Facsímil 05

Agentes y orquestación

Capítulo 02

Qué es un agente: estado, acción y observación

686f6c61 · 2026

Capítulo 02: Qué es un agente: estado, acción y observación

El bucle que convierte un modelo en sistema

En el capítulo anterior distinguimos prompt, tool call, workflow y agente. Ahora toca abrir la caja del agente con calma.

Un agente no se define por lo impresionante que suena su respuesta. Se define por un bucle: mantiene un estado, elige una acción, recibe una observación, actualiza el estado y decide si continuar. Esa frase parece simple, pero es la frontera entre “un modelo que contesta” y “un sistema que trabaja”.

Si lo llevamos a una escena concreta: una persona pide “revisa por qué no puedo matricularme”. Un modelo puede redactar consejos generales. Un agente puede consultar la política, revisar pagos, comprobar documentación pendiente, preparar una respuesta y detenerse antes de enviar nada si falta aprobación. La diferencia está en la secuencia observable.

Qué no queremos llamar agente todavía

No llamaremos agente a una cadena de prompts sin estado verificable. Si el sistema no sabe qué ocurrió en el paso anterior salvo por texto acumulado, todavía no hay una estructura operativa sólida.

Tampoco llamaremos agente a una función que siempre ejecuta los mismos pasos. Eso puede ser un workflow magnífico, y muchas veces es justo lo que conviene. Pero si el orden está cerrado de antemano y el modelo no decide el siguiente paso a partir de observaciones, no necesitamos la palabra agente.

Y no llamaremos agente a una herramienta con nombre bonito. Una tool consulta, calcula o cambia algo. Un agente decide cuándo usarla, interpreta la observación, actualiza estado y decide si falta otra acción.

La definición útil

Russell y Norvig formulan los agentes como sistemas que reciben percepciones y ejecutan acciones sobre un entorno, guiados por una medida de rendimiento.[1] Nilsson presenta la acción inteligente como selección de operadores sobre estados para alcanzar objetivos.[2] Esa

definición clásica sigue viva. Lo que ha cambiado es que ahora el LLM puede participar en interpretar el objetivo, elegir acciones y resumir observaciones.

Para este facsímil, una definición operativa será:

> Un agente es un sistema que mantiene un estado verificable, elige acciones disponibles, recibe observaciones estructuradas, actualiza su estado y decide cuándo parar según un objetivo, permisos, presupuesto y evidencia.

Newell, Shaw y Simon ya planteaban resolución de problemas como selección de operadores para reducir diferencias entre estado actual y meta.[3] La novedad práctica de los agentes con LLM no es que exista una secuencia de pasos. Es que el lenguaje natural, las tools y el contexto hacen que el espacio de acciones sea mucho más flexible y, por tanto, más difícil de controlar.

Agente clásico y agente moderno

La comparación ayuda a no perderse:

PIEZA	AGENTE CLÁSICO	AGENTE CON LLM	PREGUNTA DE INGENIERÍA
Percepción	Sensores o entrada formal.	Mensajes, documentos, resultados de tools, memoria recuperada.	¿Qué entra como dato y qué entra como instrucción?
Estado	Variables del entorno.	Objetivo, historial, observaciones, permisos, presupuesto, memoria.	¿Qué debe persistir fuera del prompt?
Acción	Operador definido en el dominio.	Tool call, pregunta al usuario, cálculo, lectura, respuesta, handoff.	¿Qué acciones están permitidas desde este estado?
Política	Regla, búsqueda, planificador o aprendizaje.	LLM más reglas, router, validadores y límites.	¿Quién decide el siguiente paso y con qué restricciones?
Observación	Nuevo percepto o resultado del operador.	Salida estructurada de tool, error, evidencia, diff, test, métrica.	¿La observación permite actualizar estado sin ambigüedad?
Parada	Meta alcanzada o fallo.	Done, falta evidencia, falta permiso, límite de coste, revisión humana.	¿Podemos demostrar por qué terminó?

Anthropic distingue workflows y agents: los workflows siguen rutas predefinidas por código; los agents dejan que el LLM dirija dinámicamente el proceso y el uso de herramientas.[4] Esa distinción encaja perfectamente con nuestra definición: si no hay decisión dinámica sobre la siguiente acción, probablemente estás diseñando un workflow, no un agente.

La anatomía formal

El formalismo del capítulo anterior, el POMDP, nos da la anatomía exacta de un agente. No inventamos una tupla: usamos la que la literatura definió para decidir bajo observación parcial, $\langle S, A, T, R, \Omega, O, \gamma \rangle$.[5]

$$\langle S, A, T, R, \Omega, O, \gamma \rangle$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
S	Estados posibles del sistema.	Política leída, saldo consultado, respuesta preparada.
A	Acciones posibles.	Buscar política, consultar saldo, preparar respuesta, pedir aprobación.
$T(s' \mid s, a)$	Transición: probabilidad de pasar a s'.	Tras consultar saldo, el estado incorpora «pago pendiente».
$R(s, a)$	Recompensa de actuar.	Resolver bien la consulta, descontando coste y riesgo.
Ω	Observaciones posibles.	«saldo pendiente: 180 EUR», «política vigente encontrada».
$O(o \mid s', a)$	Probabilidad de observar o tras llegar a s'.	Una tool fiable emite observaciones informativas del estado.
γ	Descuento del futuro.	Cuánto pesa terminar pronto.

En palabras: un agente es estados, acciones, cómo se transita entre ellos, qué recompensa cada paso, qué se puede observar, cuán informativa es la observación y cuánto importa el futuro. En ingeniería añadimos tres refinamientos que la teoría deja implícitos: un objetivo explícito, un **presupuesto** finito (lo que convierte el problema en un MDP con restricciones)[6] y un criterio de parada.

La política mapea la creencia (no el estado real, que el agente no ve) a una acción:

$$a_t = \pi(b_t)$$

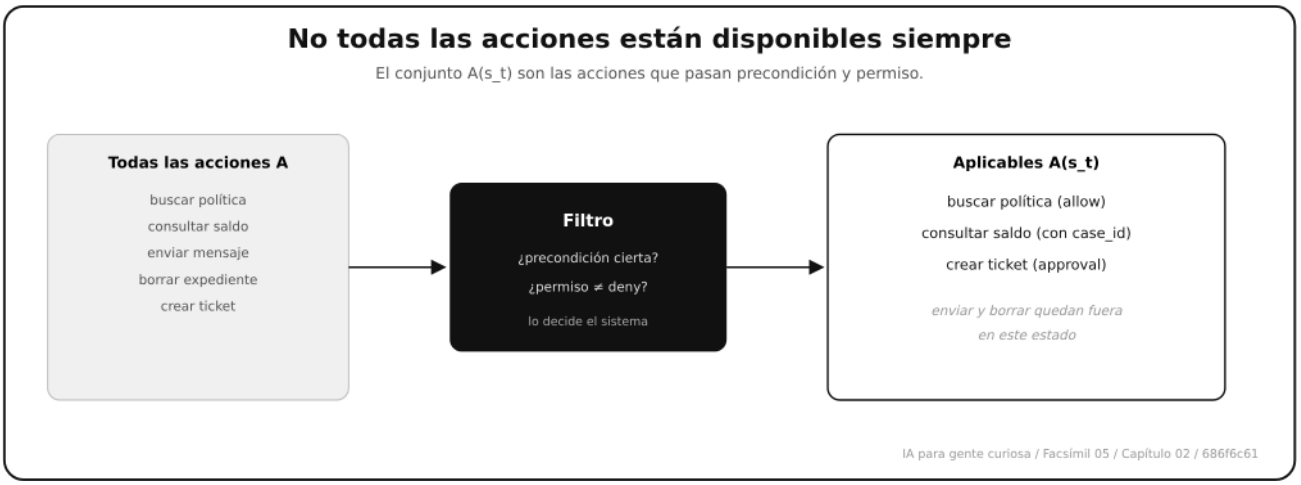
SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
π	Política de decisión (regla, LLM o combinación).	El modelo elige la siguiente tool.
b_t	Creencia: estimación del estado a partir de las observaciones.	Ya se leyó política; se cree que falta consultar saldo.
a_t	Acción elegida en el paso t.	`consultar_saldo_expediente`.

Las acciones disponibles no son todas las imaginables: son las que cumplen sus precondiciones y su permiso. Esta es la noción de **acción aplicable** de la planificación clásica, donde una acción se define por sus precondiciones y efectos.[7]

$$A(s_t) = \{\, a \in A \mid \mathrm{pre}(a, s_t) \ \wedge\ \mathrm{perm}(a, s_t) \in \{\mathrm{allow}, \mathrm{approval}\} \,\}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
$A(s_t)$	Acciones aplicables desde el estado actual.	Buscar política y consultar saldo, pero no enviar mensaje.
$\mathrm{pre}(a, s_t)$	Precondición de la acción.	Para consultar saldo hace falta `case_id`.
$\mathrm{perm}(a, s_t)$	Permiso de la acción.	`allow`, `approval`, `deny`.

En palabras: una acción solo entra en juego si su precondición se cumple y su permiso lo autoriza; el permiso lo decide el sistema, no el modelo.



Tras actuar, el entorno emite una observación según la función de observación del POMDP, y la creencia se actualiza:

$$o_{t+1} \sim O(\cdot \mid s_{t+1}, a_t), \qquad b_{t+1} = \tau(b_t, a_t, o_{t+1})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
o_{t+1}	Observación posterior a la acción.	Resultado de la consulta, error de schema, test fallido.
τ	Actualización de creencia (belief update).	Incorpora la observación, marca pasos hechos, descuenta presupuesto.
b_{t+1}	Nueva creencia.	Incluye política leída, saldo pendiente y respuesta preparada.

En palabras: la acción produce una observación y esa observación revisa la creencia; ese ciclo, no una sola llamada, es lo que distingue a un agente. Poole, Mackworth y Goebel separan representación, razonamiento y acción por la misma razón: el modelo propone, pero el sistema representa, ejecuta, valida y registra.[8]

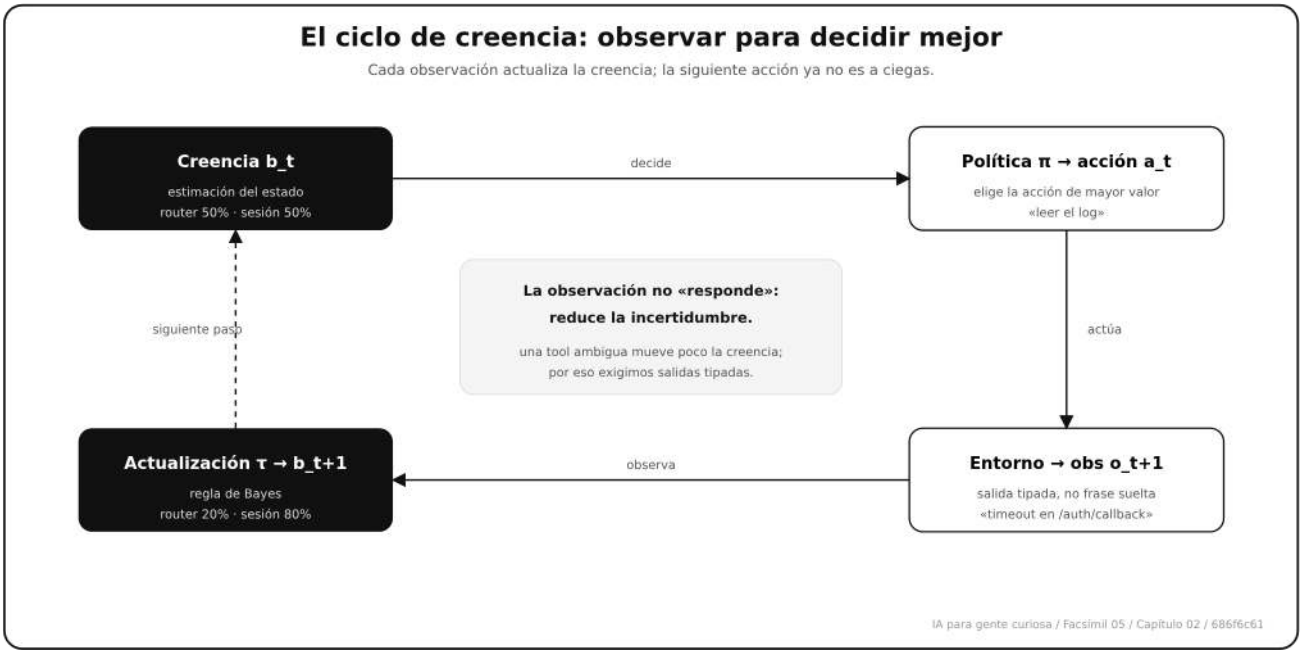
Un ejemplo de creencia con números

La creencia suena abstracta hasta que se pone un número. Un agente de código diagnostica un login que falla. No sabe la causa: duda entre dos estados, «el fallo está en el router» y «el fallo está en la sesión». Sin evidencia, reparte su creencia a partes iguales: 50% y 50%.

Entonces ejecuta una acción barata, leer el log, y recibe una observación: `timeout en /auth/callback`. Esa observación es mucho más probable si el problema está en la sesión que si está en el router. Al aplicar la actualización de creencia (la regla de Bayes que hay detrás de τ), la creencia se desplaza:

ESTADO POSIBLE	CREENCIA ANTES	OBSERVACIÓN	CREENCIA DESPUÉS
Fallo en el router	50%	poco compatible con un timeout de callback	20%
Fallo en la sesión	50%	muy compatible con un timeout de callback	80%

La acción siguiente ya no es a ciegas: con la sesión al 80%, el agente mira el manejo de sesión antes que el router. Eso es lo que hace una observación útil: no «da la respuesta», reduce la incertidumbre lo bastante para que la siguiente decisión sea mejor. Una tool cuya salida no mueve la creencia (ambigua, sin estructura) es, en términos de POMDP, una observación poco informativa, y por eso insistimos en salidas tipadas.



El contrato de observación: por qué la salida tipada importa tanto

Hay una pieza del bucle que casi siempre se subestima y que, sin embargo, decide si un agente es fiable o un castillo de naipes: la **observación**. En el formalismo del POMDP, la observación es lo único que conecta al agente con el mundo real; es la señal de la que depende la función de observación *O* y, con ella, toda la actualización de creencia. Si esa señal es buena, la creencia se afina; si es ambigua, la creencia se ensucia, y a partir de ahí cada decisión hereda el ruido. Por eso conviene tratar la observación no como «lo que devuelve una función», sino como un contrato que el sistema impone al mundo.

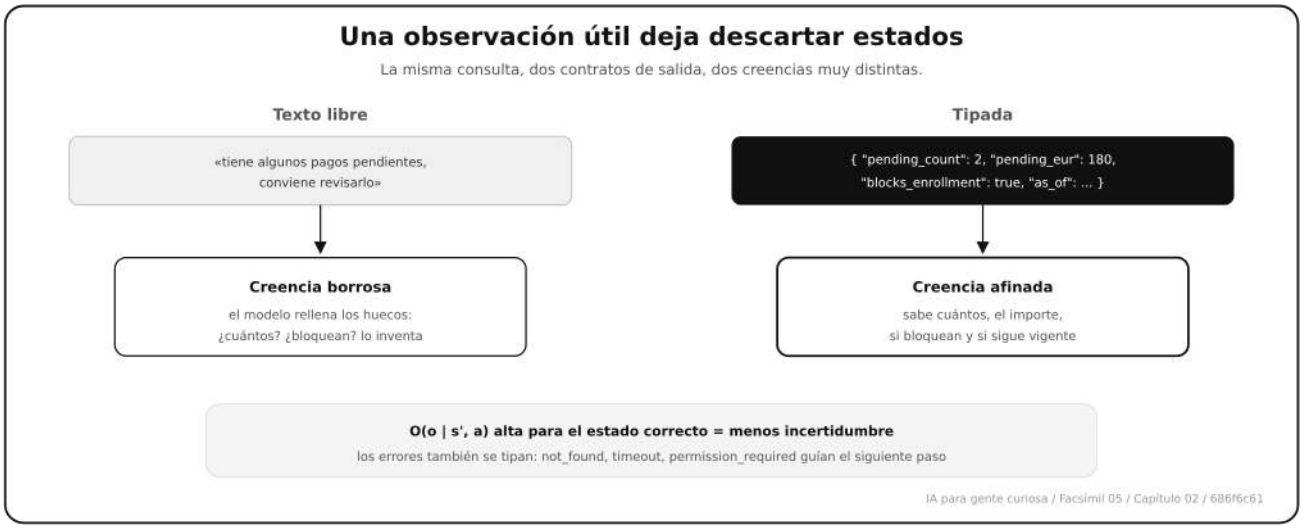
Veámoslo con un contraste que aparece en cualquier proyecto real. Un agente de soporte consulta el saldo de un expediente. Una tool mal diseñada le devuelve una frase: «El alumno tiene algunos pagos pendientes, conviene revisarlo». Suena informativo, pero para el agente es casi inútil. ¿Cuántos pagos? ¿De qué importe? ¿Bloquean la matrícula o no? El modelo, ante esa vaguedad, hará lo que mejor sabe hacer con el lenguaje: rellenar los huecos con algo plausible. Y «plausible» no es «verdadero». La creencia se actualiza, sí, pero hacia un estado inventado.

Ahora la misma tool, bien diseñada, devuelve una observación tipada: `{"pending_count": 2, "pending_eur": 180.0, "blocks_enrollment": true, "as_of": "2026-06-10"}`. La diferencia no es estética. Esta observación es **discriminativa**: distingue con claridad entre estados del mundo que antes se confundían. El agente ya no tiene que adivinar si los pagos bloquean la matrícula, porque el campo `blocks_enrollment` se lo dice; ya no tiene que inventar una cifra, porque `pending_eur` está ahí; y `as_of` le permite saber si el dato sigue vigente o si está

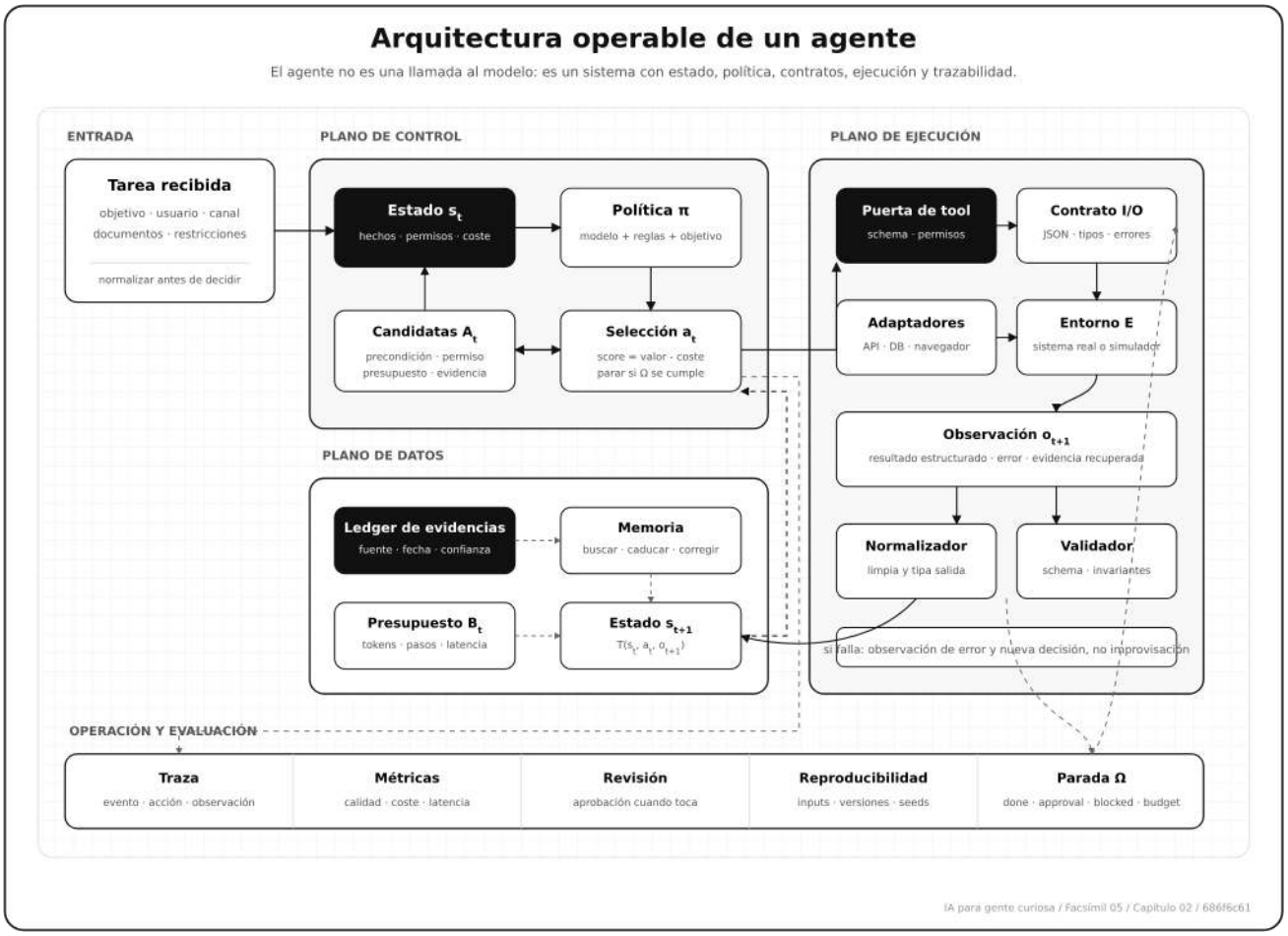
consultando una foto antigua. En términos del POMDP, una observación así hace que $O(o \mid s', a)$ sea muy alta para el estado correcto y baja para los demás, que es justo lo que reduce la incertidumbre.

La lección práctica se resume en una regla que conviene grabar: **una observación útil no es la que el modelo puede leer, sino la que le permite descartar estados**. Tres exigencias salen de ahí. Primero, la salida debe ser estructurada (un objeto con campos tipados), no prosa, porque la prosa invita a la interpretación libre. Segundo, debe llevar metadatos de confianza y de tiempo, porque una observación sin fecha no permite distinguir lo vigente de lo caduco. Tercero, los errores también son observaciones y también se tipan: `not_found`, `timeout` o `permission_required` le dicen al agente exactamente qué estado del mundo ha encontrado y qué acción siguiente tiene sentido. Un agente que recibe «algo ha fallado» no puede decidir; uno que recibe `timeout` sabe que puede reintentar, y uno que recibe `permission_required` sabe que debe pedir aprobación en vez de insistir.

Cuando en los próximos capítulos hablemos de contratos de tool (capítulo 3), de harness y trazas (capítulo 6) o de evaluación (capítulo 10), estaremos en el fondo protegiendo esta misma idea: que cada observación que entra en el estado sea un dato del que el agente pueda fiarse, y no una frase que el modelo tenga que creerse.



La arquitectura operable de un agente



El diagrama ya no mira solo el bucle conceptual; mira el sistema que habría que operar. Primera idea: el modelo no debería ejecutar directamente, sino proponer una acción que pasa por contrato, permisos, presupuesto y validación. Segunda idea: cada observación vuelve al estado como dato tipado, no como frase suelta. Tercera idea: sin traza, métricas y criterios de parada, no sabes si el agente resolvió, pidió aprobación, agotó presupuesto o quedó bloqueado.

En el día a día

Piensa en un agente de código. Recibe: “la página de login falla”. Si su estado solo contiene esa frase, puede hacer casi cualquier cosa. Si el estado contiene navegador abierto, error de consola, archivos relevantes, tests ejecutados, presupuesto y permisos, la siguiente acción es mucho más razonable.

El agente prudente no edita a ciegas. Primero observa: lee el error, localiza el componente, ejecuta un test pequeño. Luego actúa: cambia una función concreta. Después observa otra vez: el test pasa o falla. Ese ciclo es lo que queremos enseñar.

ReAct formalizó una manera de intercalar razonamiento y acción en modelos de lenguaje, mostrando que alternar pasos de decisión con observaciones de herramientas puede mejorar tareas que requieren información externa.[9] Toolformer mostró otra dirección: modelos que aprenden a decidir cuándo usar herramientas como calculadoras, buscadores o sistemas externos.[10] En ambos casos, lo importante no es la etiqueta del paper: es separar decidir, actuar y observar.

Por qué debería importarte

Si no defines estado, el agente decide con memoria borrosa. Si no defines acciones disponibles, cualquier tool parece válida. Si no defines observación, una salida textual puede parecer evidencia aunque no lo sea. Si no defines parada, el sistema puede seguir gastando o declarar éxito antes de tiempo.

Hart, Nilsson y Raphael mostraron con A* que combinar coste acumulado y estimación restante permite guiar la búsqueda de manera más disciplinada.[11] En agentes modernos no siempre implementamos A*, pero la intuición sigue siendo valiosa: cada acción debe justificar qué aporta y qué cuesta.

OpenAI Agents SDK estructura agentes con instrucciones, herramientas, handoffs, guardrails, output types y trazas.[12] También registra eventos como generation spans, function spans, handoff spans y guardrail spans.[13] Eso confirma una idea práctica: los agentes se operan mirando trayectoria, no solo respuesta final.

Cómo lo manejan OpenAI, Claude y OpenCode

OpenAI te empuja a pensar el agente como una unidad con instrucciones, modelo, herramientas, posibles handoffs, guardrails y tipo de salida.[14] Es una forma cómoda de convertir nuestra tupla $\mathcal{A}$ en código: las instrucciones describen G, las tools definen A, los esquemas de salida ayudan a tipar O, y la traza deja visible qué ocurrió entre s_t y s_{t+1}.

Claude, visto desde la API, trabaja con un patrón más explícito de tool use: tú declaras herramientas con nombre, descripción y esquema de entrada; el modelo puede pedir un `tool_use`; tu aplicación ejecuta la herramienta y devuelve un `tool_result` en la conversación.[15] En Claude Code aparece otra pieza muy útil para pensar sistemas reales: memoria mediante archivos `CLAUDE.md` y subagentes definidos como Markdown con frontmatter, herramientas permitidas y contexto separado.[16][17]

OpenCode encaja bien para explicar esta idea a ingeniería porque separa agentes configurables y plugins. Los agentes se pueden definir como perfiles especializados con herramientas y permisos, y los plugins permiten añadir comportamiento o herramientas propias al entorno.[18][19] Dicho de forma práctica: no basta con “un modelo que sabe escribir”; queremos una mesa de trabajo donde cada especialista tenga una misión, un contrato y una traza.

ENTORNO	UNIDAD PRINCIPAL	CÓMO APARECEN A_T Y O_T	QUÉ DEBE PONER EL INGENIERO
OpenAI Agents SDK	`Agent` ejecutado por `Runner`.	Tools, handoffs, output types y eventos de tracing.	Instrucciones, esquemas, límites, validadores y observabilidad.
Claude API	Loop de mensajes con `tool_use` y `tool_result`.	La aplicación ejecuta tools y devuelve resultados al modelo.	Estado externo, permisos, parseo de tool results y criterio de parada.
Claude Code	Subagentes Markdown y memoria `CLAUDE.md`.	Cada subagente recibe una tarea y un conjunto de herramientas.	Fronteras de responsabilidad, herramientas permitidas y contexto persistente.
OpenCode	Agentes configurables y plugins.	Agentes especializados más herramientas añadidas por plugin.	Ficheros de agente, plugin, contratos de entrada/salida y gates.

Crear el mismo diseño en OpenAI y Claude

Imagina un plugin editorial para este facsímil. Recibe un fragmento de capítulo y una lista de citas. Queremos tres agentes:

AGENTE	OBJETIVO	ENTRADA	SALIDA
`rae_normas`	Revisar ortografía, puntuación, mayúsculas, comillas y usos dudosos según norma panhispánica.[20]	Texto del capítulo.	Lista de hallazgos con ubicación, explicación y propuesta.
`apa_citas`	Convertir o revisar referencias en APA 7.[21]	Metadatos de fuentes y citas en texto.	Referencias normalizadas y errores de campo.
`verificador_browser`	Abrir la URL citada y comprobar si el contenido respalda la afirmación.	URL, afirmación y fragmento citado.	`supported`, `partial` o `not_found`, con evidencia.

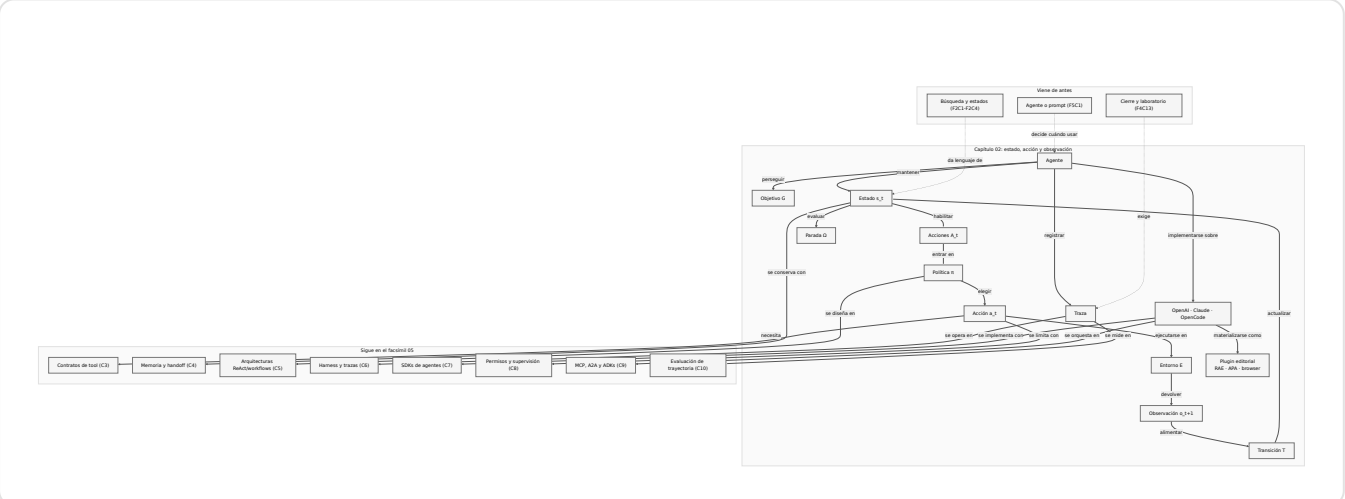
En OpenAI lo natural es crear tres `Agent` especializados y un cuarto agente coordinador. Los subagentes pueden exponerse como herramientas del coordinador. La idea importante no es la sintaxis exacta, sino el reparto de responsabilidad:

```python
from pydantic import BaseModel
from agents import Agent, Runner, function_tool, trace


class HallazgoRAE(BaseModel):
    ubicacion: str
    problema: str
    propuesta: str


class RevisionAPA(BaseModel):
    referencia_normalizada: str
    errores: list[str]


class VerificacionFuente(BaseModel):
    url: str
    estado: str
    evidencia: str


@function_tool
async def browser_check(url: str, afirmacion: str) -> VerificacionFuente:
    """Abre una URL y devuelve evidencia verificable para la afirmación."""
    ...


rae_normas = Agent(
    name="rae_normas",
    instructions="Revisa el texto con criterio RAE/ASALE. Devuelve hallazgos concretos.",
    output_type=list[HallazgoRAE],
)

apa_citas = Agent(
    name="apa_citas",
    instructions="Revisa referencias en APA 7. No inventes campos ausentes.",
    output_type=list[RevisionAPA],
)

verificador_browser = Agent(
    name="verificador_browser",
    instructions="Comprueba si la URL respalda la afirmación citada.",
    tools=[browser_check],
    output_type=list[VerificacionFuente],
)

editorial_plugin = Agent(
    name="editorial_plugin",
    instructions=(
        "Coordina tres revisiones: norma lingüística, APA 7 y verificación de fuentes. "
        "No apruebes el texto si una cita clave no queda respaldada."
    ),
```

```
    tools=[
        rae_normas.as_tool("revisar_rae", "Revisa norma lingüística."),
        apa_citas.as_tool("revisar_apa", "Revisa referencias APA 7."),
        verificador_browser.as_tool("verificar_fuentes", "Comprueba URLs citadas."),
    ],
)


async def revisar(texto: str):
    with trace("plugin_editorial_tres_agentes"):
        return await Runner.run(editorial_plugin, texto)
```

En Claude API no tienes que imitar esa clase `Agent`. Puedes crear el mismo comportamiento con un loop anfitrión: declaras tools `revisar_rae`, `revisar_apa` y `verificar_fuentes`; Claude pide una tool con `tool_use`; tu aplicación ejecuta la función; devuelves `tool_result`; y repites hasta que el estado diga `done`, `approval_required` o `blocked`. En Claude Code, el mismo reparto puede vivir como tres subagentes Markdown. La diferencia técnica es clara: OpenAI te da una abstracción de agente más directa; Claude te deja muy visible el protocolo de herramientas y, en Claude Code, el patrón de subagentes por archivo.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Agente	Sistema que mantiene estado, elige acciones, observa resultados y avanza hacia un objetivo.
Estado	Representación compacta de lo que importa para decidir la siguiente acción.
Acción	Operación disponible desde un estado concreto.
Observación	Resultado estructurado que vuelve después de actuar.
Política	Regla, modelo o combinación que elige la siguiente acción.
Función de transición	Actualización que produce el siguiente estado.
Precondición	Condición necesaria para que una acción esté disponible.
Subagente	Agente especializado que recibe una tarea parcial, herramientas concretas y contexto acotado.
Plugin	Extensión que añade herramientas, gates o comportamiento operativo al entorno de trabajo.
Puerta de calidad	Comprobación automática que decide si una salida puede seguir adelante o debe volver a revisión.
Criterio de parada	Regla que decide terminar, pedir ayuda, repetir o bloquearse.
Traza	Registro de trayectoria suficiente para depurar y evaluar.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Meter todo en el estado	El estado grande encarece y confunde la decisión.	Guardar solo lo necesario para elegir el siguiente paso.
No distinguir acción y observación	Una tool no es útil si su salida no actualiza el estado.	Diseñar cada tool con resultado estructurado y uso claro.
Tratar permiso como una frase del prompt	Los permisos deben aplicarse fuera del modelo.	Calcular A_t filtrando por precondición y permiso.
Parar cuando hay texto	Una respuesta redactada no siempre significa tarea terminada.	Definir `done`, `approval_required`, `blocked` y `budget_exhausted`.
Hacer un agente comodín	Si el mismo agente revisa lengua, referencias y fuentes, las observaciones se mezclan.	Separar subagentes por responsabilidad y unificar con una puerta de calidad.
Confundir SDK con arquitectura	OpenAI, Claude y OpenCode cambian la sintaxis, pero no la necesidad de estado y trazas.	Diseñar primero $G, S, A, O, \pi, T, \Omega, B$; después elegir proveedor.
No registrar la trayectoria	Sin eventos no sabes por qué el agente decidió lo que decidió.	Registrar acción, observación, permiso, coste y parada.

Antes de pasar página

- [ ] ¿Puedo definir un agente con $G, S, A, O, \pi, T, \Omega, B$?
- [ ] ¿Sé explicar la diferencia entre estado, contexto y memoria?
- [ ] ¿Puedo escribir A_t como acciones filtradas por precondiciones y permisos?
- [ ] ¿Entiendo por qué una observación debe ser estructurada?
- [ ] ¿Sé distinguir workflow fijo de agente con decisión dinámica?
- [ ] ¿Puedo explicar por qué el entorno ejecuta y el modelo propone?
- [ ] ¿Sé diseñar un criterio de parada que no sea solo “hay respuesta”?
- [ ] ¿Puedo leer una traza y reconstruir la trayectoria?
- [ ] ¿Sé traducir el mismo diseño a OpenAI Agents SDK, Claude API o OpenCode?

- [ ] ¿Puedo separar un plugin práctico en subagentes con responsabilidades verificables?

En resumen

IDEA FUERZA	DETALLE
Un agente es un bucle observable.	Estado, política, acción, entorno, observación y transición forman la unidad mínima.
El estado no es todo el contexto.	Es la representación compacta que permite decidir el siguiente paso.
Las acciones disponibles se filtran.	Precondiciones, permisos y presupuesto determinan A_t.
La observación cambia el futuro.	Si no actualiza el estado, la acción no aportó evidencia útil.
La parada también se diseña.	`done`, `approval_required`, `blocked` y `budget_exhausted` evitan falsa autonomía.
El proveedor no sustituye la arquitectura.	OpenAI, Claude y OpenCode ofrecen caminos distintos, pero todos necesitan estado, tools, trazas y contratos.
Un plugin útil separa responsabilidades.	RAE, APA y browser pueden ser tres agentes coordinados por una puerta de calidad común.

Para saber más

Altman, E. (1999). *Constrained Markov Decision Processes*. Chapman & Hall/CRC.

American Psychological Association. (2020). *Publication manual of the American Psychological Association: The official guide to APA style* (7.ª ed.). American Psychological Association.

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.

Fikes, R. E. y Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4), 189-208. https://doi.org/10.1016/0004-3702(71)90010-5

Hart, P. E., Nilsson, N. J. y Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107. https://doi.org/10.1109/TSSC.1968.300136

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

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: Tracing*. Documentación oficial.

OpenCode. (2026). *Agents*. Documentación oficial.

OpenCode. (2026). *Plugins*. Documentación oficial.

Poole, D., Mackworth, A. y Goebel, R. (1998). *Computational intelligence: a logical approach*. Oxford University Press.

Real Academia Española y Asociación de Academias de la Lengua Española. (2018). *Libro de estilo de la lengua española según la norma panhispánica*. Espasa.

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

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.

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. Los autores definen agentes racionales como sistemas que perciben y actúan, y conectan esa definición con funciones de agente y medidas de rendimiento. ↩

2. Nilsson, N. J. (1998). *Artificial intelligence: a new synthesis*. Morgan Kaufmann. Nilsson trabaja la relación entre estado, operador, planificación y agente. ↩

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. ↩

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

5. 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 Define el POMDP y la decisión sobre creencias. ↩

6. Altman, E. (1999). *Constrained Markov Decision Processes*. Chapman & Hall/CRC. Formaliza la decisión secuencial sujeta a límites de coste acumulado. ↩

7. Fikes, R. E. y Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4), 189-208. https://doi.org/10.1016/0004-3702(71)90010-5 Formaliza una acción por sus precondiciones y efectos, base del conjunto de acciones aplicables. ↩

8. Poole, D., Mackworth, A. y Goebel, R. (1998). *Computational intelligence: a logical approach*. Oxford University Press. Su marco separa representación, razonamiento y acción. ↩

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

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

11. Hart, P. E., Nilsson, N. J. y Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107. https://doi.org/10.1109/TSSC.1968.300136 ↩

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

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

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

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

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

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

.8. OpenCode. (2026). *Agents*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

.9. OpenCode. (2026). *Plugins*. Documentación oficial. Consultado el 10 de junio de 2026. ↩

:0. Real Academia Española y Asociación de Academias de la Lengua Española. (2018). *Libro de estilo de la lengua española según la norma panhispánica*. Espasa. https://www.rae.es/obras-academicas/obras-linguisticas/libro-de-estilo-de-la-lengua-espanola ↩

:1. American Psychological Association. (2020). *Publication manual of the American Psychological Association: The official guide to APA style* (7.ª ed.). American Psychological Association. ↩