIDIAN	EN UN AGENTE
Nota Markdown	Unidad recuperable de conocimiento.
Backlink	Relación explícita entre conceptos.
Propiedad YAML	Metadato filtrable.
Graph view	Vista de dependencia conceptual.
Canvas	Mapa visual de decisiones o arquitectura.
Base	Consulta estructurada sobre notas.
Vault local	Fuente auditable y portable.

En el mercado hay varias familias: editores locales enlazados como Obsidian, workspaces colaborativos tipo Notion, outliners enlazados como Logseq o Roam Research, espacios personales como Anytype o Capacities, y capas de memoria para agentes como Zep, Mem0, Letta, LangGraph o LlamaIndex. No son equivalentes. Unos están pensados para que una persona piense y escriba; otros para que una aplicación guarde, recupere y evalúe memoria. La pregunta profesional no es “cuál está de moda”, sino “qué unidad de conocimiento necesito, quién la mantiene, cómo se borra, cómo se cita y cómo se evalúa”.

Mercado y estado actual de memoria para agentes

OpenAI Agents SDK ofrece sesiones para mantener historial entre ejecuciones de un agente, evitando que la aplicación tenga que reconstruir manualmente la lista de entradas en cada turno.[11] En la versión JavaScript, la documentación también describe compaction manual para streaming de baja latencia: la compaction puede reescribir la sesión subyacente, y por eso conviene ejecutarla entre turnos si pesa demasiado.[12]

Anthropic documenta memorias de Claude Code mediante ficheros `CLAUDE.md` en varias capas: instrucciones gestionadas por organización, instrucciones de usuario, instrucciones de proyecto e instrucciones locales. También explica que se cargan según jerarquía y que los ficheros de subdirectorios pueden entrar bajo demanda cuando se leen archivos de esas zonas.[13] Esta idea es muy práctica: memoria procedimental versionada, visible y revisable.

Google ADK separa sesiones y memoria. Su `InMemoryMemoryService` sirve para prototipos, permite búsquedas sencillas y documenta herramientas como `load_memory` o `PreloadMemoryTool`; también muestra un callback para extraer memorias desde una sesión mediante `add_session_to_memory`.[14] LangGraph, por su parte, distingue checkpoints de estado por thread y un store para memorias compartibles entre threads. La documentación deja clara la diferencia: el checkpointer permite reanudar una ejecución; el store permite recordar información entre ejecuciones.[15]

LlamaIndex trata la memoria como componente central de sistemas agentic: permite almacenar y recuperar información pasada, combinar memoria corta con bloques de memoria larga y configurar límites de tokens.[16] Zep ofrece una API de memoria donde se añaden mensajes por sesión y se construye un grafo de conocimiento a nivel de usuario a partir de la conversación.[17] Mem0 se presenta como una capa gestionada de memoria para agentes, con memorias de usuario, agente y sesión, además de memoria de grafo, rerankers y controles de plataforma.[18] Letta, heredera del patrón MemGPT, explica la memoria como gestión del contexto: el agente decide qué poner en contexto y qué consultar en almacenamiento externo mediante herramientas.[19]

La tabla honesta sería:

FAMILIA	QUÉ RESUELVE	QUÉ NO RESUELVE SOLA	SEÑAL DE BUEN USO
Sesión	Mantener conversación de una ejecución.	Memoria duradera y curada.	Puedes inspeccionar, limpiar y reanudar.
Checkpoint	Recuperar estado de una trayectoria.	Saber qué recuerdos son útiles en otra tarea.	Cada paso tiene estado serializable.
Store semántico	Guardar hechos o preferencias.	Veracidad, caducidad y permisos.	Cada memoria tiene fuente, ámbito y fecha.
Grafo temporal	Relacionar entidades y cambios en el tiempo.	Coste operativo y evaluación automática.	Puedes explicar por qué se recuperó un hecho.
Vault humano	Pensar, escribir y enlazar conocimiento.	Ingesta automática fiable.	Notas atómicas, enlazadas y con metadatos.
Compaction	Reducir contexto activo.	Corregir decisiones equivocadas.	Conserva objetivo, límites, evidencia y siguiente paso.
Handoff	Continuar entre sesiones o personas.	Ejecutar el trabajo por sí mismo.	Alguien puede seguir sin preguntar “¿dónde estábamos?”.

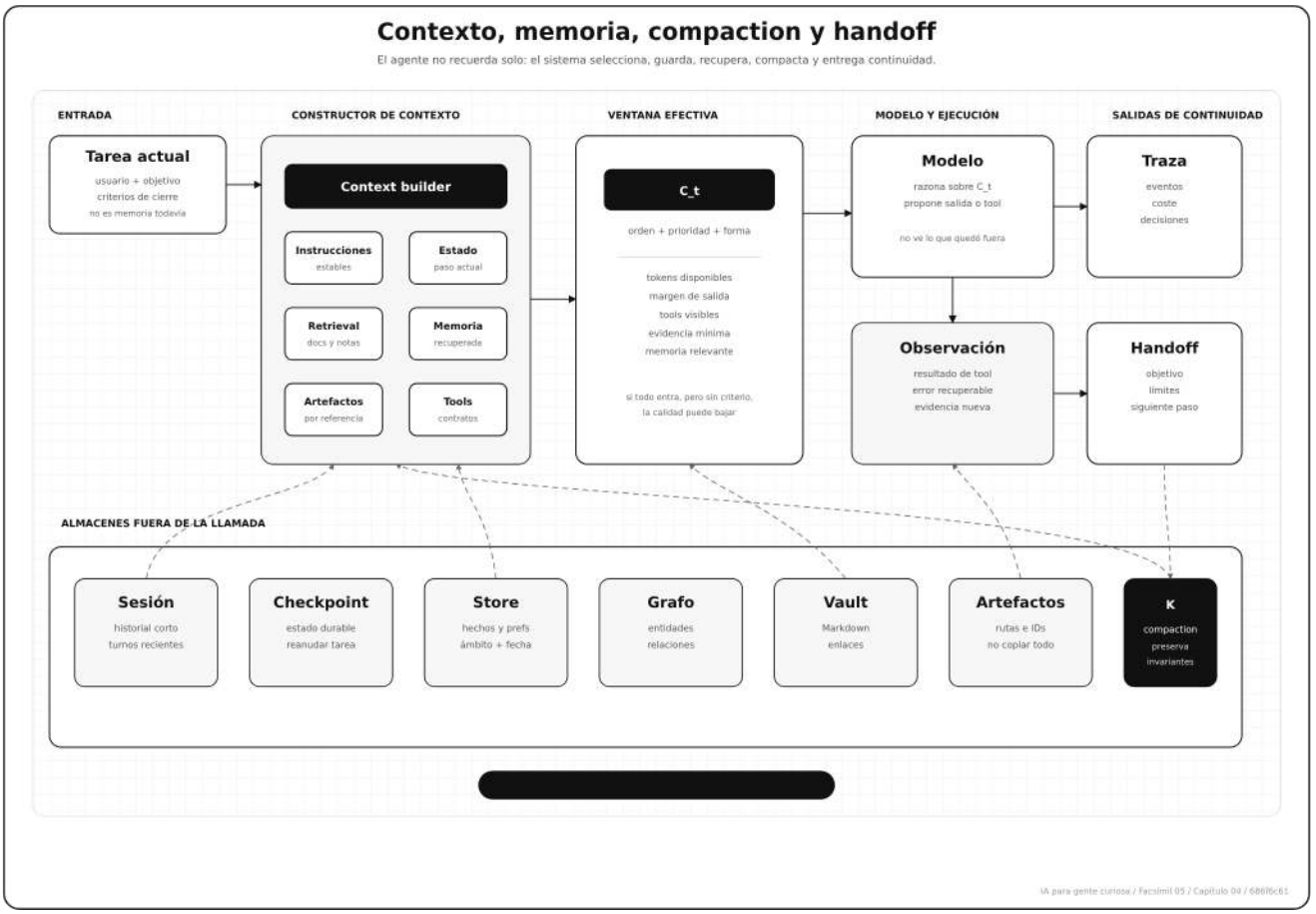
Diseño de memoria por capas

Un sistema serio no tiene “una memoria”. Tiene capas con tiempos de vida distintos.

CAPA	VIDA ÚTIL	DÓNDE VIVE	EJEMPLO	ERROR TÍPICO
Contexto activo	Segundos o minutos	Llamada al modelo	Instrucciones, tarea, tools, fragmentos recuperados.	Meter demasiados tokens “por si acaso”.
Historial de sesión	Minutos u horas	Session store	Turnos recientes y tool calls.	Confundirlo con memoria a largo plazo.
Estado de ejecución	Durante la tarea	Checkpoint o `run_state`	Paso actual, presupuesto, bloqueos, evidencias.	Guardarlo solo en texto conversacional.
Memoria episódica	Días o meses	Log/event store	Qué pasó, cuándo, con qué resultado.	Guardar eventos sin consulta ni expiración.
Memoria semántica	Semanas o años	Store, grafo, DB	Hechos consolidados, preferencias, entidades.	Aceptar cualquier frase como hecho.
Memoria procedimental	Meses o años	Markdown versionado	`AGENTS.md`, `CLAUDE.md`, reglas de proyecto.	Reglas largas, contradictorias y sin dueño.
Artefactos	Según proyecto	Filesystem, Drive, DB, Git	Archivos, capturas, diffs, métricas.	Copiar contenido entero en vez de citar rutas.
Índice documental	Según corpus	Vector DB, search, graph	Políticas, manuales, papers, notas.	Recuperar por similitud sin evaluar utilidad.

La regla es sencilla: lo que cambia rápido no debería vivir como verdad permanente; lo que debe auditarse no debería vivir solo en el contexto; lo que es grande debería entrar por referencia; lo que afecta a comportamiento debe estar versionado.

Arquitectura visual de contexto, memoria y handoff



El diagrama muestra la separación que conviene mantener en código. El constructor de contexto no es el modelo. El store no es el contexto activo. La compaction no es el handoff completo. La traza no es decoración: es la fuente que permite reconstruir qué pasó.

Compaction: resumir no basta

La compaction aparece cuando el historial crece demasiado o cuando queremos continuar en otra sesión. El error habitual es pedir “resume la conversación” y confiar en que eso alcanza. Para agentes, el objetivo no es producir una crónica; es producir un estado operativo.

Un handoff útil debería tener este aspecto:

```yaml
handoff_version: "1.0"
objetivo_actual: "Terminar el capítulo 04 del facsímil 05"
criterios_de_cierre:
  - "Tiene fuentes actuales y papers académicos"
  - "Incluye SVG, Mermaid y práctica ejecutable"
  - "Mantiene el estilo sobrio del libro"
decisiones_tomadas:
  - decision: "Distinguir contexto, memoria, compaction y handoff"
    motivo: "Evita confundir historial con memoria duradera"
limites:
  - "No usar lenguaje provisional en el texto público"
  - "No mostrar rangos internos del taller"
artefactos:
  - ruta: "fasciculo-05-agentes-orquestacion/04-contexto-memoria-compaction-handoff.md"
    tipo: "capitulo"
fuentes_clave:
  - "OpenAI Agents SDK Sessions"
  - "Anthropic Claude Code Memory"
  - "Lost in the Middle"
errores_ya_probados:
  - "Tratar memoria como chat completo"
pendientes:
  - "Validar SVG"
  - "Ejecutar build"
siguiente_paso: "Abrir la ruta local y revisar que capítulo 04 aparece en el menú"
no_rehacer:
  - "No volver a buscar definición básica de RAG; ya está en facsímil 04"
```

Fíjate en tres detalles. Primero, los artefactos entran por referencia. Segundo, las fuentes importantes se nombran para poder reabrirlas. Tercero, aparece `no_rehacer`. Esta pieza es humilde y potentísima: evita gastar tiempo repitiendo caminos ya descartados.

Memoria en productos reales: qué guardar y qué no

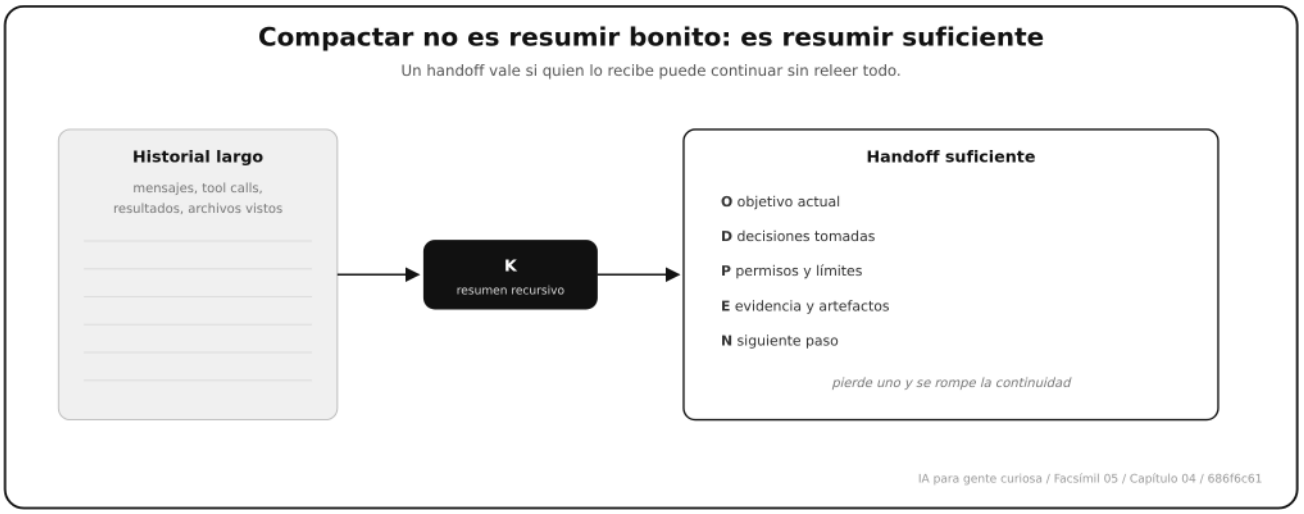
La memoria tiene que tener ciclo de vida. Si un usuario dice “prefiero respuestas cortas”, quizá merece guardarse como preferencia. Si dice “hoy estoy cansado”, quizá solo importa en esa conversación. Si dice “mi dirección es…”, quizá no deberíamos guardarlo salvo que el producto lo necesite y el usuario lo controle. Si una tool devuelve un error transitorio, puede servir para la sesión pero no para siempre.

Una memoria profesional debería tener metadatos mínimos:

CAMPO	PARA QUÉ SIRVE	EJEMPLO
`id`	Trazabilidad.	`mem_2026_05_26_001`
`scope`	Ámbito.	`usuario`, `proyecto`, `equipo`, `cliente`, `sesion`.
`type`	Naturaleza.	`episodica`, `semantica`, `procedimental`, `artefactual`.
`statement`	Contenido atómico.	“El facsímil usa SVGs monocromos firmados”.
`source_ref`	De dónde sale.	`docs/plan-libro.md:626`.
`confidence`	Confianza.	`0.92`.
`created_at`	Fecha de creación.	`2026-06-10`.
`expires_at`	Caducidad si aplica.	`2026-06-26` o `null`.
`owner`	Quién la mantiene.	`autor`, `equipo`, `sistema`.
`delete_policy`	Cómo se retira.	Manual, TTL, reemplazo por memoria más nueva.

El paper de *Generative Agents* ya organizaba agentes con memoria, reflexión y planificación: registraban experiencias, sintetizaban reflexiones y recuperaban recuerdos para planificar comportamiento.[20] MemGPT llevó la analogía más cerca de los sistemas operativos: gestión de contexto virtual, capas de memoria y movimiento de información entre memoria rápida y almacenamiento externo.[21]

El patrón común es este: no basta con recordar; hay que **decidir qué merece volver al contexto**.



Cómo se compacta una trayectoria

Una trayectoria de agente tiene eventos:

1. Usuario pide algo.
2. Sistema fija objetivo y límites.
3. Modelo propone una tool.
4. Runtime valida.
5. Tool devuelve observación.
6. Sistema actualiza estado.
7. Se toman decisiones.
8. Se crean artefactos.
9. Aparecen errores recuperables.
10. Se decide continuar, parar o transferir.

La compaction debería leer esa traza y construir una representación menor. No debería inventar. No debería embellecer. No debería borrar el motivo de una decisión.

TIPO DE EVENTO	QUÉ CONSERVAR	QUÉ DESCARTAR
Objetivo	Resultado esperado y criterios de cierre.	Frases repetidas del usuario.
Tool call	Tool, argumentos importantes, permiso y resultado.	Logs largos sin señal.
Observación	Evidencia, IDs, errores y métricas.	Texto bruto si está guardado por referencia.
Decisión	Qué se eligió, alternativas y motivo.	Debate irrelevante.
Artefacto	Ruta, hash, versión, captura o enlace.	Contenido completo si puede reabrirse.
Error recuperable	Qué falló y qué se probó.	Stacktrace entero si ya existe como archivo.
Pendiente	Siguiente acción concreta.	Deseos vagos.

Una buena compaction debe ser **verificable**. Si el handoff afirma que ya se ejecutó `npm run build`, la traza debería tener el evento. Si dice que una fuente se consultó, debería tener URL. Si dice que una decisión está tomada, debería conservar el motivo.

La parte científica: medir memoria como un experimento

Un sistema de memoria no se valida preguntando “¿parece que recuerda?”. Se valida como cualquier pieza seria de ingeniería: con hipótesis, baseline, conjunto de pruebas, métricas, trazas y comparación.

La hipótesis debe ser falsable:

> “Para tareas largas de revisión técnica, una memoria con store semántico, compaction estructurada y handoff reduce tokens al menos un 35% frente a reenviar el historial completo, manteniendo o mejorando la tasa de continuación correcta.”

Esa frase tiene algo importante: podría salir falsa. Si sale falsa, aprendemos. Si solo decimos “la memoria mejora la experiencia”, no estamos haciendo ingeniería científica; estamos contando una intención.

Sculley y colaboradores advertían que los sistemas de machine learning acumulan deuda técnica oculta cuando no se controlan dependencias, datos, configuración, evaluación y cambios entre versiones.[22] Amershi y colaboradores muestran algo parecido desde la ingeniería de software para ML: los equipos necesitan procesos específicos para datos, evaluación, monitorización y mantenimiento porque el comportamiento no depende solo del código.[23] En memoria de agentes ocurre igual: no basta con que el código compile; hay que medir qué recuerda, qué olvida, qué recupera mal y qué coste añade.

Un diseño de experimento mínimo:

PIEZA	QUÉ FIJAR	EJEMPLO
Unidad de evaluación	Qué cuenta como caso.	Una tarea larga con interrupción y reanudación.
Golden set	Casos representativos con respuesta esperada.	40 tareas: código, documentación, citas, RAG y producto.
Baseline A	Sistema sin memoria.	Solo prompt actual y tools.
Baseline B	Historial completo.	Reenviar todo hasta llenar contexto.
Baseline C	Resumen libre.	Pedir “resume la conversación”.
Variante D	Handoff estructurado.	Schema con objetivo, límites, decisiones y artefactos.
Variante E	Memoria recuperable.	Store con reglas, hechos, eventos y caducidad.
Variables fijas	Lo que no debe cambiar.	Modelo, temperatura, tools, dataset, criterios y presupuesto.
Traza	Qué registrar.	Contexto usado, memorias recuperadas, handoff, coste, salida y decisión final.

Las ablaciones son esenciales. Una ablación consiste en quitar una pieza para ver si realmente aportaba. Si quitamos `no_rehacer` y el agente repite más trabajo, esa pieza vale. Si quitamos memoria semántica y no cambia nada, quizá esa memoria no estaba aportando.

ABLACIÓN	QUÉ QUITAMOS	QUÉ ESPERAMOS OBSERVAR SI ERA ÚTIL
Sin memoria episódica	Eventos pasados.	Más repetición de pasos y errores ya vistos.
Sin memoria semántica	Hechos consolidados.	Más preguntas repetidas y decisiones incoherentes.
Sin memoria procedimental	Reglas de trabajo.	Más incumplimiento de convenciones.
Sin caducidad	Expiración de recuerdos.	Más memorias obsoletas influyendo.
Sin referencias a artefactos	Rutas, IDs y enlaces.	Más copia de texto y menos trazabilidad.
Sin schema de handoff	Campos obligatorios.	Más resúmenes bonitos pero incompletos.
Sin reranking	Ordenación final de memorias.	Más recuerdos parecidos pero poco útiles en top-k.

Métricas para saber si la memoria funciona

La memoria de agentes se evalúa en varios niveles. Un RAG puede medir si recuperó documentos relevantes. Una memoria de agente debe medir además si permitió continuar, si evitó repetir trabajo, si redujo coste y si no introdujo recuerdos obsoletos.

Una primera métrica:

$$\text{Precision@k} = \frac{|\{\text{memorias relevantes en las } k \text{ primeras}\}|}{k}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
k	Número de memorias recuperadas.	5 memorias.
Memorias relevantes	Recuerdos que ayudan a resolver la tarea actual.	4 de las 5.
Precision@k	Proporción útil en el top-k.	$4/5 = 0,80$.

Otra métrica importante es el éxito de continuación:

$$\text{continuacion\_ok} = \frac{N_{\text{tareas reanudadas correctamente}}}{N_{\text{tareas interrumpidas}}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$N_{\text{tareas reanudadas correctamente}}$	Tareas que continúan sin perder objetivo, límites ni evidencia.	31.
$N_{\text{tareas interrumpidas}}$	Tareas cortadas y retomadas.	40.
continuacion_ok	Tasa de continuidad válida.	$31/40 = 0,775$.

Y una tercera métrica detecta memoria vieja:

$$\text{tasa\_obsolescencia} = \frac{N_{\text{memorias obsoletas usadas}}}{N_{\text{memorias usadas}}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$N_{\text{memorias obsoletas usadas}}$	Recuerdos recuperados que ya no deberían influir.	3.
$N_{\text{memorias usadas}}$	Memorias que entraron en contexto.	60.
$\text{tasa\_obsolescencia}$	Fracción de memoria vieja usada.	$3/60 = 0,05$.

La métrica de compresión dice si estamos ahorrando contexto:

$$\text{ratio\_compaction} = \frac{\text{tokens}(h_t)}{\text{tokens}(C_{1:t})}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
h_t	Handoff compacto.	1.800 tokens.
$C_{1:t}$	Historial acumulado antes de compactar.	28.000 tokens.
ratio_compaction	Tamaño relativo tras compactar.	$1800/28000 \approx 0,064$.

Pero cuidado: un ratio bajo no siempre es bueno. Si compactamos a 300 tokens y perdemos el objetivo, el sistema ha comprimido muy bien y ha continuado fatal.

MÉTRICA	QUÉ MIDE	SEÑAL DE ALARMA
`memory_precision@k`	Cuántas memorias recuperadas son útiles.	Top-k lleno de recuerdos vagamente parecidos.
`memory_recall@k`	Cuántas memorias necesarias aparecen.	Faltan reglas críticas del proyecto.
`stale_memory_rate`	Uso de memoria obsoleta.	Decisiones antiguas ganan a reglas nuevas.
`contradiction_rate`	Memorias recuperadas que se pisan.	El modelo recibe instrucciones incompatibles.
`handoff_completeness`	Campos obligatorios presentes.	Falta objetivo, permisos o siguiente paso.
`resume_success_rate`	Tareas reanudadas correctamente.	El agente pregunta otra vez “¿qué hacemos?”.
`token_saving`	Tokens ahorrados frente a baseline.	Se ahorra poco o se pierde calidad.
`human_correction_minutes`	Minutos de corrección humana.	El sistema parece barato pero desplaza coste a revisión.
`latency_p95`	Tiempo de respuesta en casos largos.	La memoria funciona, pero vuelve el producto lento.
`unsupported_claim_rate`	Afirmaciones sin soporte en fuente o traza.	La memoria se convierte en rumor operativo.

RAGAS y LangSmith documentan métricas y flujos de evaluación para RAG, como relevancia de contexto, fidelidad y evaluación de respuestas sobre conjuntos de datos.[24][25] Aquí no las copiamos sin más: las extendemos a memoria de agentes, donde también importan continuidad, caducidad, permisos, artefactos y handoff.

Memoria, RAG, prompt cache y KV cache no son lo mismo

Esta confusión merece una tabla propia. Las cuatro piezas pueden aparecer en la misma aplicación y todas “suenan” a recordar, pero operan en lugares distintos.

CONCEPTO	DÓNDE VIVE	CUÁNTO DURA	QUIÉN LO CONTROLA	PARA QUÉ SIRVE	NO SIRVE PARA
Context o activo	Llamada al modelo	Una inferencia	Aplicación	Dar información al siguiente token.	Recordar entre sesiones si no se vuelve a enviar.
RAG	Índice documental externo	Mientras exista el corpus	Equipo de datos/producto	Recuperar documentos o fragmentos relevantes.	Guardar estado de una trayectoria por sí solo.
Memoria de agente	Store, sesión, grafo o ficheros	Variable	Producto y usuario	Reusar hechos, preferencias, eventos y reglas.	Sustituir verificación o fuentes.
Prompt cache	Infraestructura del proveedor o runtime	Corto o dependiente del proveedor	Runtime/proveedor	Ahorrar coste/latencia con prefijos repetidos.	Elegir qué recuerdos son relevantes.
KV cache	Memoria de inferencia	Durante generación o sesión servida	Runtime	No recalcular atención de tokens ya procesados.	Ser memoria semántica ni fuente citable.
Handoff	Documento o estado estructurado	Hasta continuar o archivar	Sistema/equipo	Reanudar una tarea larga.	Decidir automáticamente qué es verdad permanente.

OpenTelemetry define trazas y spans como unidades para observar trabajo distribuido.[26] En agentes, esa idea ayuda a separar “lo que el modelo vio” de “lo que el sistema hizo”. Si no registramos contexto, memoria recuperada, tools y handoff como eventos, luego solo tendremos una conversación larga y pocas respuestas.

La memoria que envejece mal

Hay un fallo de agentes que no aparece en las demos y que arruina sistemas en producción: la memoria que fue verdad y dejó de serlo. Merece contarse despacio, porque es exactamente el tipo de error que el entusiasmo por «darle memoria al agente» suele pasar por alto.

Imagina un agente de proyecto que, hace tres meses, aprendió un hecho legítimo: «este repositorio ejecuta los tests con `npm test`». En su momento era correcto, alguien lo verificó, y guardarlo ahorró trabajo: a partir de entonces el agente proponía `npm test` sin tener que redescubrirlo. El problema llega el día en que el equipo migra a `pnpm`. El comando bueno pasa a ser `pnpm test`, pero la memoria vieja sigue ahí, con la misma autoridad que el primer

día, entrando en cada decisión como si fuera una verdad permanente. El agente, fiel a su memoria, sigue proponiendo `npm test`; los tests fallan; y lo peor es que el fallo no parece un fallo de memoria, sino un fallo del proyecto. Nadie sospecha de un recuerdo que un día fue cierto.

Este ejemplo pequeño contiene la lección entera. Una memoria no es un hecho eterno; es una afirmación con una fecha de validez que casi nunca conocemos de antemano. Por eso una memoria fiable necesita, como mínimo, tres cosas que un «cajón de texto pegado al prompt» no puede ofrecer. Necesita **procedencia**: de dónde salió el hecho, para poder volver a la fuente cuando hay duda. Necesita **fecha**: cuándo se aprendió, para poder preguntarse si sigue vigente. Y necesita **operaciones de mantenimiento**: poder corregir una memoria que resultó falsa, caducar una que ha envejecido y, sobre todo, saber qué memoria influyó en una decisión concreta para poder rastrear el error hasta su origen.

La diferencia con el contexto del prompt es justo esta. El contexto se construye y se descarta en cada llamada; si era erróneo, el daño dura una respuesta. La memoria persiste, y por eso su error también persiste y se multiplica. El score de recuperación que vimos antes (relevancia, recencia, importancia) ya empuja en la dirección correcta, porque el término de recencia penaliza lo viejo; pero la recencia por sí sola no basta cuando un hecho cambia de golpe, como en la migración a `pnpm`. Ahí hace falta una operación explícita: invalidar la memoria vieja en el momento del cambio, no esperar a que decaiga sola. La regla que conviene llevarse es incómoda pero honesta: si no puedes editar una memoria falsa, caducar una vieja ni explicar cuál influyó en una decisión, no tienes memoria fiable; tienes arrastre de contexto con otro nombre.

Reglas de escritura: cuándo guardar, actualizar o borrar

La memoria debe tener política de escritura. Sin política, cada conversación se convierte en una bolsa de frases.

SITUACIÓN	ACCIÓN RECOMENDADA	EJEMPLO
Preferencia estable del usuario	Guardar con ámbito `usuario`.	“Prefiere explicaciones paso a paso”.
Regla del proyecto	Guardar como memoria procedimental versionada.	“Los SVG del facsímil son monocromos y firmados”.
Decisión temporal de una tarea	Guardar en estado o handoff, no como verdad permanente.	“Hoy revisamos solo capítulo 04”.
Dato sensible o personal	Guardar solo si el producto lo necesita y el usuario lo controla.	Dirección, teléfono, información privada.
Resultado de tool	Guardar referencia y resumen mínimo.	`ticket_id`, estado, fecha, URL.
Error recuperable	Guardar como evento y quizá `no_rehacer`.	“No usar parser X; falló por Y”.
Memoria contradicha por una fuente nueva	Actualizar o retirar.	Regla editorial reemplazada por versión nueva.
Memoria sin fuente	No promover a memoria duradera.	“Creo que el usuario prefiere...”.

Una política mínima:

```
memory_policy:
  write_when:
    - "es estable"
    - "tiene fuente"
    - "ayuda a tareas futuras"
    - "tiene ámbito claro"
  do_not_write_when:
    - "solo sirve para este turno"
    - "no tiene evidencia"
    - "puede caducar pronto"
    - "pertenece a otro ámbito"
  update_when:
    - "hay una fuente más nueva"
    - "otra memoria la contradice"
    - "el usuario corrige explícitamente"
  delete_when:
    - "caducó"
    - "el usuario lo pide"
    - "la fuente fue retirada"
    - "la memoria produce errores repetidos"
```

NIST AI RMF insiste en gobernar sistemas de IA mediante medición, gestión y documentación de riesgos y efectos a lo largo del ciclo de vida.[27] En memoria, esa idea se traduce en algo muy concreto: el usuario o el equipo deben poder saber qué se guarda, por qué se recupera, cuándo caduca y cómo se elimina.

Arquitectura de referencia: proyecto, Obsidian, docs y agente

Veamos una arquitectura útil para un equipo pequeño que trabaja con un proyecto técnico, un vault de Obsidian y un agente con tools.

CAPA	COMPONENTE	RESPONSABILIDAD	EVIDENCIA QUE DEJA
Conocimiento humano	Obsidian vault	Notas, decisiones, conceptos, enlaces.	Markdown, YAML, backlinks.
Corpus técnico	Docs, repo, tickets, papers	Información verificable y artefactos.	URLs, rutas, commits, IDs.
Ingesta	Parser + normalizador	Trocear, limpiar, fechar y etiquetar.	`document_id`, `chunk_id`, hash.
Índice	Búsqueda híbrida	Recuperar por texto, vector y metadatos.	Top-k, score, filtro aplicado.
Memoria	Store por ámbito	Guardar preferencias, reglas y eventos.	`memory_id`, fuente, caducidad.
Context builder	Ensamblador	Elegir qué entra al modelo.	Context manifest.
Agente	Modelo + tools	Proponer pasos y usar herramientas.	Tool calls y observaciones.
Compaction	Reescritor controlado	Reducir historial a handoff.	Handoff validado.
Evaluación	Harness	Medir continuidad, utilidad y coste.	Métricas, baseline, ablaciones.

El `context manifest` es una pieza que merece nombre propio. Es el recibo de lo que vio el modelo:

```
{
  "run_id": "run_2026_05_26_004",
  "task": "revisar capitulo 04",
  "model": "modelo-configurado",
  "context_parts": [
    {"type": "instruction", "source": "docs/plan-libro.md", "tokens": 620},
    {"type": "memory", "source": "memory:project:svg_rules", "tokens": 80},
    {"type": "retrieval", "source": "obsidian://Agentes/Context engineering.md", "tokens":
430},
    {"type": "artifact_ref", "source": "fasciculo-05/.../04-contexto.md", "tokens": 45}
  ],
  "excluded_because": [
    {"source": "nota-antigua.md", "reason": "obsoleta"},
    {"source": "chat-completo", "reason": "supera presupuesto y duplica handoff"}
  ]
}
```

Sin ese manifiesto, cuando el sistema falle será difícil responder a preguntas básicas: qué vio, qué no vio, qué memoria entró, qué documento pesó demasiado y qué se dejó fuera.

Cómo estructuraría un vault de Obsidian para agentes

Si el vault va a alimentar agentes, lo diseñaría con menos romanticismo y más contrato. No hace falta convertir Obsidian en una base de datos rígida, pero sí conviene que las notas tengan forma legible para humanos y máquinas.

Una estructura razonable:

```
VaultIA/
  00-inbox/
  10-conceptos/
  20-decisiones/
  30-proyectos/
  40-fuentes/
  50-retrospectivas/
  90-plantillas/
```

Plantilla de nota de decisión:

```
---
type: decision
project: libro-ia-gente-curiosa
status: active
decided_at: 2026-06-10
owner: 686f6c61
source:
  - docs/plan-libro.md
expires_at:
tags: [agentes, memoria, contexto]
---

# Decisión: cada capítulo lleva handoff si hay trabajo largo

## Contexto
Qué problema resuelve esta decisión.

## Decisión
Qué se hará a partir de ahora.

## Motivo
Por qué elegimos esto frente a otras opciones.

## Cómo lo comprobará un agente
Qué regla, test o búsqueda puede verificarlo.

## Relacionado
- [[Context engineering]]
- [[Compaction]]
- [[Handoff operativo]]
```

Para un agente, esa nota vale más que una página larga sin estructura. Tiene tipo, estado, fecha, owner, fuente, relación y regla de verificación.

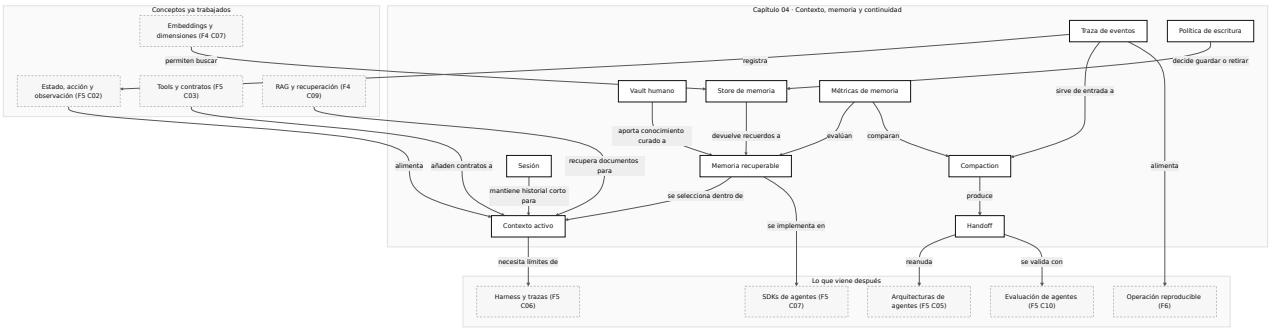
Control humano de la memoria

Un producto con memoria necesita controles visibles. Si el usuario no puede inspeccionar, corregir o borrar memoria, la memoria se convierte en comportamiento opaco.

CONTROL	QUÉ PERMITE	POR QUÉ IMPORTA
Ver memoria	Mostrar qué recuerda el sistema.	Detecta errores y sorpresas.
Editar memoria	Corregir una preferencia o hecho.	Evita que el sistema repita una mala inferencia.
Borrar memoria	Retirar recuerdos.	Da control y reduce carga innecesaria.
Separar ámbitos	Usuario, proyecto, equipo, sesión.	Evita mezclar contextos distintos.
Ver fuente	Saber de dónde salió.	Permite auditar.
Ver última recuperación	Saber cuándo influyó.	Ayuda a depurar decisiones.
Pausar escritura	Impedir guardar durante una tarea.	Útil en trabajo sensible o exploratorio.
Exportar handoff	Continuar con otra persona o herramienta.	Reduce dependencia de una interfaz concreta.

Martin Fowler usa *harness engineering* para hablar del entorno que rodea al agente y permite trabajar con más control: instrucciones, pruebas, contexto, revisión y mecanismos de seguridad operativa.[28] En memoria, el harness no es un extra. Es el lugar donde viven políticas, trazas, evaluaciones y controles humanos.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN ÚTIL
Contexto activo	Tokens que el modelo ve en una llamada concreta.
Ventana de contexto	Límite máximo de tokens que puede procesar el modelo en una llamada.
Memoria	Información guardada fuera de la llamada y recuperada cuando aporta valor.
Sesión	Historial o estado de una conversación concreta.
Checkpoint	Foto serializable del estado de una ejecución para poder reanudar.
Store	Almacén consultable de memorias o hechos.
Memoria episódica	Recuerdo de eventos ocurridos.
Memoria semántica	Hechos consolidados y reutilizables.
Memoria procedimental	Reglas sobre cómo trabajar.
Vault	Carpeta de notas enlazadas, normalmente Markdown.
Compaction	Reescritura estructurada del historial para conservar continuidad con menos tokens.
Handoff	Paquete de continuidad para otra sesión, persona o agente.
Artifact reference	Ruta, ID, hash o enlace que permite reabrir un artefacto sin copiarlo entero.
Context engineering	Diseño del conjunto de información que el modelo debe ver para el siguiente paso.
Expiración	Regla que indica cuándo una memoria deja de ser válida.
Baseline	Variante mínima contra la que comparamos un sistema nuevo.
Ablación	Prueba que quita una pieza para medir si realmente aportaba valor.
Golden set	Conjunto de casos revisados que sirve como referencia de evaluación.
Prompt cache	Caché de prefijos de prompt para ahorrar coste o latencia, sin decidir relevancia.
KV cache	Memoria interna del runtime para no recalcular atención durante inferencia.
Context manifest	Recibo estructurado de qué partes entraron y quedaron fuera del contexto.

TÉRMINO	DEFINICIÓN ÚTIL
Tasa de obsolescencia	Proporción de memorias antiguas usadas cuando ya no deberían influir.

Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Confundir historial con memoria	El chat completo parece cómodo porque no hay que decidir.	Separar sesión, estado, memoria y artefactos.
Compactar como narrador	El resumen suena bien, pero pierde IDs, rutas y decisiones.	Usar un schema de handoff con campos obligatorios.
Guardarlo todo	Parece prudente, pero llena el store de ruido.	Exigir fuente, ámbito, confianza y caducidad.
Recuperar por similitud sin criterio	Lo parecido semánticamente no siempre es útil para la tarea.	Puntuar relevancia, vigencia, autoridad, ruido y coste.
Tratar Obsidian como memoria automática	Tener notas enlazadas no significa que el agente las use bien.	Diseñar notas atómicas, propiedades y reglas de recuperación.
Olvidar el borrado	Una memoria vieja puede mandar sobre una decisión nueva.	Añadir expiración, owner y política de sustitución.

Antes de pasar página

Antes de pasar al capítulo 05, deberías poder responder:

PREGUNTA	SI DUDAS, VUELVE A...
¿Cuál es la diferencia entre contexto, memoria, compaction y handoff?	`La definición útil`.
¿Por qué una ventana larga no sustituye a una memoria bien diseñada?	`Por qué contexto largo no resuelve la memoria`.
¿Qué debe conservar una compaction para no romper continuidad?	`La anatomía formal del contexto` y `Compaction: resumir no basta`.
¿Qué capas tendría una memoria de agente en producción?	`Diseño de memoria por capas`.
¿Por qué Obsidian puede servir como fuente, pero no como memoria automática?	`Obsidian: memoria humana, no memoria automática`.
¿Qué campos mínimos pondrías a una memoria duradera?	`Memoria en productos reales: qué guardar y qué no`.
¿Qué baseline usarías para demostrar que una memoria mejora algo?	`La parte científica: medir memoria como un experimento`.
¿Qué diferencia hay entre memoria, RAG, prompt cache y KV cache?	`Memoria, RAG, prompt cache y KV cache no son lo mismo`.
¿Qué controles humanos necesita un producto con memoria?	`Control humano de la memoria`.
¿Qué tendría que aparecer en un handoff para que otra persona continuase mañana?	Practícalo en el cuaderno del facsímil.

Para saber más

- Anthropic. (2026). *How Claude remembers your project*. https://code.claude.com/docs/en/memory
- Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., & Zimmermann, T. (2019). Software Engineering for Machine Learning: A Case Study. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, 291-300. https://doi.org/10.1109/ICSE-SEIP.2019.00042
- Fowler, M. (2025). *Harness Engineering for Coding Agent Users*. https://martinfowler.com/articles/harness-engineering.html
- Google. (2026). *Agent Development Kit: Memory*. https://adk.dev/sessions/memory/
- Karpathy, A. (2025, 19 de junio). *Software Is Changing (Again)*. Y Combinator AI Startup School. https://rosetta.to/u/ycombinator/andrej-karpathy-software-is-changing-again

- LangChain. (2026). *LangGraph persistence*. https://docs.langchain.com/oss/python/langgraph/persistence
- LangChain. (2026). *Evaluate a RAG application*. https://docs.langchain.com/langsmith/evaluate-rag-tutorial
- Letta. (2026). *Understanding memory management*. https://docs.letta.com/concepts/memory-management
- Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems 33*, 9459-9474.
- Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics, 12*, 157-173. https://doi.org/10.1162/tacl_a_00638
- LlamaIndex. (2026). *Memory*. https://developers.llamaindex.ai/python/framework/module_guides/deploying/agents/memory/
- Martin, L. (2025, 23 de junio). *Context Engineering for Agents*. https://rlancemartin.github.io/2025/06/23/context_engineering/
- Mem0. (2026). *Platform Overview*. https://docs.mem0.ai/platform/overview
- National Institute of Standards and Technology. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. https://doi.org/10.6028/NIST.AI.100-1
- Obsidian. (2026). *Bases syntax*. https://obsidian.md/help/bases/syntax
- Obsidian. (2026). *Graph view*. https://obsidian.md/help/Plugins/Graph%2Bview
- OpenTelemetry. (2026). *Tracing API*. https://opentelemetry.io/docs/specs/otel/trace/api/
- OpenAI. (2026). *Agents SDK: Sessions*. https://openai.github.io/openai-agents-python/sessions/
- OpenAI. (2026). *Agents SDK JS: Sessions*. https://openai.github.io/openai-agents-js/guides/sessions/
- Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., & Gonzalez, J. E. (2024). MemGPT: Towards LLMs as Operating Systems. arXiv. https://doi.org/10.48550/arXiv.2310.08560
- Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *Proceedings of UIST 2023*. https://doi.org/10.1145/3586183.3606763
- RAGAS. (2026). *Available metrics*. https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/
- Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden Technical Debt in Machine Learning Systems. *Advances in Neural Information Processing Systems 28*.

- Sumers, T. R., Yao, S., Narasimhan, K., & Griffiths, T. L. (2023). Cognitive Architectures for Language Agents. *Transactions on Machine Learning Research*. https://arxiv.org/abs/2309.02427
- Tulving, E. (1985). How many memory systems are there? *American Psychologist*, 40(4), 385-398. https://doi.org/10.1037/0003-066X.40.4.385
- Wu, J., Ouyang, L., Ziegler, D. M., Stiennon, N., Lowe, R., Leike, J., & Christiano, P. (2021). Recursively Summarizing Books with Human Feedback. https://arxiv.org/abs/2109.10862
- Zep. (2026). *Memory*. https://help.getzep.com/v2/memory

En resumen

IDEA	QUÉ TE LLEVAS
Contexto no es memoria.	El contexto es lo que el modelo ve ahora; la memoria vive fuera y debe recuperarse con criterio.
Compaction no es resumir bonito.	Una buena compaction conserva objetivo, límites, decisiones, evidencia, artefactos y siguiente paso.
El mercado ofrece piezas, no milagros.	Sesiones, stores, grafos, vaults y SDKs resuelven partes distintas del problema.
Obsidian ayuda si está curado.	Un vault bien enlazado puede ser una fuente excelente; sin metadatos y disciplina solo es texto acumulado.
La ingeniería está en decidir qué entra.	Context engineering consiste en preparar el entorno exacto que el modelo necesita para el siguiente paso.
Una memoria se demuestra con evaluación.	Baselines, ablaciones, métricas, trazas y control humano separan una demo prometedora de un sistema mantenible.

Notas

1. Karpathy, A. (2025, 19 de junio). *Software Is Changing (Again)*. Y Combinator AI Startup School. https://rosetta.to/u/ycombinator/andrej-karpathy-software-is-changing-again. Consultado el 10 de junio de 2026. ↩

2. Martin, L. (2025, 23 de junio). *Context Engineering for Agents*. https://rlancemartin.github.io/2025/06/23/context_engineering/. Consultado el 10 de junio de 2026. ↩

3. Sumers, T. R., Yao, S., Narasimhan, K. y Griffiths, T. L. (2023). Cognitive Architectures for Language Agents. *Transactions on Machine Learning Research*. https://arxiv.org/abs/2309.02427 Organiza la memoria de un agente de lenguaje en working, episodic, semantic y procedural. ↩

4. Tulving, E. (1985). How many memory systems are there? *American Psychologist*, 40(4), 385-398. https://doi.org/10.1037/0003-066X.40.4.385 Distingue memoria episódica, semántica y procedimental. ↩

5. Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P. y Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *Proceedings of UIST 2023*. https://doi.org/10.1145/3586183.3606763 Define el score de recuperación de memoria como suma ponderada de recencia, importancia y relevancia. ↩

6. Wu, J., Ouyang, L., Ziegler, D. M., Stiennon, N., Lowe, R., Leike, J. y Christiano, P. (2021). Recursively Summarizing Books with Human Feedback. https://arxiv.org/abs/2109.10862 Formaliza el resumen recursivo de contenido extenso. ↩

7. Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics, 12*, 157-173. https://doi.org/10.1162/tacl_a_00638. ↩

8. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems 33*, 9459-9474. https://papers.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html. ↩

9. Obsidian. (2026). *Graph view*. https://obsidian.md/help/Plugins/Graph%2Bview. Consultado el 10 de junio de 2026. ↩

10. Obsidian. (2026). *Bases syntax*. https://obsidian.md/help/bases/syntax. Consultado el 10 de junio de 2026. ↩

11. OpenAI. (2026). *Agents SDK: Sessions*. https://openai.github.io/openai-agents-python/sessions/. Consultado el 10 de junio de 2026. ↩

12. OpenAI. (2026). *Agents SDK JS: Sessions*. https://openai.github.io/openai-agents-js/guides/sessions/. Consultado el 10 de junio de 2026. ↩

13. Anthropic. (2026). *How Claude remembers your project*. https://code.claude.com/docs/en/memory. Consultado el 10 de junio de 2026. ↩

14. Google. (2026). *Agent Development Kit: Memory*. https://adk.dev/sessions/memory/. Consultado el 10 de junio de 2026. ↩

.5. LangChain. (2026). *LangGraph persistence*. https://docs.langchain.com/oss/python/langgraph/persistence. Consultado el 10 de junio de 2026. ↩

.6. LlamaIndex. (2026). *Memory*. https://developers.llamaindex.ai/python/framework/module_guides/deploying/agents/memory/. Consultado el 10 de junio de 2026. ↩

.7. Zep. (2026). *Memory*. https://help.getzep.com/v2/memory. Consultado el 10 de junio de 2026. ↩

.8. Mem0. (2026). *Platform Overview*. https://docs.mem0.ai/platform/overview. Consultado el 10 de junio de 2026. ↩

.9. Letta. (2026). *Understanding memory management*. https://docs.letta.com/concepts/memory-management. Consultado el 10 de junio de 2026. ↩

20. Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *Proceedings of UIST 2023*. https://doi.org/10.1145/3586183.3606763. ↩

21. Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., & Gonzalez, J. E. (2024). MemGPT: Towards LLMs as Operating Systems. arXiv. https://doi.org/10.48550/arXiv.2310.08560. ↩

22. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden Technical Debt in Machine Learning Systems. *Advances in Neural Information Processing Systems 28*. https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems. ↩

23. Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., & Zimmermann, T. (2019). Software Engineering for Machine Learning: A Case Study. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, 291-300. https://doi.org/10.1109/ICSE-SEIP.2019.00042. ↩

24. RAGAS. (2026). *Available metrics*. https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/. Consultado el 10 de junio de 2026. ↩

25. LangChain. (2026). *Evaluate a RAG application*. https://docs.langchain.com/langsmith/evaluate-rag-tutorial. Consultado el 10 de junio de 2026. ↩

26. OpenTelemetry. (2026). *Tracing API*. https://opentelemetry.io/docs/specs/otel/trace/api/. Consultado el 10 de junio de 2026. ↩

27. National Institute of Standards and Technology. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. https://doi.org/10.6028/NIST.AI.100-1. ↩

28. Fowler, M. (2025). *Harness Engineering for Coding Agent Users*. https://martinfowler.com/articles/harness-engineering.html. Consultado el 10 de junio de 2026. ↩