Facsímil 05

Agentes y orquestación

Capítulo 01

Agente o prompt: cuándo merece la pena actuar

686f6c61 · 2026

Capítulo 01: Agente o prompt: cuándo merece la pena actuar

La pregunta que abre el facsímil

Venimos de construir una caja de herramientas: APIs, modelos locales, embeddings, RAG, SQL, evaluación, trazas y laboratorios mínimos. Ahora aparece una tentación normal: si ya tenemos herramientas, ¿por qué no dejar que un modelo las coordine?

La respuesta corta es: a veces sí. La respuesta útil es: solo cuando la tarea necesita estado, acciones, observaciones y criterio de parada. Si lo único que necesitas es redactar, clasificar, transformar un texto o devolver JSON en una sola vuelta, un prompt bien diseñado puede ser mejor que un agente.

Este facsímil va de esa frontera. No vamos a tratar los agentes como moda ni como caja negra. Los vamos a tratar como sistemas de decisión alrededor de un modelo. Un agente puede ser muy valioso, pero también introduce más coste, más latencia, más superficie de fallo, más necesidad de trazas y más responsabilidad de diseño.

Qué no es un agente

Un agente no es un prompt largo. Puedes escribir una instrucción enorme, con muchas reglas y ejemplos, y seguir teniendo una sola llamada al modelo. Eso puede estar bien. Pero no convierte el sistema en agente.

Un agente tampoco es “autonomía total”. La autonomía se diseña por grados. Un sistema puede leer documentos sin pedir permiso, preparar una propuesta, pedir confirmación antes de enviar algo y bloquear acciones que no estén dentro de su alcance. Llamarlo agente no significa entregarle todas las llaves.

Y un agente no es una excusa para no diseñar. Si no sabes qué herramientas puede usar, qué datos ve, qué permisos tiene, cómo se mide el éxito y cuándo debe parar, el agente no está “razonando libremente”: está operando sin contrato. Ahí es donde nacen muchos fallos caros.

Qué sí es un agente

En IA clásica, Russell y Norvig definen un agente como algo que percibe su entorno y actúa sobre él para maximizar una medida de rendimiento.[1] Nilsson ya conectaba búsqueda, planificación y agentes como formas de decidir acciones en un espacio de estados.[2] En

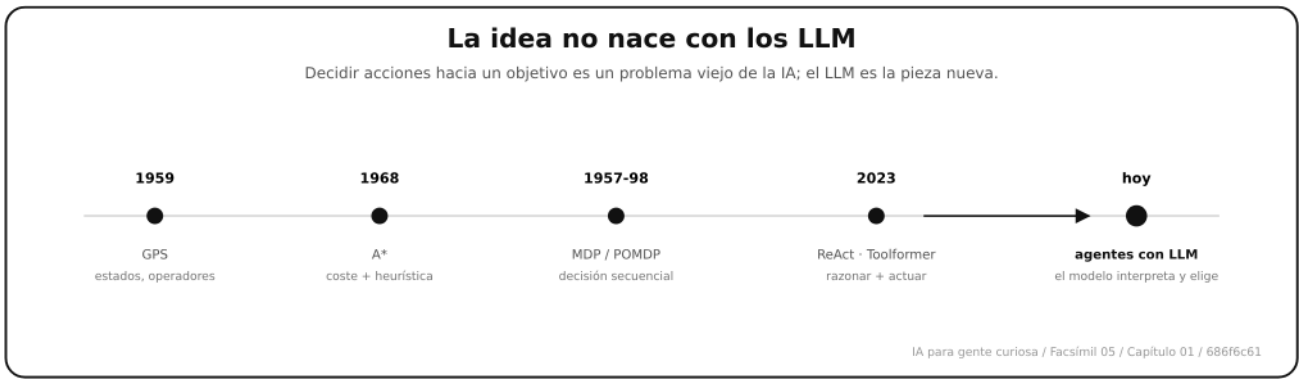
sistemas con LLMs, cambiamos las piezas, pero no la idea de fondo.

Un agente moderno suele tener estas partes:

PIEZA	PREGUNTA QUE RESPONDE	EJEMPLO
Objetivo	¿Qué intenta conseguir?	“Encuentra por qué falla este test”.
Estado	¿Qué sabe hasta ahora?	Archivos leídos, errores vistos, plan activo.
Acciones	¿Qué puede hacer?	Leer fichero, ejecutar test, consultar API, pedir aclaración.
Observaciones	¿Qué volvió después de actuar?	Salida del test, respuesta de API, error validado.
Política de decisión	¿Qué paso toma ahora?	Elegir herramienta, responder, repetir o parar.
Criterio de parada	¿Cuándo termina?	Test pasa, falta permiso, presupuesto agotado.

La diferencia con un prompt aislado es la trayectoria. Un prompt aislado produce una salida. Un agente produce una secuencia: observa, decide, actúa, vuelve a observar y decide si seguir.

Esta idea no empieza con los LLMs. El General Problem Solver de Newell, Shaw y Simon ya formulaba resolución de problemas como búsqueda guiada por objetivos, diferencias entre estado actual y meta, y selección de operadores.[3] Lo nuevo aquí no es querer decidir pasos; lo nuevo es que el modelo de lenguaje participa en interpretar el objetivo, elegir herramientas y resumir observaciones.

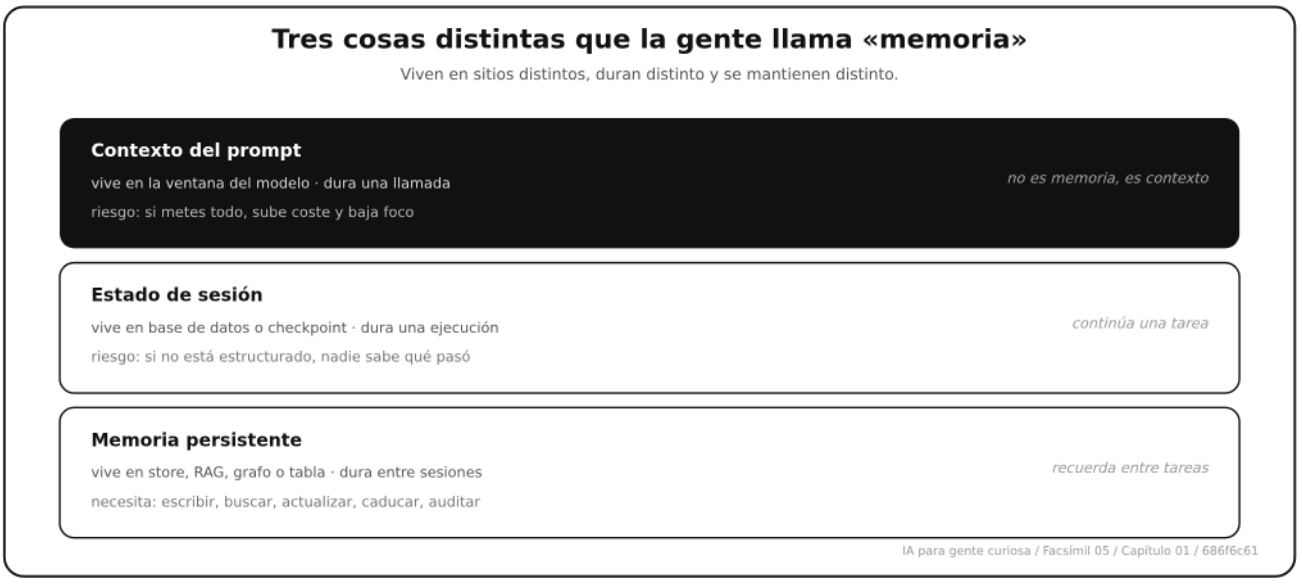


Memoria no significa meter más texto en el prompt

Aquí conviene frenar un segundo, porque la palabra memoria se usa para demasiadas cosas. En un chatbot sencillo, mucha gente llama “memoria” a que el modelo vea mensajes anteriores en el prompt. Eso es **contexto**, no memoria en sentido operativo. El modelo no recuerda por sí mismo: recibe tokens en una llamada concreta.

En agentes, necesitamos separar tres capas:

CAPA	QUÉ ES	DÓNDE VIVE	QUÉ PROBLEMA RESUELVE	QUÉ PROBLEMA CREA
Context o del prompt	Texto que entra en la llamada actual.	Ventana de contexto del modelo.	Da información inmediata para responder o decidir.	Si metes todo, sube coste y baja foco.
Estado de sesión	Historial y variables de una ejecución concreta.	Base de datos, checkpoint, sesión o runtime.	Permite continuar una tarea sin reconstruirla desde cero.	Si no está estructurado, nadie sabe qué pasó.
Memoria persiste nte	Hechos, decisiones, preferencias o aprendizajes reutilizables.	Store externo, RAG, base vectorial, grafo o tabla.	Permite recordar entre sesiones o tareas distintas.	Si no se corrige o caduca, contamina decisiones futuras.



La documentación del Agents SDK distingue sesiones que mantienen historial entre ejecuciones y pueden usar distintos backends, como SQLite, Redis, SQLAlchemy, MongoDB o sesiones gestionadas por OpenAI.[4] Google ADK separa `Session` y `State`, útiles para una conversación concreta, de `MemoryService`, pensado como conocimiento de largo plazo

consultable por el agente.[5] LangGraph, por su parte, habla de persistencia mediante checkpoints de estado, lo que permite memoria conversacional, reanudación, inspección humana y recuperación ante fallos.[6]

La lectura práctica es sencilla: **la memoria nueva de un agente no debería ser un cajón de texto pegado al prompt**. Debe tener operaciones: escribir, buscar, actualizar, borrar, caducar, auditar y explicar por qué se recuperó. Si no puedes editar una memoria falsa, caducar una memoria vieja o saber qué memoria influyó en una decisión, no tienes memoria fiable; tienes arrastre de contexto.

Un ejemplo cercano: si un agente de proyecto aprende "este repositorio usa `npm test`", eso puede guardarse como memoria de proyecto. Pero debe tener fuente, fecha y posibilidad de corrección. Si mañana el repositorio cambia a `pnpm test`, la memoria vieja no puede seguir entrando en todas las decisiones como si fuera verdad permanente.

Estado del arte con fecha de corte

Fecha de corte: 10 de junio de 2026.
Fuentes consultadas ese día: referencias trabajadas en el facsímil sobre agentes clásicos, ReAct, Toolformer, function calling, OpenAI Agents SDK, sesiones y memoria, Google ADK Memory, LangGraph persistence, estrategias agentic en LlamaIndex, Anthropic tool use, Claude Code y principios de mínimo privilegio.

ReAct propuso combinar razonamiento y acción en LLMs para intercalar pasos de pensamiento con llamadas a entornos o herramientas.[7] Toolformer exploró cómo un modelo podía aprender a usar herramientas externas mediante ejemplos generados automáticamente.[8] No son recetas cerradas para producto, pero sí cambiaron la conversación: el modelo ya no solo responde; puede decidir cuándo necesita una herramienta.

En documentación práctica, OpenAI describe function calling como forma de dar herramientas al modelo mediante esquemas y recibir argumentos estructurados.[9] Su Agents SDK se presenta como una capa para coordinar agentes, herramientas, handoffs, guardrails y trazas.[10] LlamaIndex agrupa estrategias agentic como routing, transformaciones de consulta, motores de subpreguntas y agentes sobre datos.[11]

La lección estable no es una librería concreta. La lección estable es esta: cuando un sistema decide acciones, el diseño debe separar modelo, herramientas, permisos, estado, observaciones, límites y evaluación.

La fórmula mínima: un proceso de decisión secuencial

Un agente no es magia: es la versión con LLM de algo que la IA estudia desde hace décadas, el **proceso de decisión secuencial**. Su forma estándar es el proceso de decisión de Markov (MDP), definido formalmente como una tupla.[1][2]

$$\mathcal{M} = \langle S, A, P, R, \gamma \rangle$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
S	Conjunto de estados.	Lo que el agente sabe tras leer logs y ejecutar un test.
A	Conjunto de acciones.	Leer fichero, ejecutar test, consultar API, pedir aclaración.
$P(s' \mid s, a)$	Probabilidad de transición a s' al tomar a en s.	Tras ejecutar el test, el estado pasa a «fallo por timeout».
$R(s, a)$	Recompensa de tomar a en s.	Acercar el diagnóstico, descontando el coste del paso.
γ	Factor de descuento del futuro, $0 \leq \gamma \leq 1$.	Cuánto pesa resolver pronto frente a tarde.

En palabras: un agente es un conjunto de estados, las acciones disponibles en cada uno, cómo se pasa de un estado a otro al actuar, qué recompensa da cada paso y cuánto importa el futuro.

Pero un agente con LLM **no observa el estado completo del mundo**, solo señales parciales: la salida de una tool, un error, un fragmento recuperado. Ese caso es el MDP parcialmente observable (POMDP), que añade un espacio de observaciones Ω y una función de observación O, y obliga a decidir sobre una *creencia* b en vez de sobre el estado real.[1][3]

$$\mathcal{P} = \langle S, A, T, R, \Omega, O, \gamma \rangle$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
Ω	Conjunto de observaciones posibles.	«test falla por timeout», «saldo: 180 EUR».
$O(o \mid s', a)$	Probabilidad de observar o tras llegar a s'.	Una tool fiable hace o muy informativa de s'.
$b(s)$	Creencia: distribución de probabilidad sobre estados.	El agente no sabe el estado exacto; estima cuál es más probable.

En palabras: como el agente solo ve pistas, mantiene una estimación de en qué estado está y la actualiza con cada observación. Todo lo que en ingeniería llamamos «contexto, historial, memoria y presupuesto» es la forma práctica de representar esa creencia b.

La política π elige la acción, y la **política óptima** es la que maximiza la recompensa esperada descontada. Su forma canónica es la ecuación de optimalidad de Bellman sobre el valor de acción Q^*:[14]

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, \max_{a'} Q^*(s',a') \qquad \pi^*(s) = \arg\max_a Q^*(s,a)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
$Q^*(s,a)$	Valor óptimo de tomar a en s y seguir óptimamente.	Cuánto vale, en total, leer el error antes de tocar código.
$\max_{a'} Q^*(s',a')$	Mejor valor alcanzable desde el estado siguiente.	El mejor desenlace posible tras observar el resultado.
$\pi^*(s)$	Política óptima: la acción de mayor valor.	Elegir la acción que más acerca al objetivo neto de coste.

En palabras: la mejor acción es la que da más recompensa ahora más el mejor futuro que abre, descontado. La recompensa R ya incorpora el balance entre utilidad y coste, que es la utilidad esperada en el sentido de von Neumann y Morgenstern (1944).[15] Por eso un agente bien diseñado prefiere leer el error exacto antes que modificar código a ciegas.

En la práctica no resolvemos esta ecuación: el LLM **aproxima** la política, eligiendo la siguiente acción a partir de la creencia (el contexto) sin calcular Q^* de forma explícita. Pero la ecuación fija el criterio: cada acción debe justificar su recompensa esperada frente a su coste, y la observación que devuelve actualiza la creencia para la siguiente decisión. Tras actuar, el estado y la observación evolucionan según P y O del POMDP:

$$s_{t+1} \sim P(\cdot \mid s_t, a_t), \qquad o_{t+1} \sim O(\cdot \mid s_{t+1}, a_t)$$

En palabras: la acción lleva a un nuevo estado y este emite una observación; con ella el agente revisa su creencia y vuelve a decidir. Si una herramienta no devuelve observación estructurada, O es ruidosa y la creencia se contamina. Si no hay presupuesto ni criterio de parada, el bucle puede seguir aunque el valor Q^* ya no mejore.

Del formalismo a las piezas que controlas

El MDP es abstracto a propósito. En un agente real, cada símbolo se convierte en una pieza de ingeniería concreta. Esta es la traducción, y es la que de verdad diseñas:

SÍMBOLO DEL (PO)MDP	PIEZA DE INGENIERÍA	QUÉ CONTROLAS TÚ
Creencia b (estado)	Objetivo, historial, contexto, memoria, presupuesto.	Qué entra en el contexto y qué se persiste fuera del prompt.
Acciones A	Tools, preguntar, leer, calcular, responder, handoff.	El catálogo de tools y el permiso de cada una.
Observación O	Salida tipada de la tool, error, evidencia, test.	Que la observación sea estructurada, no una frase suelta.
Recompensa R	Valor de la tarea menos coste y riesgo.	La función de éxito y los límites de coste.
Descuento γ y parada	Presupuesto y criterio de parada.	Cuándo el agente termina o pide aprobación.

Esto no obliga a implementar un agente con ecuaciones explícitas. Sirve para no perder el mapa: el agente es un POMDP que el LLM aproxima, y todo lo que diseñas (tools, memoria, permisos, presupuesto, trazas) es la forma de hacer ese POMDP operable y observable.

El bucle, contado despacio: un agente que diagnostica un login

La teoría del apartado anterior se entiende mejor si la vemos correr una vez, paso a paso, sin prisa. Imagina un agente de código al que le llega una sola frase: «la página de login falla en producción». Esa frase es toda su evidencia inicial. No sabe la causa, no sabe qué archivos importan, no sabe si el problema es del navegador, del servidor o de la sesión. En términos del POMDP, su creencia de partida es casi plana: muchos estados posibles, ninguno claramente más probable que otro. Un buen agente no disimula esa ignorancia respondiendo con seguridad; la reconoce y diseña su primera acción para reducirla.

Primer paso. La política mira la creencia y se pregunta qué acción aporta más información por menos coste. Editar código a ciegas sería caro y arriesgado (alto coste, alto riesgo, poca utilidad esperada), así que la descarta. En su lugar elige una acción barata y muy informativa: leer el log de errores. Es la lógica del valor de la información que vimos antes: no se actúa para «avanzar», se actúa para *saber*. La observación que vuelve es concreta: `timeout en /auth/callback`. Esa cadena, tipada y sin ambigüedad, es justo lo que el POMDP llama una observación informativa, porque es mucho más compatible con unos estados que con otros.

Actualización de creencia. Aquí ocurre el gesto que distingue a un agente de un prompt. La observación no «da la respuesta»; cambia la distribución de probabilidad sobre las causas. Un timeout en el callback de autenticación es muy propio de un problema de sesión y poco propio de un fallo de maquetación del router. La creencia se desplaza: lo que antes era una nube de hipótesis ahora tiene un favorito claro. El agente no ha resuelto nada todavía, pero ha pasado de no saber por dónde empezar a saber dónde mirar. Ese desplazamiento, repetido, es el trabajo entero.

Segundo paso. Con la sesión como hipótesis principal, la política vuelve a elegir. Podría editar el manejador de sesión de inmediato, pero eso seguiría siendo una acción de alto riesgo sobre una creencia que aún no es certeza. Elige antes una acción de verificación de coste medio: ejecutar el test de integración del flujo de sesión. La observación vuelve a estrechar la creencia, esta vez casi hasta la certeza: el test reproduce el `timeout` y señala una cookie de sesión que caduca antes de tiempo. Solo ahora, con la creencia concentrada en un único estado y con evidencia que lo respalda, la acción de modificar código deja de ser temeraria y pasa a ser razonable.

Criterio de parada. El agente no termina porque «ha dicho algo». Termina cuando se cumple una condición verificable: el test vuelve a pasar tras el cambio, o se agota el presupuesto, o aparece una acción que exige un permiso que no tiene. Si hubiera tenido que tocar la configuración de producción, un agente bien diseñado se habría detenido antes para pedir aprobación, por mucho que su creencia fuera firme. La firmeza de la creencia no es lo mismo que la autorización para actuar: lo primero lo decide la evidencia, lo segundo lo deciden los permisos.

Fíjate en lo que ha pasado en estas cuatro etapas. En ningún momento el agente ha «razonado libremente». En cada paso ha hecho lo mismo: mirar su creencia, elegir la acción de mayor valor neto, recibir una observación tipada y actualizar la creencia. Esa repetición disciplinada (y no la potencia del modelo) es lo que convierte una frase vaga en un diagnóstico defendible. Y es también la razón por la que un agente necesita estado, observaciones estructuradas, presupuesto y trazas: cada una de esas piezas sostiene una parte del bucle. Quítale las observaciones tipadas y la creencia se contamina; quítale el presupuesto y el bucle no sabe parar; quítale la traza y, aunque acierte, nadie podrá reconstruir por qué.

El primer criterio: ¿una vuelta o varias?

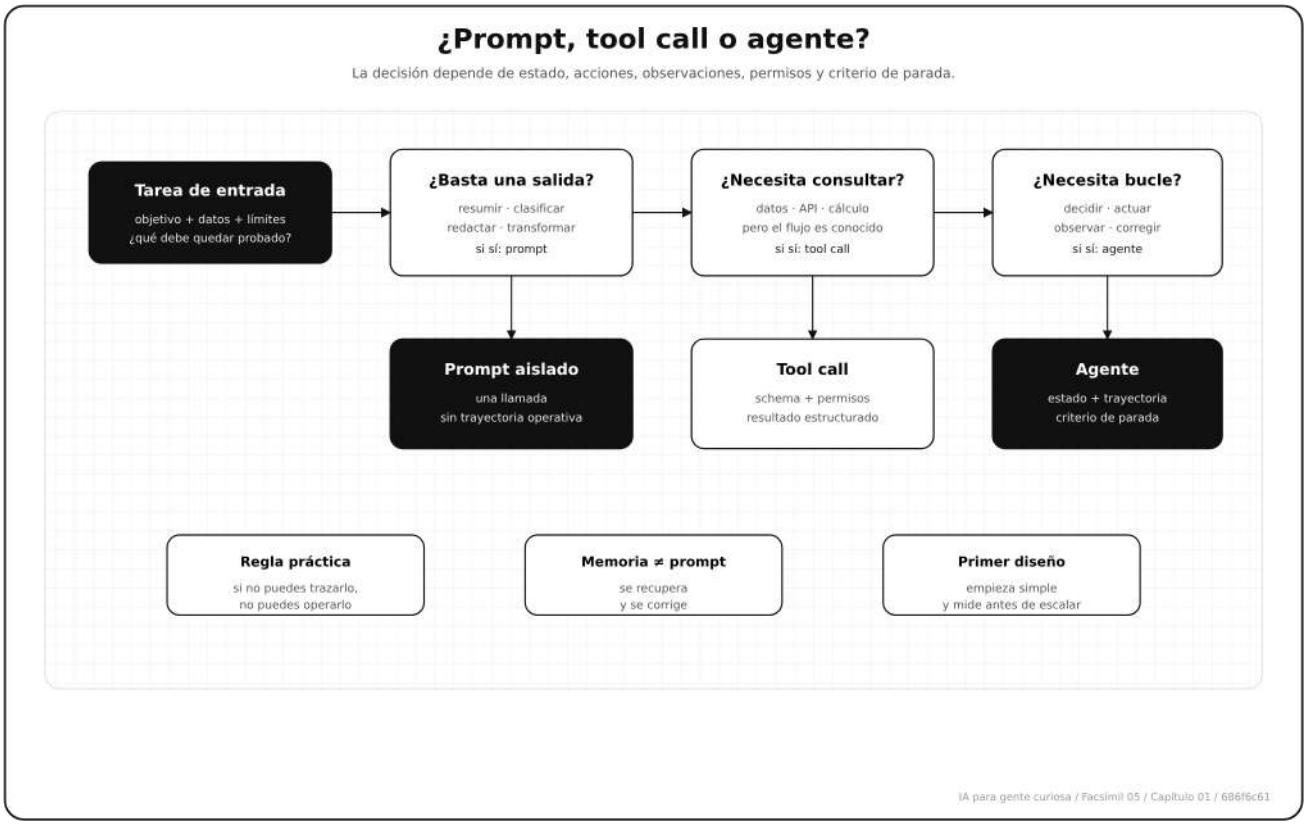
El diagnóstico más simple es preguntar si la tarea cabe en una sola vuelta.

SI LA TAREA...	SUELE BASTAR CON PROMPT	EMPIEZA A PEDIR AGENTE
Necesita redactar, resumir o clasificar una entrada cerrada	Sí	No necesariamente
Necesita consultar estado real	No	Sí
Necesita elegir entre varias herramientas	No	Sí
Necesita probar, observar y corregir	No	Sí
Tiene acciones con permisos	No	Sí, pero con límites
Tiene éxito verificable por tests o métricas	A veces	Sí
Requiere memoria operativa entre pasos	No	Sí

Un caso cercano: “resume este correo” es prompt. “Busca en el CRM los últimos pedidos del cliente, comprueba si hay una incidencia abierta, redacta una respuesta y espera aprobación” ya no es solo prompt. Ahí hay herramientas, estado, permisos y criterio de parada.

La pregunta profesional no es “¿podemos hacerlo con un agente?”. Casi siempre podremos. La pregunta es: ¿el agente reduce trabajo real más de lo que añade coste, latencia y complejidad?

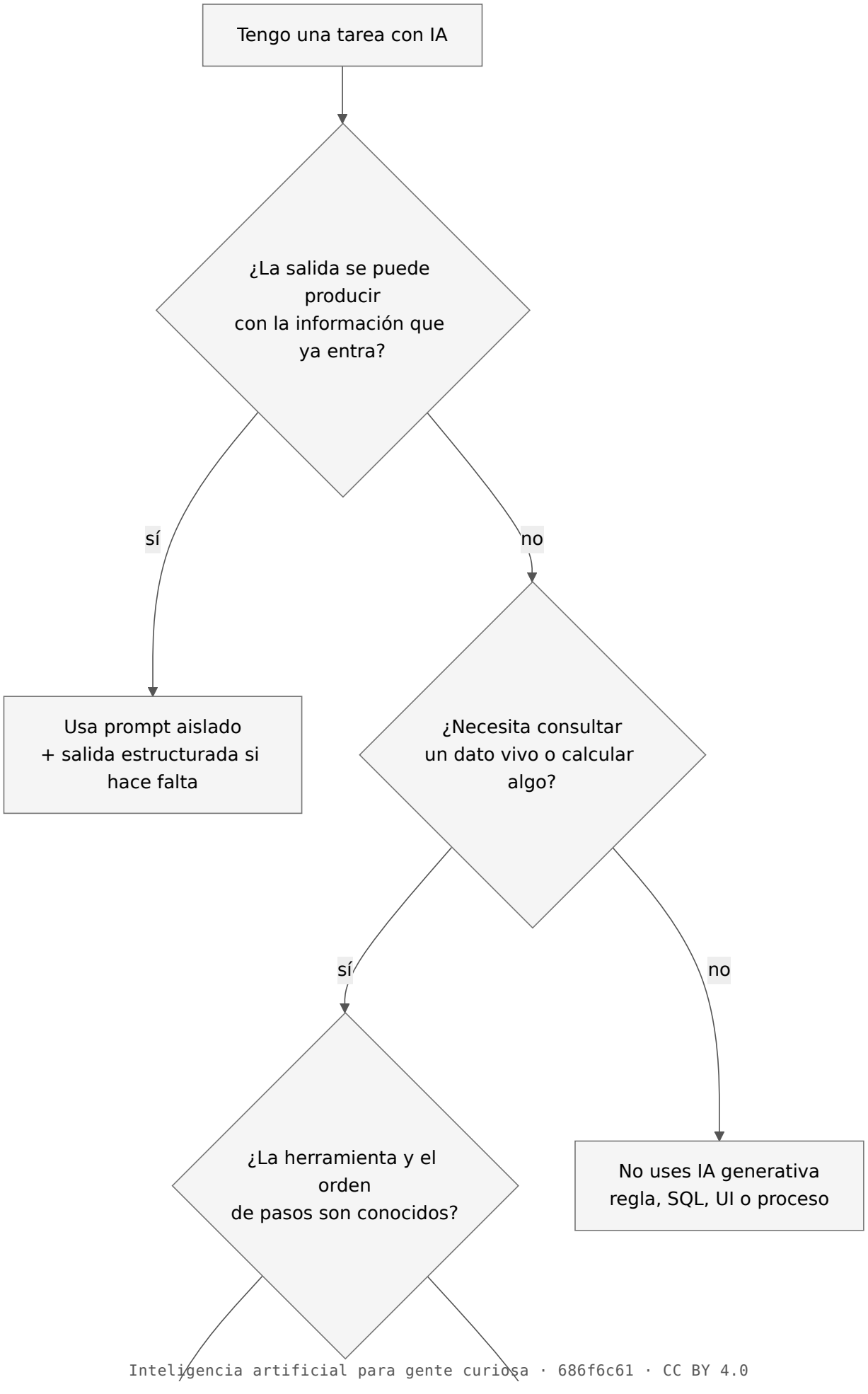
Mapa visual de decisión

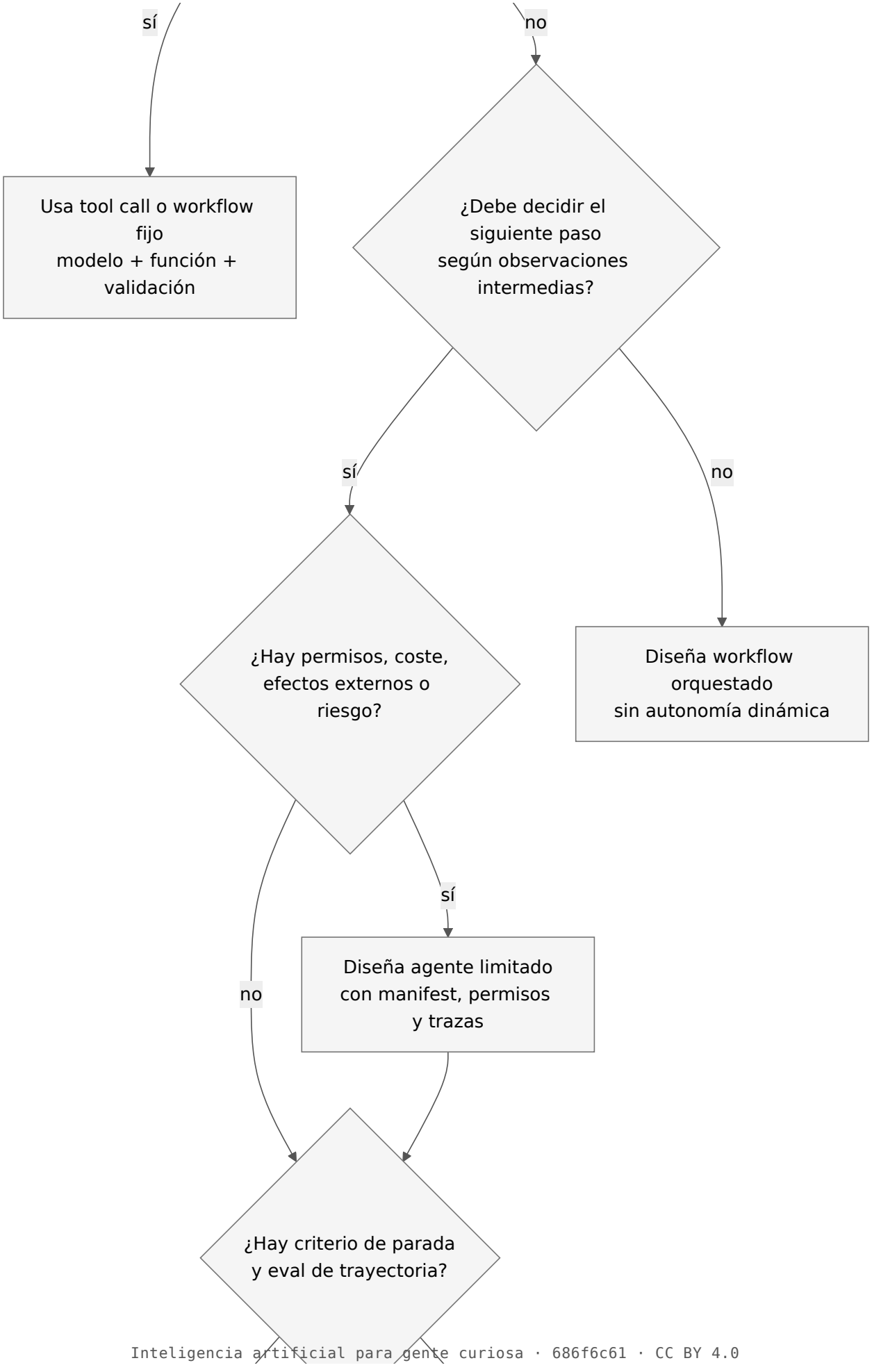


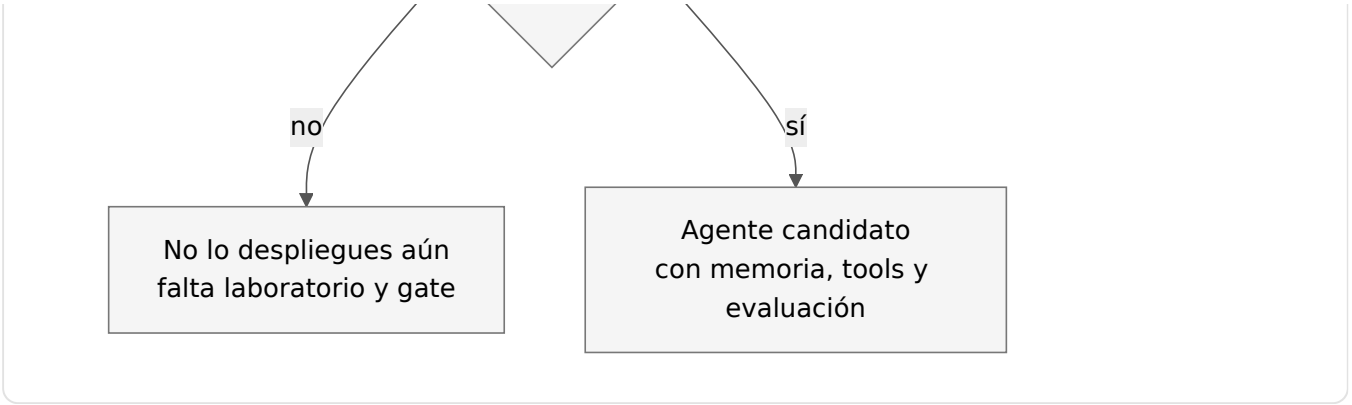
La figura separa tres niveles. El prompt resuelve una transformación cerrada. La llamada a herramienta resuelve una consulta o acción conocida. El agente aparece cuando el sistema debe elegir varios pasos según lo que vaya observando.

Árbol de decisión para no complicarte demasiado pronto

El árbol siguiente sirve para una primera conversación de diseño. No sustituye la evaluación, pero evita una confusión muy frecuente: llamar agente a cualquier cosa que use un modelo.


Tengo una tarea con IA
¿La salida se puede producir con la información que ya entra?
sí
no
Usa prompt aislado + salida estructurada si hace falta
¿Necesita consultar un dato vivo o calcular algo?
sí
no
¿La herramienta y el orden de pasos son conocidos?
No uses IA generativa regla, SQL, UI o proceso


sí
no
Usa tool call o workflow fijo
modelo + función + validación
¿Debe decidir el siguiente paso según observaciones intermedias?
sí
no
¿Hay permisos, coste, efectos externos o riesgo?
Diseña workflow orquestado
sin autonomía dinámica
sí
no
Diseña agente limitado
con manifest, permisos
y trazas
¿Hay criterio de parada
y eval de trayectoria?



La lectura del árbol es sencilla:

SI RESPONDES...	LO RAZONABLE ES EMPEZAR CON...	POR QUÉ
“Sí, todo está en la entrada”	Prompt aislado	No necesitas estado ni herramientas.
“Necesito un dato vivo, pero sé exactamente cuál”	Tool call o workflow fijo	El modelo puede preparar argumentos; tu sistema ejecuta y valida.
“Necesito varios pasos, pero el orden es conocido”	Workflow orquestado	No hace falta que el modelo decida cada paso.
“El siguiente paso depende de lo observado”	Agente limitado	Ya hay bucle de decisión, acción y observación.
“Hay permisos o efectos externos”	Agente con manifest y aprobación	La autonomía debe estar graduada por acción.
“No tengo eval ni criterio de parada”	No desplegar todavía	Sin trayectoria medible, no sabes si funciona de forma repetible.

Este árbol tiene una moraleja: **el agente suele aparecer tarde**, no al principio. Primero pregunta si basta con una llamada. Luego si basta con una herramienta. Luego si basta con un workflow. Solo cuando la tarea necesita decidir dinámicamente según observaciones empieza a merecer la palabra agente.

En el día a día

Imagina un equipo de soporte universitario. Una persona escribe: “No puedo matricularme y creo que tengo un pago pendiente”. Hay varias formas de resolverlo.

Un prompt podría redactar una respuesta amable. Una tool podría consultar si el expediente tiene pagos pendientes. Un agente podría hacer algo más amplio: leer la política de matrícula vigente, consultar pagos, comprobar si falta documentación, preparar una respuesta con

evidencia y pedir aprobación antes de enviarla.

La tercera opción suena más potente, pero solo merece la pena si la tarea se repite, si la decisión depende de datos vivos y si el resultado se puede verificar. Si el equipo recibe dos casos al mes, quizá baste con una plantilla y una consulta manual. Si recibe miles, con reglas claras y trazas, el agente empieza a tener sentido.

Por qué debería importarte

Un agente cambia la unidad de evaluación. Ya no basta con leer la respuesta final. Hay que mirar la trayectoria: qué datos usó, qué herramienta eligió, qué observó, cuánto costó, qué permisos aplicó y por qué decidió parar.

Esta es la razón por la que el facsímil 04 terminó con laboratorios y trazas. Antes de coordinar herramientas, necesitamos saber medir piezas pequeñas. Un agente no elimina esa disciplina: la multiplica.

Saltzer y Schroeder formularon principios clásicos como mínimo privilegio y mediación completa: cada acceso relevante debe comprobarse y cada componente debe operar con el permiso mínimo necesario.[16] En agentes, esa idea deja de ser abstracta: cada herramienta debe tener un alcance explícito, y cada acción con efecto real debe pasar por una decisión externa al modelo.

Cómo lo estructuran OpenAI y Claude

La forma exacta cambia entre proveedores, pero la arquitectura que hay debajo se parece mucho. Un sistema de agentes no es “un modelo con ganas de hacer cosas”. Es un runtime que prepara contexto, ofrece tools, ejecuta llamadas, devuelve observaciones, conserva estado, aplica permisos y deja trazas.

En OpenAI Agents SDK, un `Agent` se describe como un LLM configurado con instrucciones, herramientas y comportamiento opcional como handoffs, guardrails y salidas estructuradas; `Agent` más `Runner` permite que el SDK gestione turnos, tools, guardrails, handoffs y sesiones.[17] Las sesiones mantienen historial entre ejecuciones y pueden apoyarse en distintos backends; eso no es “memoria humana”, sino persistencia controlada de conversación y estado.[18] La trazabilidad también es explícita: el SDK registra generaciones del modelo, llamadas a herramientas, handoffs, guardrails y eventos propios como spans dentro de una traza.[19]

En Anthropic, la guía *Building Effective Agents* separa workflows y agents: un workflow sigue rutas predefinidas por código; un agent deja que el LLM dirija dinámicamente el proceso y el uso de herramientas.[20] En la API de Claude, las tools se declaran en `tools` con nombre, descripción y esquema de entrada; el modelo puede devolver bloques `tool_use` y el cliente

continúa la conversación con bloques `tool_result`.[21] En Claude Code, la memoria se organiza con archivos `CLAUDE.md` de distintos ámbitos.[22] Los subagentes se definen como Markdown con frontmatter, propósito, herramientas permitidas y ventana de contexto separada.[23]

La traducción a piezas queda así:

PIEZA DEL FACSÍMIL	OPENAI AGENTS SDK	CLAUDE / ANTHROPIC	QUÉ DEBES MIRAR COMO INGENIERO
Instrucciones	`instructions`, prompts y configuración del agente.	System prompt, `CLAUDE.md`, configuración de subagente.	Qué manda, con qué prioridad y dónde vive versionado.
Tools	Function tools, hosted tools, agents as tools.	`tools`, `tool_use`, `tool_result`, tools de Claude Code.	Schema, descripción, permisos, timeout, errores y límites de salida.
Estado de sesión	`Session`, conversaciones, backends de persistencia.	Conversación, memoria de Claude Code, contexto del subagente.	Qué se guarda entre pasos y cómo se corrige.
Delegación	Handoffs o agentes expuestos como tools.	Subagentes con contexto y tools propias.	Cuándo delega, qué contexto pasa y qué vuelve.
Controles	Guardrails, output types, approval gates, policies propias.	Permisos de tools, límites del agente, confirmaciones de Claude Code.	Qué acción se permite, se bloquea o pide revisión.
Trazas	Traces y spans del SDK.	Historial de tool use, logs del runtime, trazas propias.	Cómo reconstruyes la trayectoria sin leer una novela.

Fíjate en la idea común: el proveedor puede darte SDK, API o entorno de código, pero el diseño sigue siendo tuyo. Tú decides qué tools existen, qué memoria entra, qué permisos se aplican, qué salida se acepta y qué traza basta para depurar.

El panorama de frameworks y el coste real

No tienes que construir el bucle a mano. Hoy hay varios marcos que ya traen estado, tools, trazas y, a veces, orquestación de varios agentes. Cambian rápido, así que aquí importa el papel de cada uno, no la versión.

Fecha de corte: 10 de junio de 2026. Fuentes: documentación oficial de cada proyecto consultada ese día.

FRAMEWORK	LENGUAJE	PARA QUÉ BRILLA
OpenAI Agents SDK	Python, TypeScript	Bucle de agente con handoffs, guardrails y tracing integrados.
Claude Agent SDK (Anthropic)	Python, TypeScript	Construir sobre Claude con tool use, subagentes y memoria de archivos.
Google ADK	Python, Java	Sesiones, `State` y `MemoryService` con soporte multi-agente.
LangGraph (LangChain)	Python, JS	Grafos de estado con checkpoints, reanudación e inspección humana.
LlamaIndex	Python, TS	Estrategias agentic sobre datos: routing, subpreguntas, agentes sobre índices.
Microsoft AutoGen	Python	Conversaciones entre varios agentes con roles y herramientas.
CrewAI	Python	Equipos de agentes con roles, tareas y delegación.
Pydantic AI	Python	Agentes tipados con validación de entrada y salida por esquema.
smolagents (Hugging Face)	Python	Agentes que actúan escribiendo y ejecutando código.

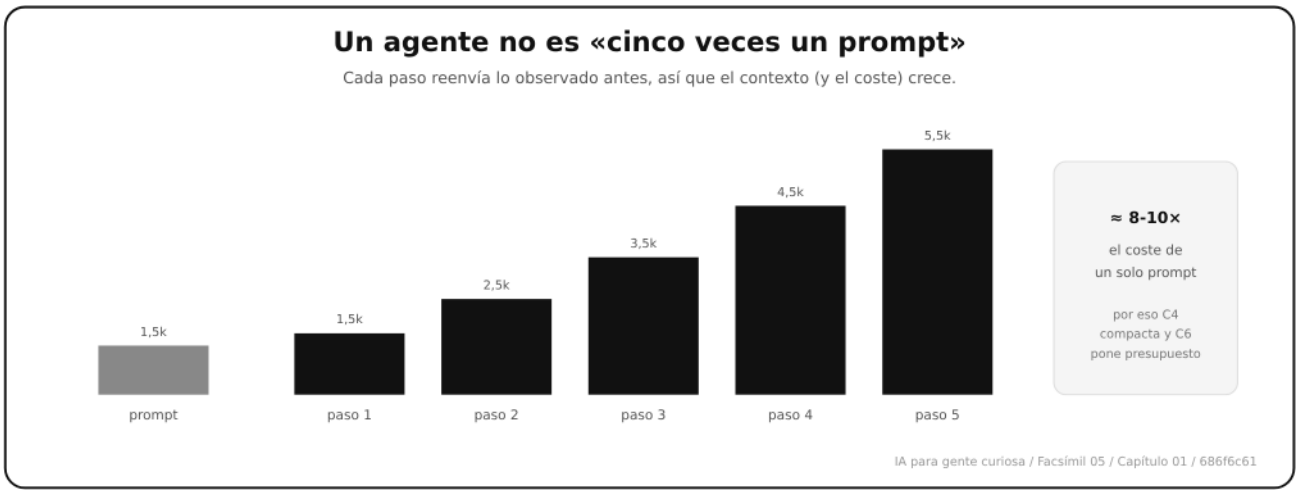
La regla al elegir no es «cuál tiene más estrellas», sino: ¿me deja ver la trayectoria, controlar permisos por acción y fijar un presupuesto? Si un framework esconde esas tres cosas, te quita justo lo que necesitas para operar el agente.

¿Cuánto cuesta un agente? una cuenta rápida

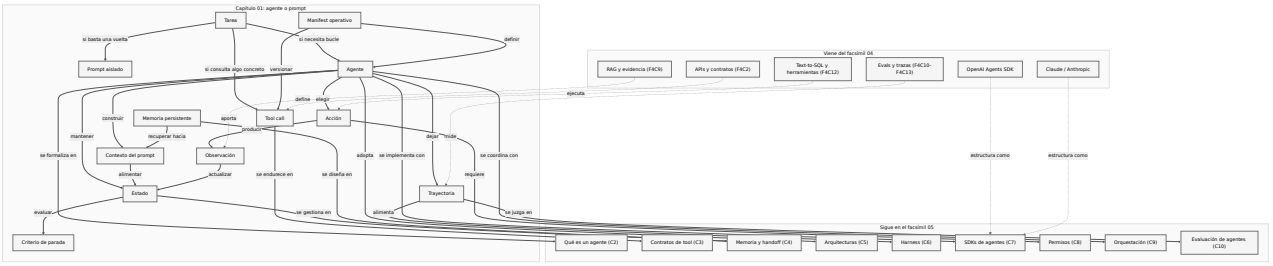
Un prompt aislado es una llamada. Un agente es un bucle: cada paso suele ser, al menos, otra llamada al modelo que arrastra el contexto acumulado. Esa es la factura escondida. Una estimación ilustrativa, con precios de orden de magnitud:

DISEÑO	LLAMADAS	TOKENS DE ENTRADA POR LLAMADA	COSTE RELATIVO
Prompt aislado	1	1.500	1×
Agente de 5 pasos (contexto que crece)	5	1.500 → 5.500	8-10×

El contexto que crece es la clave: en el paso 5 el agente reenvía todo lo observado en los pasos 1 a 4. Por eso el coste de un agente no es «cinco veces un prompt», sino más, y por eso los capítulos siguientes insisten en compactar el contexto (C4) y poner presupuesto (C6). La pregunta de diseño es la del valor de la información: ¿el siguiente paso reduce bastante la incertidumbre como para pagar su coste?[24] Si una acción cuesta más de lo que aclara, un agente bien diseñado no la toma.



Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Prompt aislado	Petición cerrada a un modelo, sin trayectoria ni acciones externas.
Agente	Sistema que decide acciones, usa herramientas, observa resultados y avanza hacia un objetivo.
Estado	Memoria operativa verificable de lo que sabe, hizo y aún puede hacer el sistema.
Contexto del prompt	Texto y artefactos que entran en la llamada actual al modelo.
Memoria de sesión	Historial persistido de una conversación o ejecución concreta.
Memoria persistente	Hechos, preferencias o decisiones recuperables más allá de una sesión.
Acción	Paso ejecutable o consultivo: tool, cálculo, lectura, pregunta o respuesta.
Observación	Resultado que vuelve al sistema después de una acción.
Trayectoria	Secuencia de estados, acciones y observaciones.
Criterio de parada	Regla que decide si terminar, repetir, pedir permiso o escalar a una persona.
Presupuesto operativo	Límite de pasos, tokens, coste, latencia o herramientas.
Manifest operativo	Especificación versionable de objetivo, tools, permisos, memoria, trazas y parada.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Llamar agente a cualquier prompt largo	Un prompt largo no observa resultados ni ejecuta acciones.	Preguntar si existe trayectoria: estado, acción, observación y parada.
Añadir agente antes de medir	La complejidad puede tapar un problema que resolvía una tool simple.	Probar primero prompt o tool call y medir el cuello real.
Dar herramientas genéricas	Una tool demasiado amplia es difícil de auditar y controlar.	Diseñar herramientas con nombre de dominio, schema, límites y errores claros.
Confundir contexto con estado	El contexto son tokens; el estado debe ser verificable y persistente cuando importa.	Guardar decisiones, permisos, artefactos y resultados fuera del prompt.
Confundir memoria con prompt largo	El sistema arrastra información sin saber si sigue siendo válida.	Tratar la memoria como store: escritura, búsqueda, actualización, caducidad y auditoría.
Evaluar solo la respuesta final	Un resultado correcto puede haber llegado por una trayectoria mala.	Medir outcome y trayectoria: herramientas, observaciones, coste y parada.

Antes de pasar página

- [ ] ¿Puedo explicar por qué un agente no es simplemente un prompt largo?
- [ ] ¿Sé distinguir prompt aislado, tool call y agente?
- [ ] ¿Puedo escribir qué contiene s_t en un agente sencillo?
- [ ] ¿Entiendo qué significa elegir $a_t \in A(s_t)$?
- [ ] ¿Puedo explicar por qué una observación debe ser estructurada?
- [ ] ¿Sé distinguir contexto del prompt, estado de sesión y memoria persistente?
- [ ] ¿Puedo explicar por qué una memoria debe poder corregirse o caducar?
- [ ] ¿Puedo leer un sistema tipo OpenAI Agents SDK o Claude Code y señalar instrucciones, tools, estado, memoria, permisos y trazas?
- [ ] ¿Puedo escribir un manifest operativo mínimo antes de implementar el agente?
- [ ] ¿Sé decir cuándo una tarea necesita varias vueltas?

- [ ] ¿Puedo justificar por qué los permisos no los decide el modelo?
- [ ] ¿Sé qué métrica miraría antes de complicar una solución?

En resumen

IDEA FUERZA	DETALLE
Un agente es una trayectoria, no una respuesta.	La diferencia está en estado, acciones, observaciones y criterio de parada.
No todo merece agente.	Si una sola llamada o una tool conocida resuelven el problema, empezar ahí suele ser mejor.
Memoria no es contexto acumulado.	La memoria útil se recupera, se versiona, se corrige y se audita fuera del prompt.
OpenAI y Claude usan piezas reconocibles.	Cambia el nombre de la API, pero siguen apareciendo instrucciones, tools, estado, memoria, controles y trazas.
La autonomía se diseña por grados.	Leer, redactar, escribir, enviar o modificar no tienen el mismo permiso ni el mismo coste.
Las trazas son parte del producto.	Sin trayectoria observable, no puedes depurar ni evaluar bien un agente.

Para saber más

Anthropic. (2024). *Building Effective Agents*. Artículo técnico.

Anthropic. (2026). *How to implement tool use*. Documentación oficial.

Anthropic. (2026). *Manage Claude's memory*. Documentación oficial.

Anthropic. (2026). *Subagents*. Documentación oficial.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

Google. (2026). *Agent Development Kit: Memory*. Documentación oficial.

Howard, R. A. (1966). Information Value Theory. *IEEE Transactions on Systems Science and Cybernetics*, 2(1), 22-26. https://doi.org/10.1109/TSSC.1966.300074

Kaelbling, L. P., Littman, M. L. y Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2), 99-134. https://doi.org/10.1016/S0004-3702(98)00023-X

LangChain. (2026). *LangGraph persistence*. Documentación oficial.

LlamaIndex. (2026). *Agentic strategies*. Documentación oficial.

Newell, A., Shaw, J. C. y Simon, H. A. (1959). *Report on a General Problem-Solving Program*. Proceedings of the International Conference on Information Processing, 256-264.

Nilsson, N. J. (1998). *Artificial intelligence: a new synthesis*. Morgan Kaufmann.

OpenAI. (2026). *Agents SDK*. Documentación oficial.

OpenAI. (2026). *Agents SDK: Agents*. Documentación oficial.

OpenAI. (2026). *Agents SDK: Sessions*. Documentación oficial.

OpenAI. (2026). *Agents SDK: Tracing*. Documentación oficial.

OpenAI. (2026). *Function calling*. Documentación oficial.

Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson.

Saltzer, J. H. y Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1278-1308. https://doi.org/10.1109/PROC.1975.9939

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N. y Scialom, T. (2023). *Toolformer: Language Models Can Teach Themselves to Use Tools*. https://doi.org/10.48550/arXiv.2302.04761

Sutton, R. S. y Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2.ª ed.). MIT Press.

von Neumann, J. y Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. y Cao, Y. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. International Conference on Learning Representations. https://arxiv.org/abs/2210.03629

Notas

1. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. Su definición de agente como sistema que percibe y actúa es el punto de partida conceptual de este facsímil. ↩

2. Nilsson, N. J. (1998). *Artificial intelligence: a new synthesis*. Morgan Kaufmann. Nilsson presenta agentes y planificación como problemas de acción, estado y objetivo. ↩

3. Newell, A., Shaw, J. C. y Simon, H. A. (1959). *Report on a General Problem-Solving Program*. Proceedings of the International Conference on Information Processing, 256-264. Es una referencia clásica para entender la idea de descomponer una tarea en estados, diferencias y operadores. ↩

4. OpenAI. (2026). *Agents SDK: Sessions*. Documentación oficial. Consultado el 10 de junio de 2026. La documentación describe sesiones como memoria de conversación entre runs y lista backends de persistencia. ↩

5. Google. (2026). *Agent Development Kit: Memory*. Documentación oficial. Consultado el 10 de junio de 2026. La documentación distingue memoria de corto plazo basada en sesión/estado y conocimiento de largo plazo mediante servicios de memoria. ↩

6. LangChain. (2026). *LangGraph persistence*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

7. Yao, S. y otros (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. International Conference on Learning Representations. https://arxiv.org/abs/2210.03629 ↩

8. Schick, T. y otros (2023). *Toolformer: Language Models Can Teach Themselves to Use Tools*. https://doi.org/10.48550/arXiv.2302.04761 ↩

9. OpenAI. (2026). *Function calling*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

10. OpenAI. (2026). *Agents SDK*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

11. LlamaIndex. (2026). *Agentic strategies*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

12. Sutton, R. S. y Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2.ª ed.). MIT Press, cap. 3. Define el MDP $\langle S, A, P, R, \gamma \rangle$ y la ecuación de optimalidad que estructura cualquier agente secuencial. ↩

13. Kaelbling, L. P., Littman, M. L. y Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2), 99-134. https://doi.org/10.1016/S0004-3702(98)00023-X Formaliza el POMDP $\langle S, A, T, R, \Omega, O, \gamma \rangle$ y la decisión sobre creencias. ↩

14. Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. La ecuación de optimalidad define la política óptima de un proceso de decisión secuencial; Sutton y Barto (2018, cap. 3) la desarrollan para Q^*. ↩

15. von Neumann, J. y Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press. Establece la maximización de la utilidad esperada como criterio del agente racional. ↩

16. Saltzer, J. H. y Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1278-1308. https://doi.org/10.1109/PROC.1975.9939 ↩

17. OpenAI. (2026). *Agents SDK: Agents*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

18. OpenAI. (2026). *Agents SDK: Sessions*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

19. OpenAI. (2026). *Agents SDK: Tracing*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

20. Anthropic. (2024). *Building Effective Agents*. Artículo técnico. Consultado el 10 de junio de 2026. ↩

21. Anthropic. (2026). *How to implement tool use*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

22. Anthropic. (2026). *Manage Claude's memory*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

23. Anthropic. (2026). *Subagents*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

24. Howard, R. A. (1966). Information Value Theory. *IEEE Transactions on Systems Science and Cybernetics*, 2(1), 22-26. https://doi.org/10.1109/TSSC.1966.300074 Formaliza cuánto vale, como mucho, obtener más información antes de decidir. ↩