Facsímil 05

Agentes y orquestación

Capítulo 05

Arquitecturas de agentes: de ReAct a sistemas multiagente

686f6c61 · 2026

Capítulo 05: Arquitecturas de agentes: de ReAct a sistemas multiagente

El patrón importa más que la etiqueta

En los capítulos anteriores ya tenemos las piezas: estado, acción, observación, tools, memoria y handoff. Ahora aparece una pregunta muy de ingeniería: **¿cómo las organizo?**

Una arquitectura agentic no es un nombre bonito. Es una decisión sobre el bucle: quién planifica, quién ejecuta, quién verifica, dónde vive la memoria, cuándo se consulta una tool, cuándo se pide ayuda y cómo se mide la trayectoria.

El repositorio de Fareed Khan, *All Agentic Architectures*, es útil precisamente porque convierte esas ideas en notebooks ejecutables. La colección declara implementaciones prácticas de arquitecturas agentic con LangChain y LangGraph, y ordena los patrones desde los más básicos hasta sistemas multiagente, memoria avanzada, simulación y metacognición.[1] Lo usaremos como catálogo práctico, no como autoridad única: cada patrón hay que traducirlo a nuestro lenguaje de $G, S, A, O, \pi, T, \Omega, B$.

Repositorio base de Fareed Khan: https://github.com/FareedKhan-dev/all-agentic-architectures. En este capítulo se han revisado los notebooks `.ipynb` del repositorio para explicar qué demuestra cada arquitectura y qué habría que añadir para llevarla a un sistema real.

La pregunta correcta

Antes de elegir una arquitectura, no preguntes “¿cuál es la más avanzada?”. Pregunta:

PREGUNTA	QUÉ DECIDE
¿La tarea se puede resolver en una sola pasada?	Si basta con prompt o si hace falta reflexión.
¿Necesita datos externos o cálculo exacto?	Si hace falta tool use.
¿El siguiente paso depende de lo observado?	Si conviene ReAct o Plan-Execute-Verify.
¿Hay varias habilidades claramente separables?	Si conviene multiagente, ensemble o meta-controlador.
¿La tarea depende de memoria persistente?	Si necesitas memoria episódica, semántica o grafo.
¿Actuar tiene coste alto?	Si necesitas dry-run, simulador o aprobación.
¿La arquitectura debe mejorar con feedback?	Si necesitas self-improvement o evaluación sistemática.

Elegir arquitectura es un problema de **decisión multicriterio**: se maximiza una función de valor aditiva que suma la utilidad y resta las penalizaciones ponderadas de coste, latencia y dificultad de verificación. Es el modelo aditivo de la teoría de la utilidad multiatributo.[2]

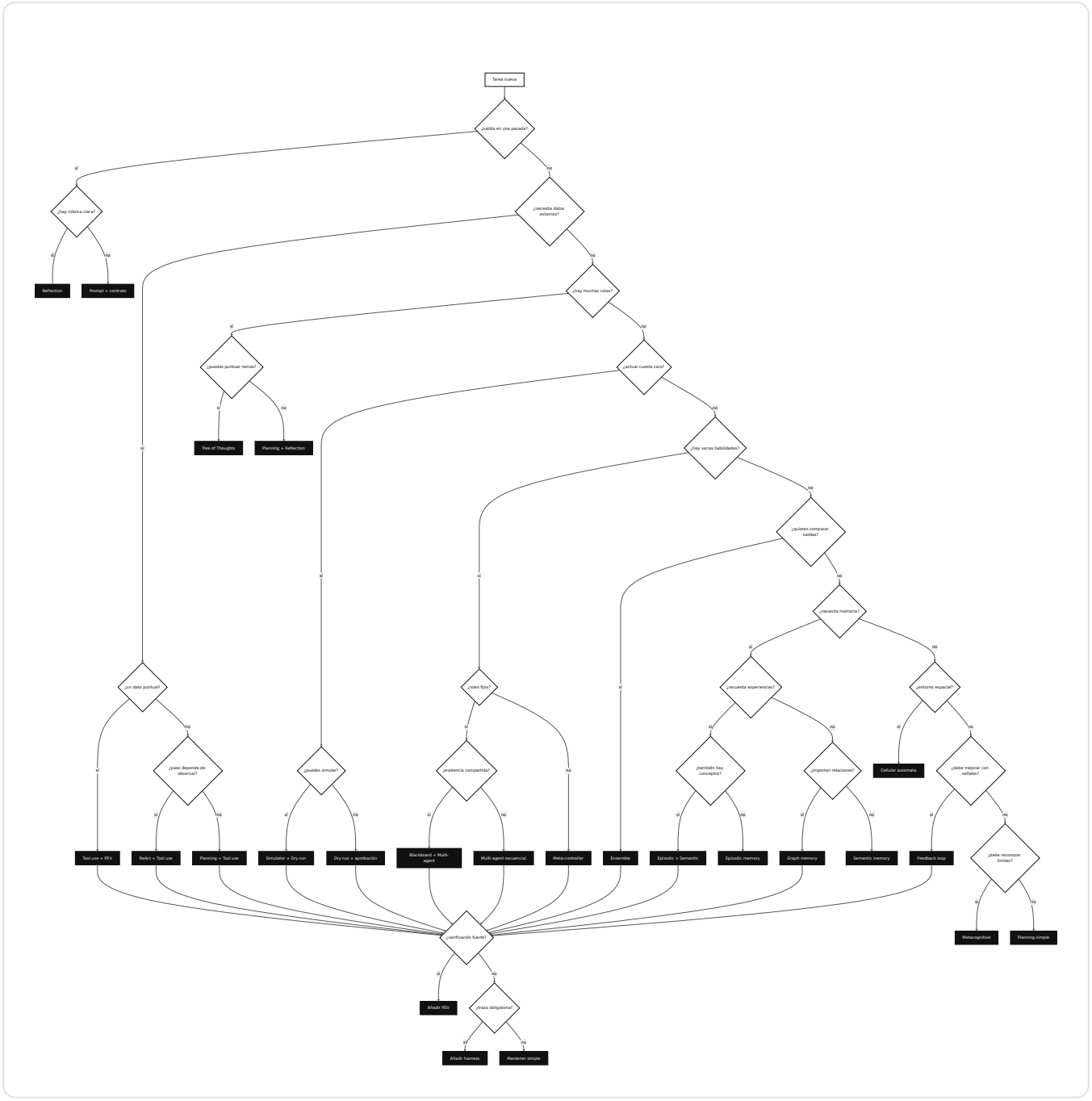
$$p^{\backslash *} = \arg\max_{p \in P} \left[U(p, x) - \lambda_c C(p) - \lambda_l L(p) - \lambda_v V(p) \right]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
p	Patrón candidato.	ReAct, planning, ensemble, graph memory.
P	Conjunto de patrones disponibles.	Las 25 arquitecturas del catálogo.
x	Tarea concreta.	“Verifica estas fuentes y corrige citas APA”.
$U(p, x)$	Utilidad esperada del patrón para esa tarea.	ReAct sube si hay que consultar páginas.
$C(p)$	Coste operativo.	Llamadas a modelo, tools, base vectorial, grafo.
$L(p)$	Latencia.	Un ensemble tarda más que una llamada simple.
$V(p)$	Dificultad de verificación.	Más agentes implican más trazas y más comparación.
λ	Peso de cada penalización.	En producción suele subir λ_l y λ_v.

En palabras: se elige el patrón cuyo valor neto (lo que aporta menos lo que cuesta, tarda y dificulta verificar) es mayor. No pretende automatizarlo todo. Sirve para no elegir arquitectura por entusiasmo. Una arquitectura más compleja solo compensa si aumenta utilidad más de lo que aumenta coste, latencia y dificultad de verificación.

Árbol de decisión para elegir arquitectura

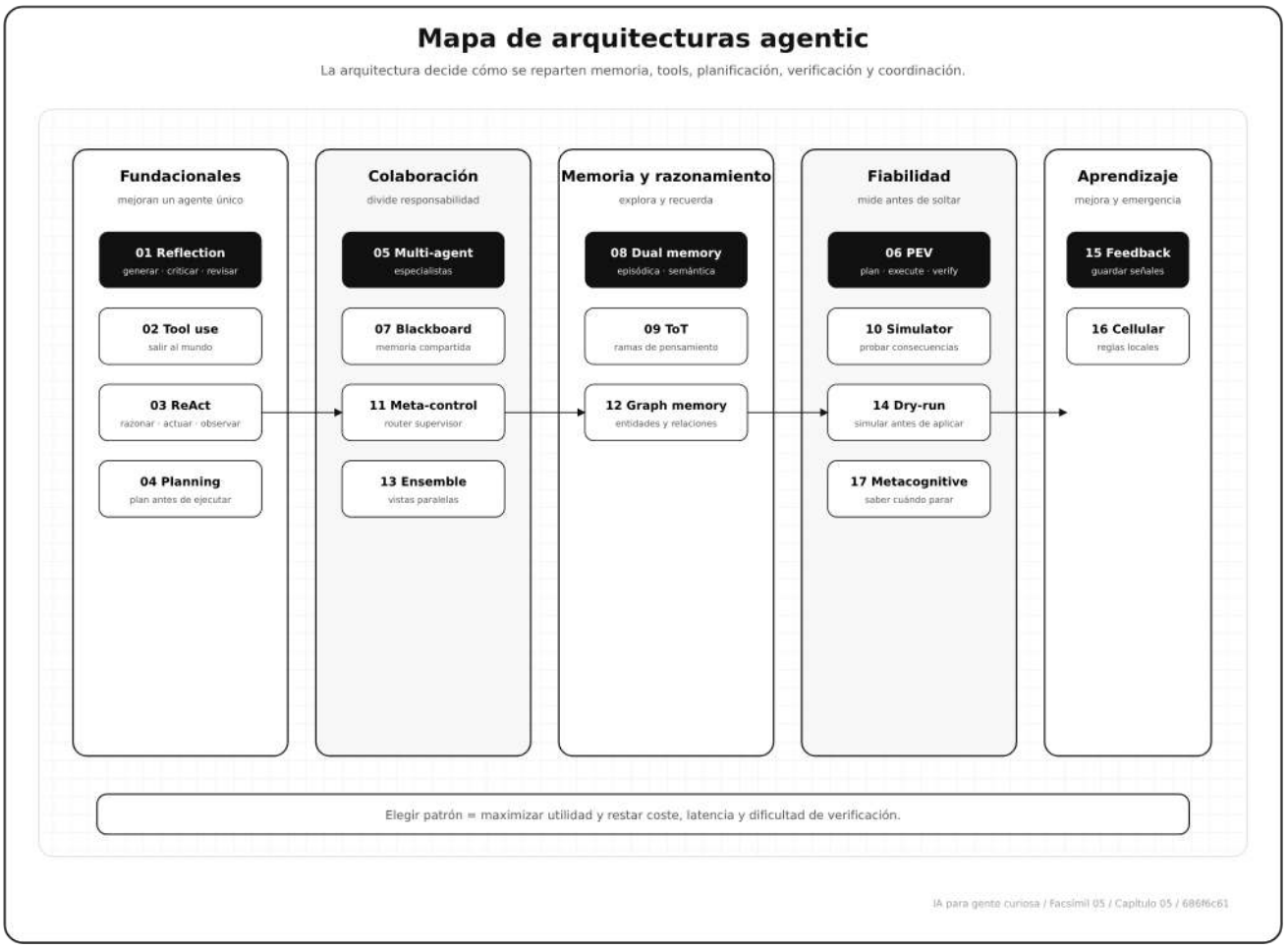
Este árbol se recorre varias veces. Una tarea real puede acabar en más de una hoja: por ejemplo, una revisión académica puede necesitar multiagente para separar roles, ReAct para consultar fuentes, PEV para comprobar resultados y dry-run para enseñar cambios antes de aplicarlos. La pregunta no es “qué nombre queda bonito”, sino **qué pieza cubre una necesidad que de verdad existe**.



La parte final del árbol es deliberada: aunque una rama ya te haya recomendado una arquitectura, todavía pregunta por verificación y traza. En sistemas con agentes no basta con elegir el patrón; hay que decidir cómo sabrás que funcionó.

SI EL ÁRBOL LLEGA A...	LÉELO ASÍ
Reflection	La tarea cabe en una salida, pero necesitas revisión explícita.
Tool use + PEV	Hay una consulta externa y debes validar el resultado.
ReAct + Tool use	El siguiente paso depende de lo que observes.
Planning + Tool use	Sabes los pasos antes de ejecutar, pero necesitas datos externos.
Tree of Thoughts	Hay varias rutas y puedes evaluar estados intermedios.
Simulator + Dry-run	Antes de actuar, puedes probar consecuencias.
Blackboard + Multi-agent	Varios especialistas necesitan escribir sobre la misma evidencia.
Meta-controller	No sabes de antemano qué especialista conviene.
Ensemble	Quieres comparar varias salidas antes de decidir.
Episodic + Semantic	Necesitas recordar experiencias y conceptos estables.
Graph memory	Las relaciones entre entidades son parte del problema.
Cellular automata	El comportamiento global sale de muchas reglas locales.
Feedback loop	La tarea se repite y quieres conservar señales de mejora.
Metacognitive	La calidad incluye saber cuándo responder, usar tool, pedir revisión o parar.

Mapa visual de arquitecturas agentic



La figura no coloca los patrones en una escalera moral. No hay una arquitectura “mejor” en abstracto. Hay familias que resuelven problemas distintos: mejorar una respuesta, usar herramientas, coordinar especialistas, recordar, verificar, simular o aprender de señales.

Arquitecturas agentic: las 25 del catálogo

El apartado se llama así a propósito: no estamos enumerando “técnicas sueltas”, sino **arquitecturas agentic**. Cada una toma las piezas del capítulo 02 y decide cómo se conectan. En todas hay que preguntar lo mismo: qué estado guarda, qué acciones permite, qué observaciones acepta, qué política decide y qué criterio de parada evita que el sistema siga por inercia.

El catálogo de Khan ha ido creciendo hasta reunir 35 arquitecturas, presentadas como un recorrido progresivo: patrones fundacionales, colaboración multiagente, memoria, razonamiento, recuperación, fiabilidad y aprendizaje.[3] Aquí recogemos 25, las que se han

consolidado y tienen respaldo en la literatura, y las explicamos con más lente de ingeniería. Las primeras diecisiete forman el núcleo; las ocho siguientes son patrones de razonamiento, multiagente, seguridad, memoria y acción que vale la pena conocer aparte.

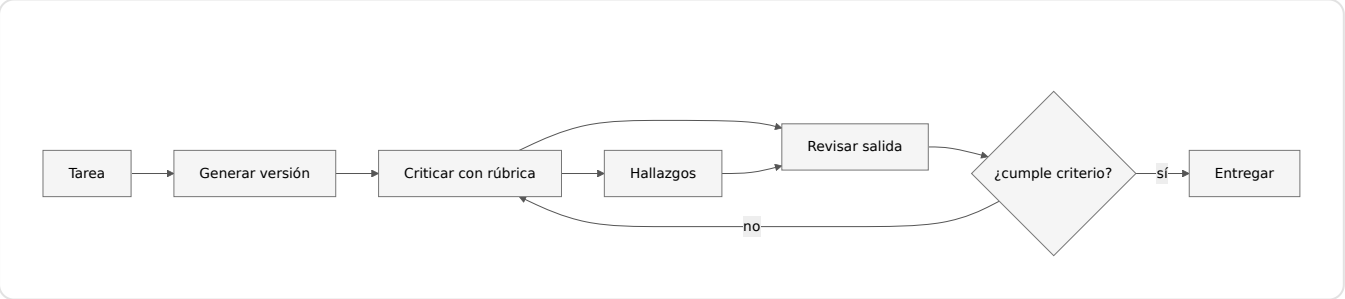
Cada arquitectura incluye un Mermaid propio. No son adornos: son una forma rápida de ver qué entra, qué estado cambia, dónde se decide, qué se comprueba y cuándo termina el flujo.

01. Arquitectura Reflection

Reflection convierte una salida en un pequeño ciclo editorial: generar, criticar, revisar y volver a evaluar. No añade conocimiento nuevo por sí misma; añade una segunda mirada estructurada sobre lo ya producido. En código o escritura técnica, esto permite separar "crear" de "revisar". El notebook `01_reflection.ipynb` lo presenta como el paso de un generador de una sola pasada a un agente que produce, evalúa y mejora antes de entregar.

El estado mínimo contiene la primera versión, una rúbrica, los hallazgos de la crítica y la versión corregida. La política decide si basta una revisión o si hace falta otra iteración. La métrica no debería ser "suena mejor", sino defectos corregidos, tests que pasan, citas arregladas o errores eliminados.

Reflexion formaliza una idea cercana: agentes que usan feedback verbal para mejorar decisiones futuras.[4] Self-Refine estudia el ciclo generar-feedback-refinar sin entrenamiento adicional.[5] La trampa: si la crítica es vaga, solo produces una segunda respuesta igual de frágil pero más cara.

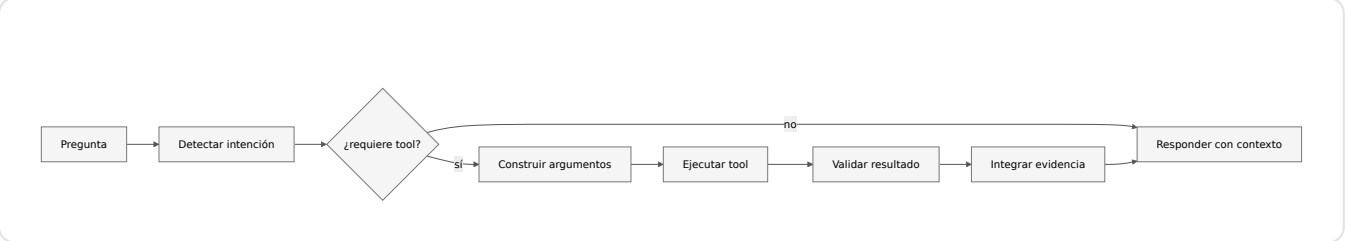


02. Arquitectura Tool use

Tool use aparece cuando el modelo no debe inventar una respuesta desde memoria paramétrica, sino pedir ayuda a una función externa: buscar, calcular, consultar una base de datos, abrir una URL o validar un JSON. Aquí la arquitectura separa lenguaje natural de ejecución. El notebook `02_tool_use.ipynb` lo plantea como el puente entre el razonamiento del LLM y datos vivos: APIs, funciones y fuentes que no caben en los pesos del modelo.

El estado mínimo contiene la intención de la tarea, el catálogo de herramientas, el esquema de entrada de cada tool, permisos, timeouts y observaciones. La salida importante no es el texto final, sino el par `tool_call -> tool_result`: ahí se ve si el agente usó una capacidad real o solo la mencionó.

Toolformer mostró que los modelos pueden aprender cuándo llamar herramientas, pero en sistemas de ingeniería se declara un contrato explícito.[6] Las APIs modernas de agentes siguen esa línea: tools con descripción, esquema y resultado verificable.[7] Cuidado: una tool sin validación es solo una nueva forma de meter ruido.

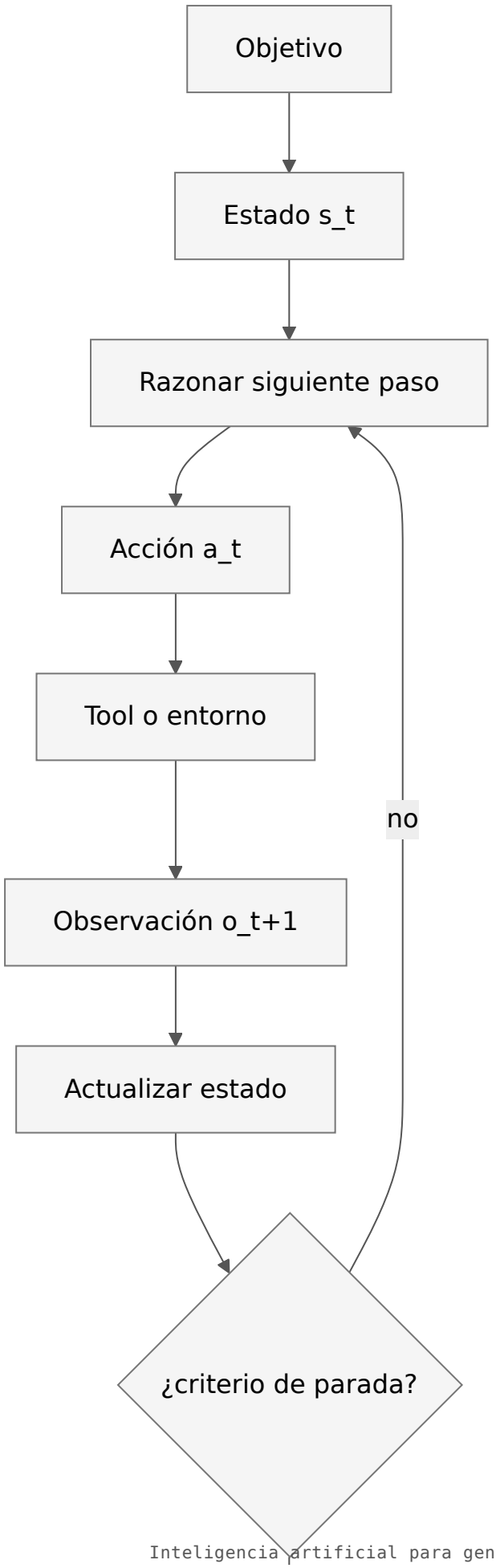


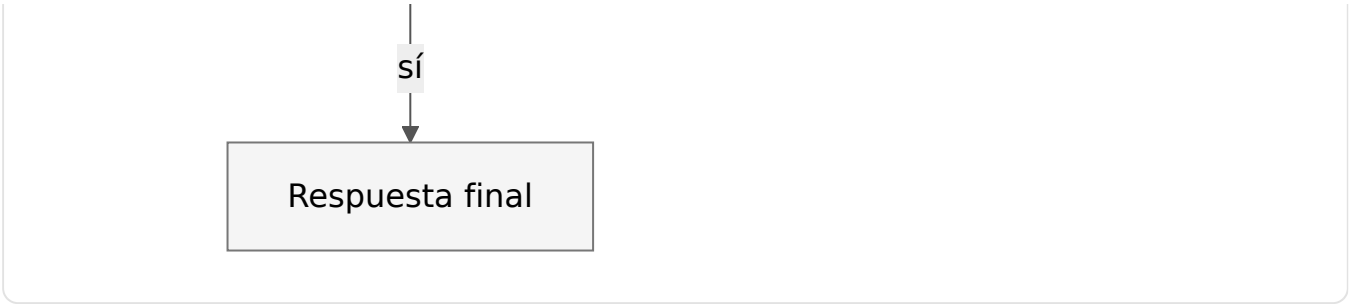
03. Arquitectura ReAct

ReAct combina razonamiento y acción en un bucle: pensar el siguiente paso, ejecutar una tool, observar, actualizar y volver a decidir. Es la arquitectura que más claramente conecta con s_t, a_t, o_{t+1}, T. El notebook `03_ReAct.ipynb` compara un agente de tool use de una sola llamada con un agente capaz de iterar `think -> act -> observe`.

Encaja cuando el siguiente paso depende de lo que se observa: investigar una librería, revisar una web, depurar un error, comparar fuentes. No encaja cuando el flujo está cerrado y siempre se ejecuta igual; ahí un workflow simple suele ser más barato y auditable.

El paper de ReAct mostró que intercalar razonamiento y acción ayuda en tareas que requieren información externa y decisiones sucesivas.[8] El cuidado principal es la parada: si una observación no cambia el estado, repetir otra tool parecida rara vez mejora el sistema.


Objetivo
Estado s_t
Razonar siguiente paso
Acción a_t
Tool o entorno
Observación o_t+1
Actualizar estado
¿criterio de parada?
no

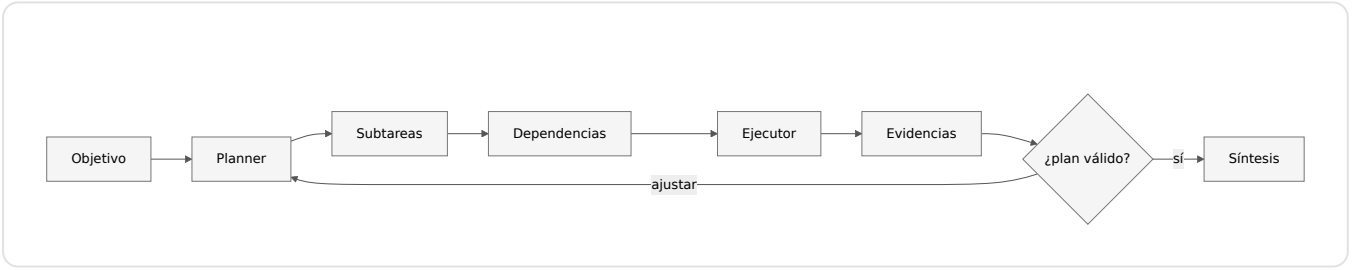


04. Arquitectura Planning

Planning separa planificar de ejecutar. El agente crea una descomposición de la tarea antes de tocar herramientas o producir la respuesta final. Esto da estructura, permite revisar el plan y hace visible si la solución omitió pasos. El notebook `04_planning.ipynb` compara esta estrategia con ReAct: en vez de reaccionar paso a paso, crea una secuencia de subtareas antes de ejecutar.

El estado mínimo contiene objetivo, lista de subtareas, dependencias, estado de cada paso y evidencias asociadas. Si el plan no se actualiza al recibir observaciones, no es una arquitectura viva: es una lista decorativa.

La planificación automática tiene una tradición larga en IA clásica, desde lenguajes como PDDL hasta enfoques de planificación heurística.[9][10] En agentes con LLM, el plan es útil si se puede verificar y corregir, no si solo parece ordenado.

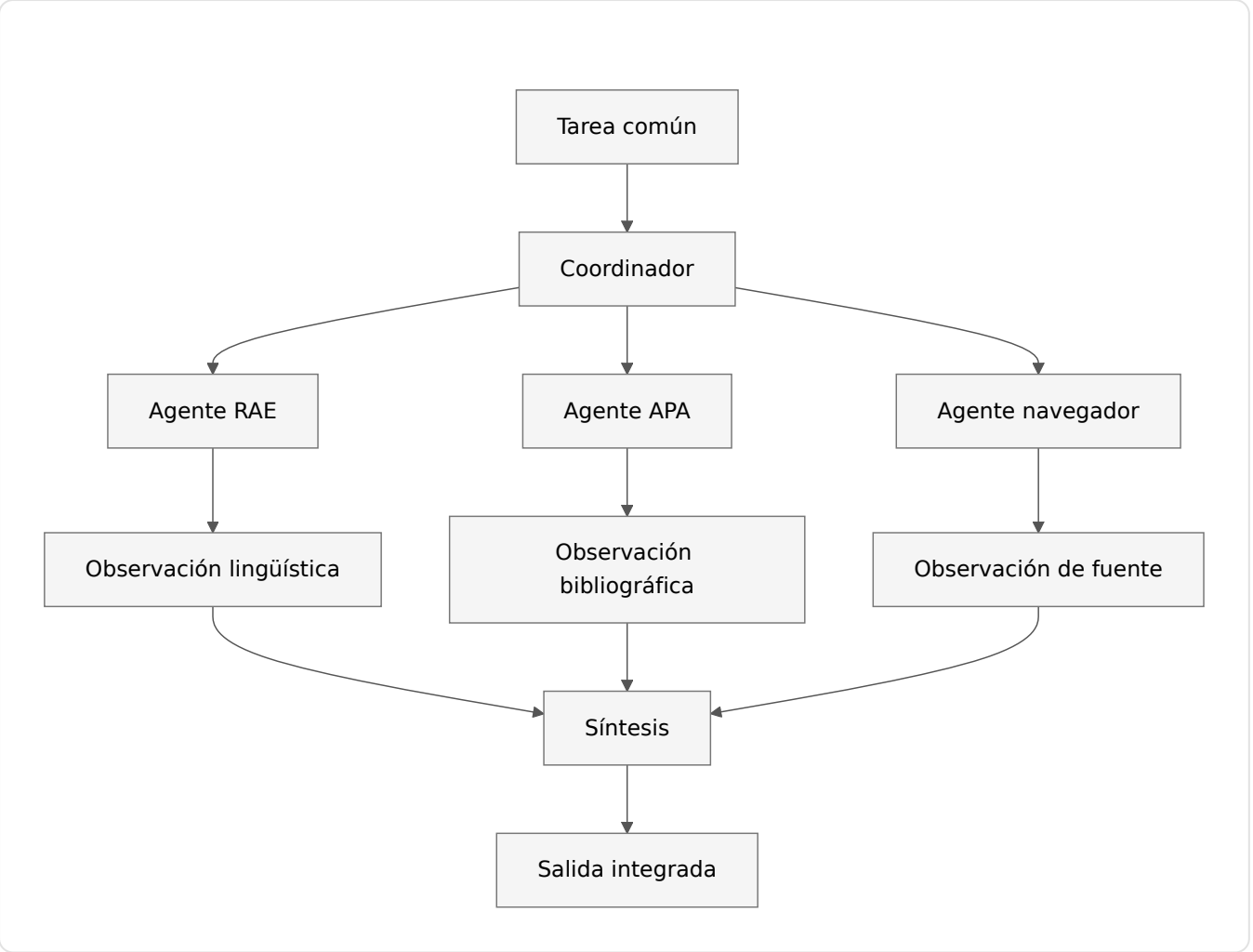


05. Arquitectura Multi-Agent Systems

Multi-agent divide el trabajo entre especialistas: un agente investiga, otro escribe, otro verifica, otro sintetiza. Su valor no está en tener muchos nombres, sino en separar responsabilidades que tienen criterios de calidad distintos. El notebook `05_multi_agent.ipynb` usa un ejemplo de análisis de mercado con analistas especializados y un agente gestor que sintetiza.

El estado mínimo incluye roles, entradas, salidas esperadas, permisos de cada agente y un mecanismo de integración. Si todos los agentes pueden hacer lo mismo, no hay arquitectura; hay redundancia cara.

El ejemplo editorial del capítulo 02 encaja aquí: un agente RAE revisa lengua, otro APA revisa referencias y otro navegador comprueba fuentes. Cada uno produce observaciones distintas. La coordinación puede ser un workflow fijo o un agente coordinador, según si el orden depende de resultados intermedios.

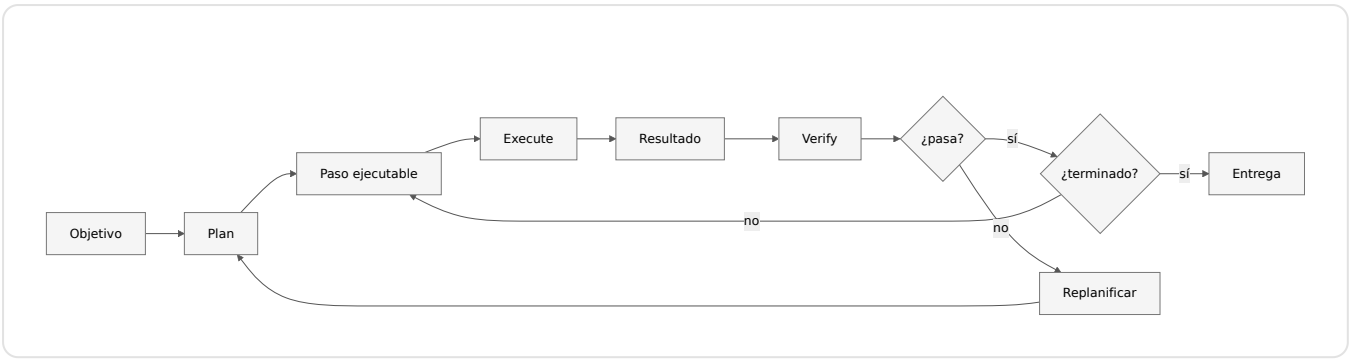


06. Arquitectura PEV: Plan, Execute, Verify

PEV separa tres momentos: planificar, ejecutar y verificar. Parece obvio, pero cambia mucho el diseño: la verificación deja de ser una sensación final y se convierte en una fase con datos propios. El notebook `06_PEV.ipynb` lo muestra como un planificador-ejecutor al que se añade un verificador para detectar fallos de herramientas y activar recuperación.

El estado mínimo contiene plan, resultado de ejecución, verificador, errores detectados y acción correctiva. Sirve cuando las herramientas fallan, las fuentes pueden no respaldar una afirmación o el código puede pasar por estados intermedios incorrectos.

Anthropic recomienda desarrollar tests para aplicaciones con LLM, precisamente porque las respuestas deben medirse en escenarios concretos y no solo leerse a ojo.[11] La trampa de PEV es poner otro LLM como “verificador” sin rúbrica, tests ni evidencia externa.

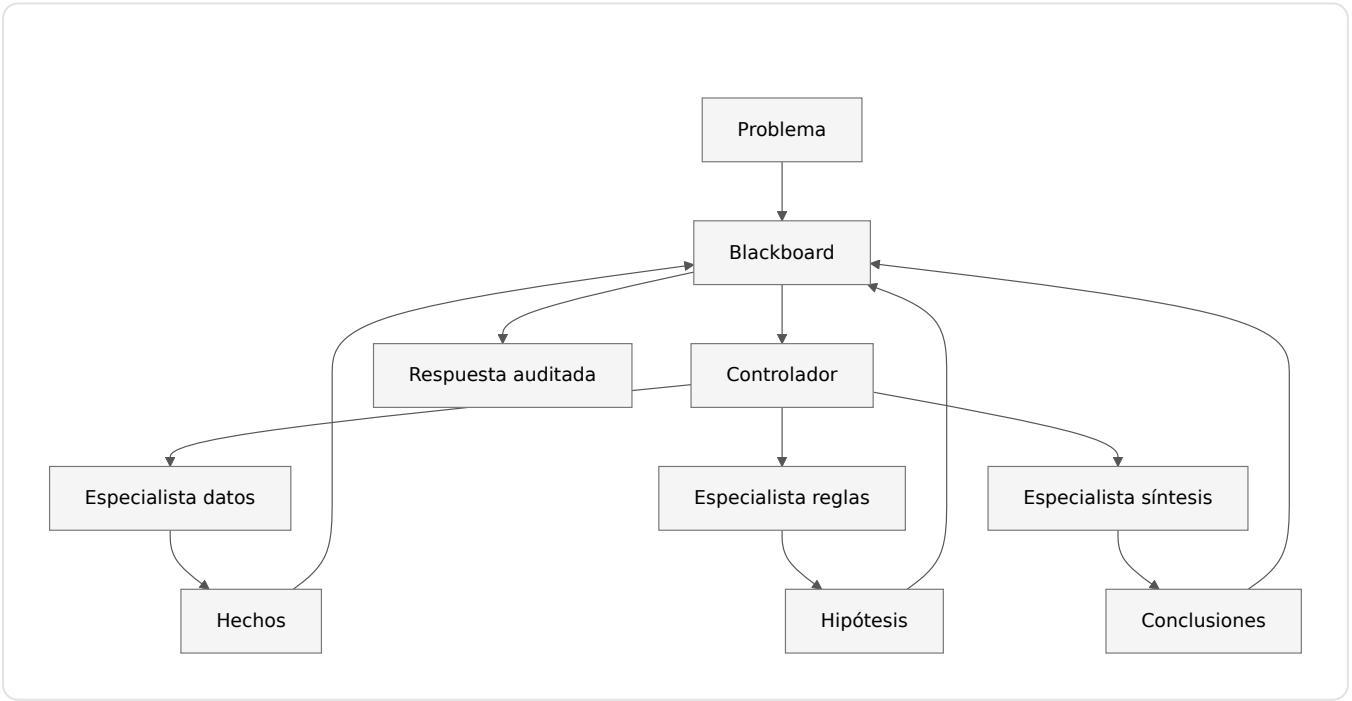


07. Arquitectura Blackboard

Blackboard usa una memoria compartida donde varios especialistas escriben hallazgos parciales. El controlador observa el estado del tablero y decide qué especialista debe actuar después. El notebook `07_blackboard.ipynb` lo contrapone a un multiagente secuencial: en vez de pasar siempre por A, B y C, el controlador activa al especialista que el tablero necesita en ese momento.

Esta idea viene de sistemas clásicos de IA: el modelo blackboard se usaba para resolver problemas complejos mediante fuentes de conocimiento especializadas que colaboran sobre una estructura común.[1][2] En agentes modernos, el blackboard puede ser una tabla, un documento, una base vectorial, un grafo o un estado de LangGraph.

El estado mínimo necesita autor, fuente, timestamp, confianza, versión y relación con otras evidencias. Si el tablero no distingue “hecho”, “hipótesis” y “conclusión”, se vuelve una pared llena de notas imposibles de auditar.

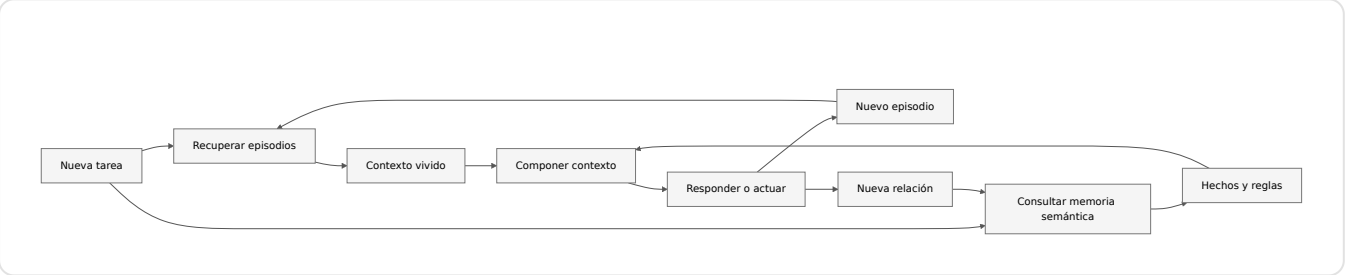


08. Arquitectura Episodic + Semantic Memory

La memoria episódica guarda experiencias: qué pasó, cuándo, con quién, en qué tarea. La memoria semántica guarda conocimiento más estable: conceptos, preferencias, hechos, reglas, entidades. Combinarlas evita dos extremos: olvidar todo o recordar texto bruto sin estructura. El notebook `08_episodic_with_semantic.ipynb` usa una base vectorial para episodios y Neo4j para hechos y relaciones.

Un asistente de proyecto puede guardar episodios como “la última vez el build falló por KaTeX” y conocimiento semántico como “este facsímil exige Mermaid en cada capítulo”. La recuperación debe explicar por qué trae una memoria concreta.

Generative Agents popularizó una arquitectura con memoria, reflexión y planificación para simular comportamiento persistente de agentes en un entorno.[13] La regla práctica: toda memoria debe tener fuente, fecha, caducidad y mecanismo de corrección.

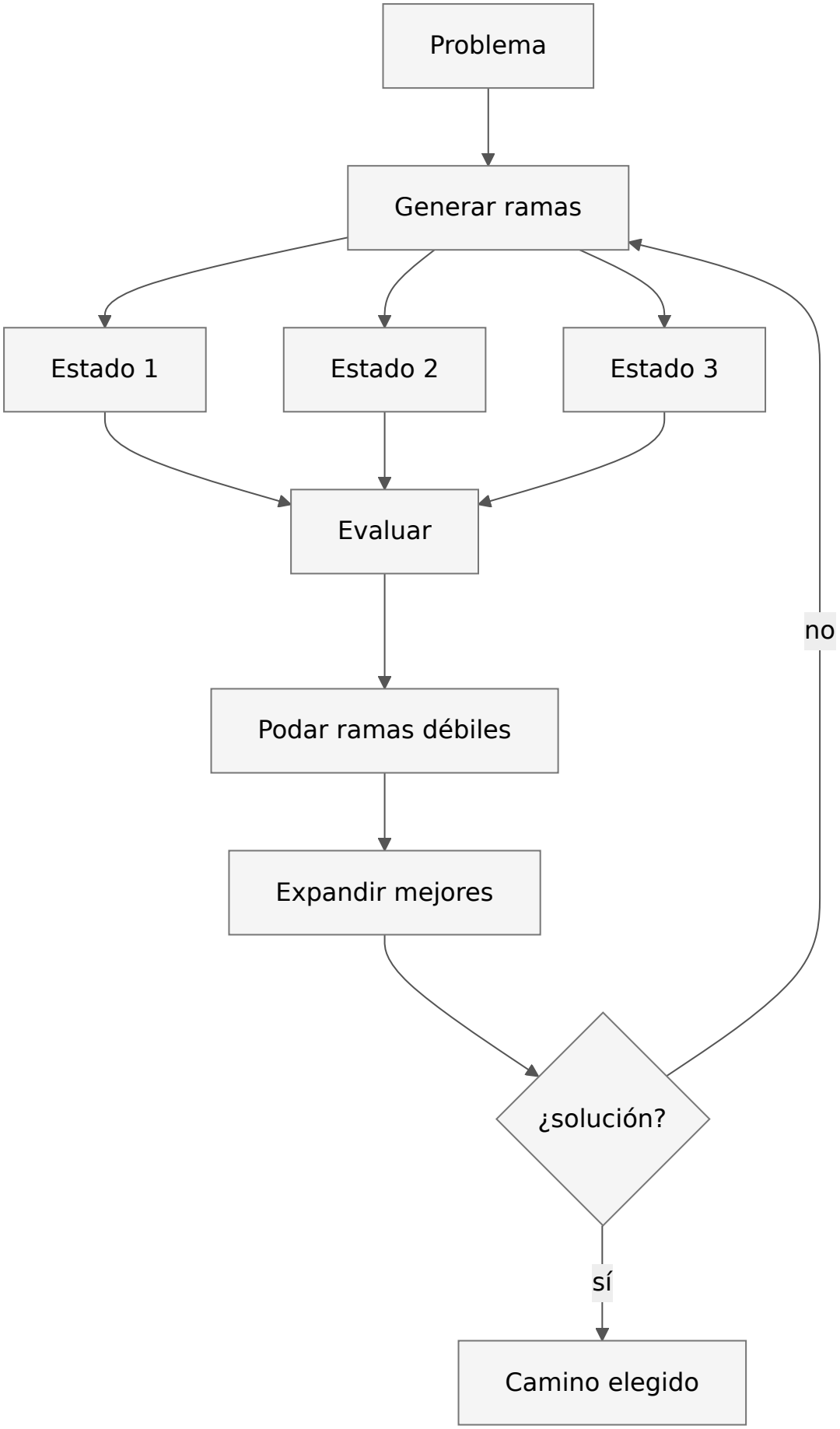


09. Arquitectura Tree of Thoughts

Tree of Thoughts no sigue una sola cadena de razonamiento. Construye varias ramas, evalúa estados intermedios y poda caminos poco prometedores. Es búsqueda aplicada al razonamiento con LLM. El notebook `09_tree_of_thoughts.ipynb` usa el problema del lobo, la cabra y la col para mostrar por qué una trayectoria lineal puede quedar atrapada y una búsqueda por ramas puede recuperar el camino.

El estado mínimo contiene nodos, puntuación de cada rama, profundidad, criterio de expansión y criterio de poda. Encaja en puzzles lógicos, planificación con restricciones, demostraciones o decisiones donde una mala primera intuición arrastra toda la respuesta.

El paper de Tree of Thoughts propone deliberar mediante búsqueda sobre unidades de pensamiento, no solo generar una secuencia lineal.[14] La factura aparece rápido: ancho de búsqueda por profundidad por coste de evaluación. Sin límites, es elegante y carísimo.

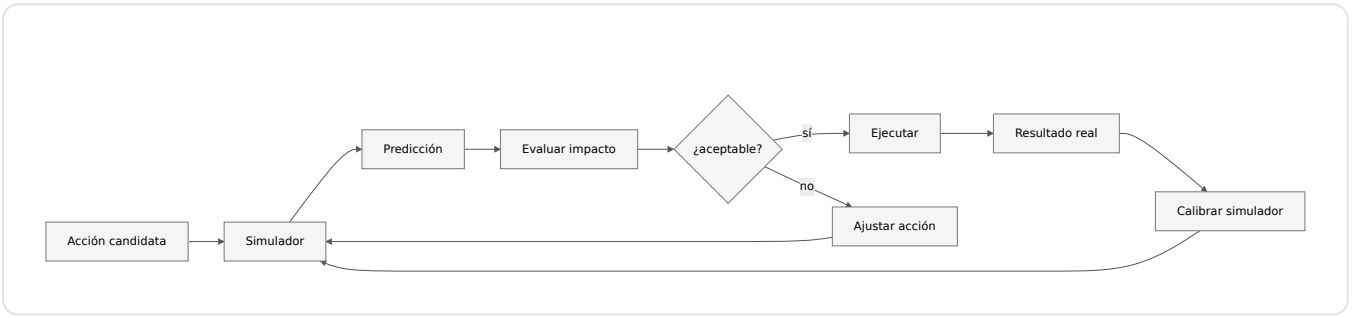

Problema
Generar ramas
Estado 1
Estado 2
Estado 3
Evaluar
Podar ramas débiles
Expandir mejores
¿solución?
no
sí
Camino elegido

10. Arquitectura Mental Loop o Simulator

Mental Loop introduce un simulador interno. Antes de actuar, el agente prueba una consecuencia en un modelo del entorno: “si hago esto, ¿qué pasaría?”. Puede ser un simulador real, una función determinista, una evaluación de impacto o una réplica controlada. El notebook `10_mental_loop.ipynb` usa un agente de trading que ensaya una estrategia en una copia del mercado antes de actuar.

El estado mínimo contiene acción propuesta, simulación, predicción, incertidumbre y decisión. Sirve cuando actuar tiene coste: cambiar una base de datos, modificar un archivo, ejecutar una orden, mover inventario o recomendar una decisión profesional.

La clave no es “imaginar” consecuencias, sino comparar predicción y resultado real en casos pequeños. Si el simulador no se calibra, tranquiliza sin proteger.

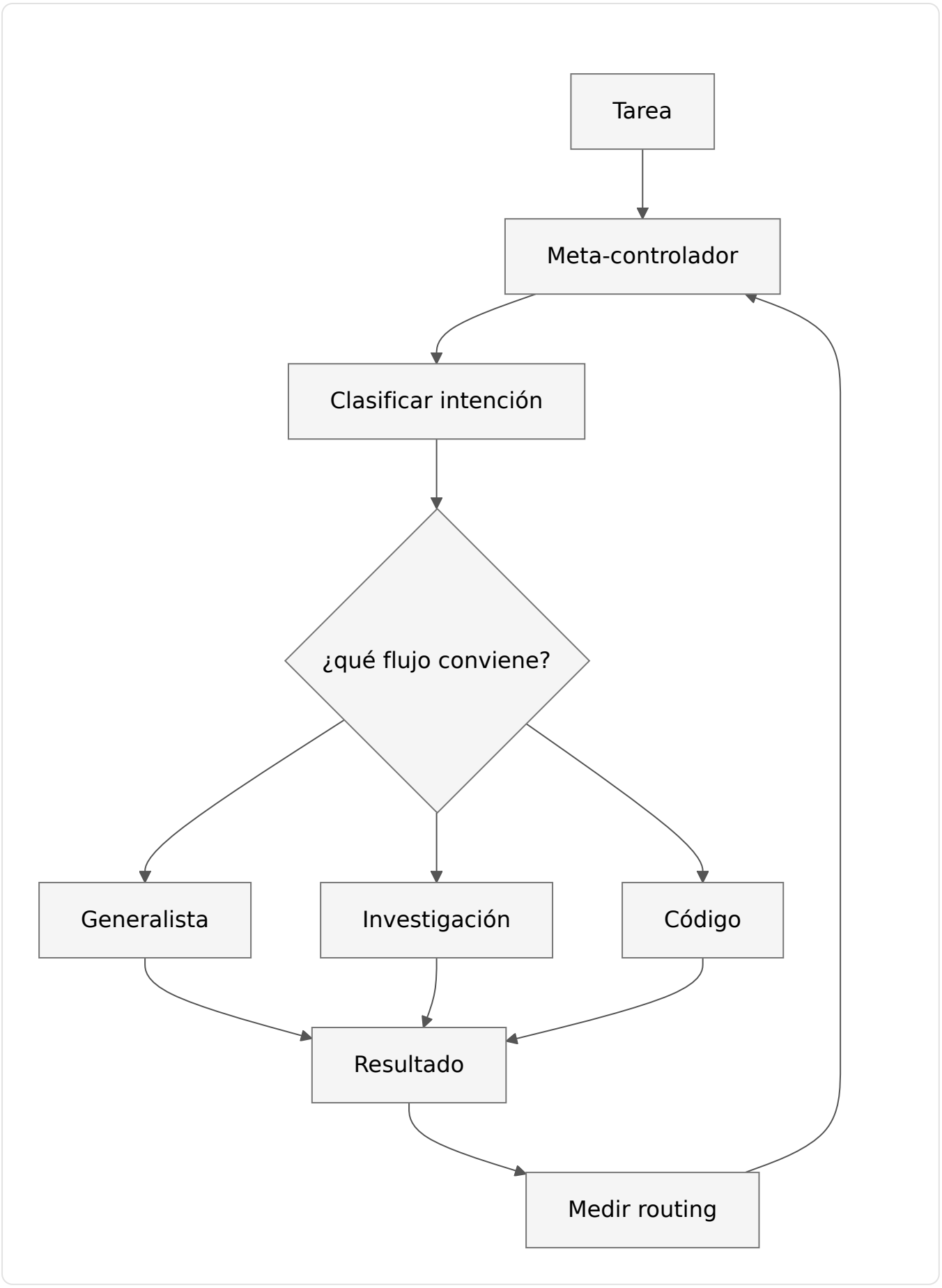


11. Arquitectura Meta-Controller

Meta-controller es un router con criterio. Recibe una tarea, estima qué especialista o flujo conviene y delega. Es útil cuando hay muchas capacidades disponibles y elegir mal la primera acción ya encarece todo. El notebook `11_meta_controller.ipynb` usa tres especialistas: generalista, investigación y código; el controlador decide quién debe responder.

El estado mínimo contiene intención clasificada, capacidades disponibles, coste esperado, restricciones y resultado del especialista. El meta-controlador debería aprender de errores de enrutamiento: cuántas veces manda una consulta técnica al agente equivocado, cuánto tarda y qué calidad final obtiene.

OpenAI Agents SDK ofrece handoffs entre agentes, una forma práctica de representar esta delegación controlada.[15] La trampa es convertir el router en un “jefe” que opina de todo. Su trabajo principal es enrutar, medir y corregir routing.

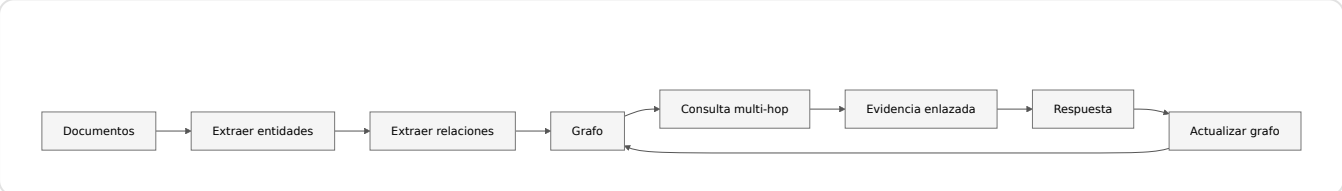

Tarea
Meta-controlador
Clasificar intención
¿qué flujo conviene?
Generalista
Investigación
Código
Resultado
Medir routing

12. Arquitectura Graph o World-Model Memory

Graph memory guarda entidades y relaciones: autor-publicó-paper, capítulo-cita-fuente, herramienta-produce-observación, concepto-depende-de-concepto. Esto permite preguntas multi-hop que una lista de chunks recuperados no resuelve bien. El notebook `12_graph.ipynb` construye un agente de inteligencia corporativa que extrae compañías, personas, productos y relaciones hacia un grafo consultable.

El estado mínimo contiene nodos, aristas, tipos, procedencia y reglas de actualización. Encaja cuando importa la estructura: ontologías, dependencias de software, investigación documental, mapas conceptuales o memoria de proyecto.

Los knowledge graphs se estudian como estructuras para representar entidades y relaciones consultables.[16] En agentes, el grafo no reemplaza RAG; lo complementa cuando las relaciones importan tanto como los documentos.

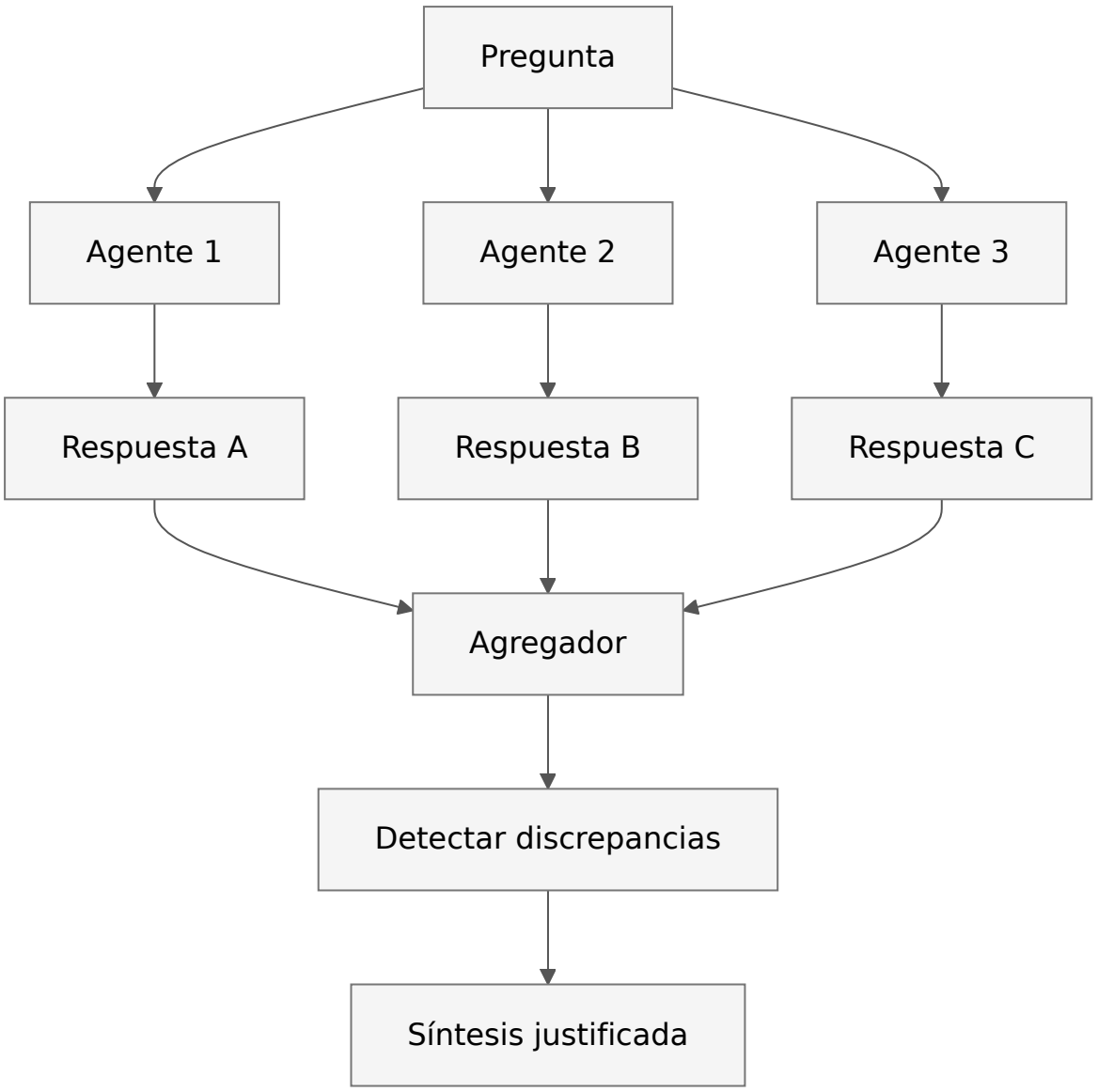


13. Arquitectura Ensemble

Ensemble ejecuta varias perspectivas y agrega. Puede significar varios modelos, varios prompts, varios agentes especialistas o varias trayectorias de razonamiento. Es útil cuando el error de una sola trayectoria sería demasiado frágil. El notebook `13_ensemble.ipynb` usa un comité de inversión con perfiles distintos y un agregador que sintetiza consenso y discrepancias.

El estado mínimo contiene respuestas candidatas, criterios de comparación, discrepancias, agregación y decisión final. Agregar no es hacer media de frases ni votar por mayoría sin mirar evidencia.

Self-consistency mostró que muestrear varios razonamientos y agregar respuestas puede mejorar razonamiento sobre chain-of-thought.[17] En un agente editorial, por ejemplo, un ensemble puede comparar tres verificaciones de una cita, pero la síntesis debe explicar por qué acepta una.

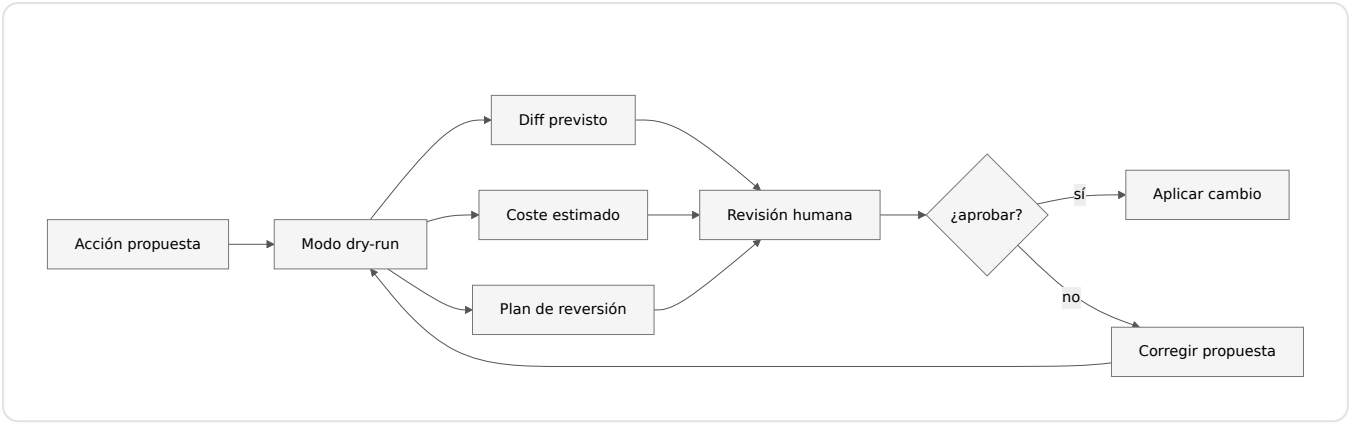


14. Arquitectura Dry-Run Harness

Dry-run harness exige simular antes de aplicar. No basta con que el agente diga “haría esto”; debe mostrar diff, efecto esperado, coste, permisos y plan de reversión antes de tocar el entorno real. El notebook `14_dry_run.ipynb` usa un agente de redes sociales corporativas que primero ejecuta en modo `dry_run=True`, muestra la traza y solo después permite publicar.

El estado mínimo contiene acción propuesta, simulación, diff, comprobaciones, aprobación y resultado tras aplicar. Encaja muy bien con herramientas de código, operaciones, migraciones y flujos editoriales de publicación.

La documentación de tracing del Agents SDK es relevante porque un dry-run sin traza no se puede auditar después.[18] En producción, el dry-run debe ser legible por una persona y comparable contra el resultado real.

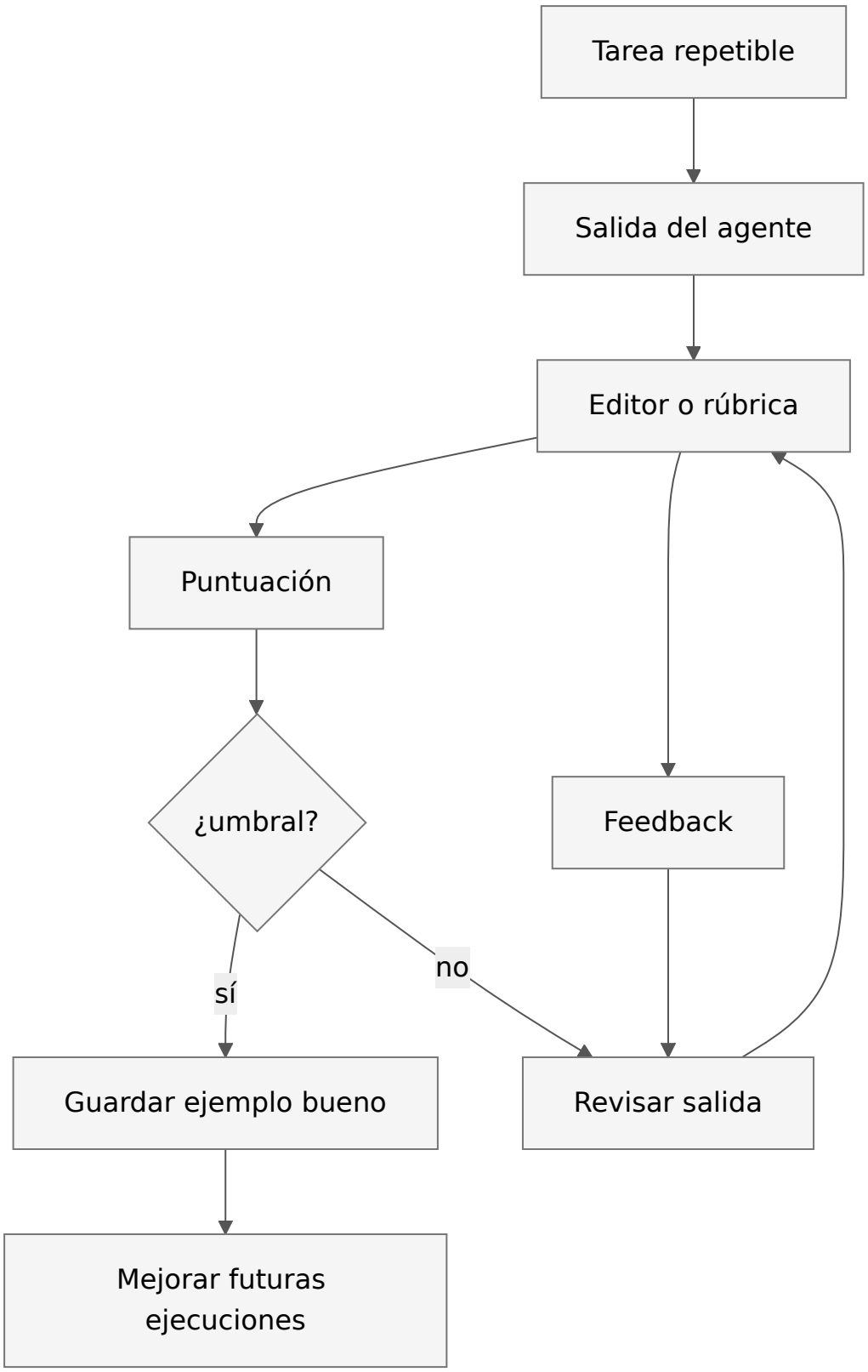


15. Arquitectura RLHF / Self-Improvement

En el catálogo aparece como un bucle de feedback: una salida se revisa, se corrige y las mejores señales se guardan para mejorar futuras ejecuciones. No hay que confundirlo con entrenar un modelo desde cero; muchas veces es curar ejemplos, rúbricas y preferencias de aplicación. El notebook `15_RLHF.ipynb` lo aproxima con un agente redactor y un editor que puntúa, da feedback y fuerza revisión hasta alcanzar un umbral.

El estado mínimo contiene salida, feedback, revisión, puntuación y memoria de ejemplos aceptados. Sirve cuando la tarea es repetitiva y medible: soporte, revisión editorial, generación de informes, clasificación de tickets.

RLHF se popularizó en modelos instruidos como forma de aprender de preferencias humanas.[19] En una aplicación pequeña, el equivalente práctico suele ser más humilde: guardar buenas correcciones, no premiar salidas dudosas y medir si el sistema mejora en un conjunto fijo.

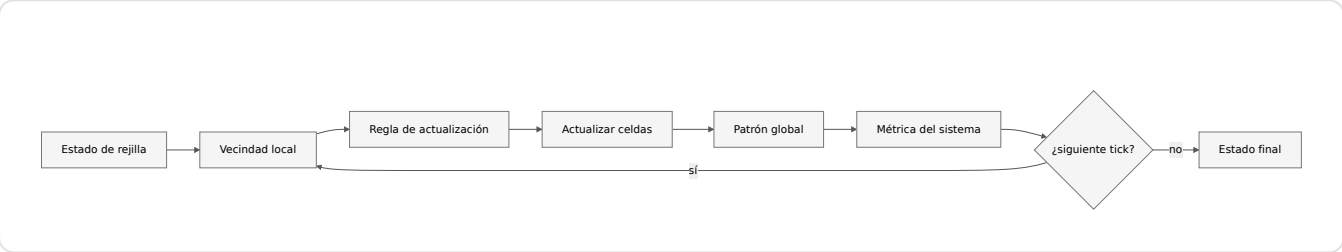

Tarea repetible
Salida del agente
Editor o rúbrica
Puntuación
¿umbral?
Feedback
sí
no
Guardar ejemplo bueno
Revisar salida
Mejorar futuras ejecuciones

16. Arquitectura Cellular Automata

Cellular Automata no es una arquitectura típica de chat. Modela muchos agentes simples con reglas locales. De esas reglas puede emerger comportamiento global: rutas, congestión, propagación, distribución de recursos. El notebook `16_cellular_automata.ipynb` usa una simulación de almacén donde las celdas de una rejilla propagan información para encontrar rutas.

El estado mínimo contiene una rejilla o grafo, celdas/agentes, vecindad, regla de actualización y métrica global. Encaja en simulación espacial, logística, planificación de rutas, ocupación de salas o fenómenos donde la interacción local importa.

Su lección para agentes LLM es conceptual: no siempre necesitas un agente muy inteligente. A veces necesitas muchos componentes simples, reglas claras y una buena visualización del estado global.

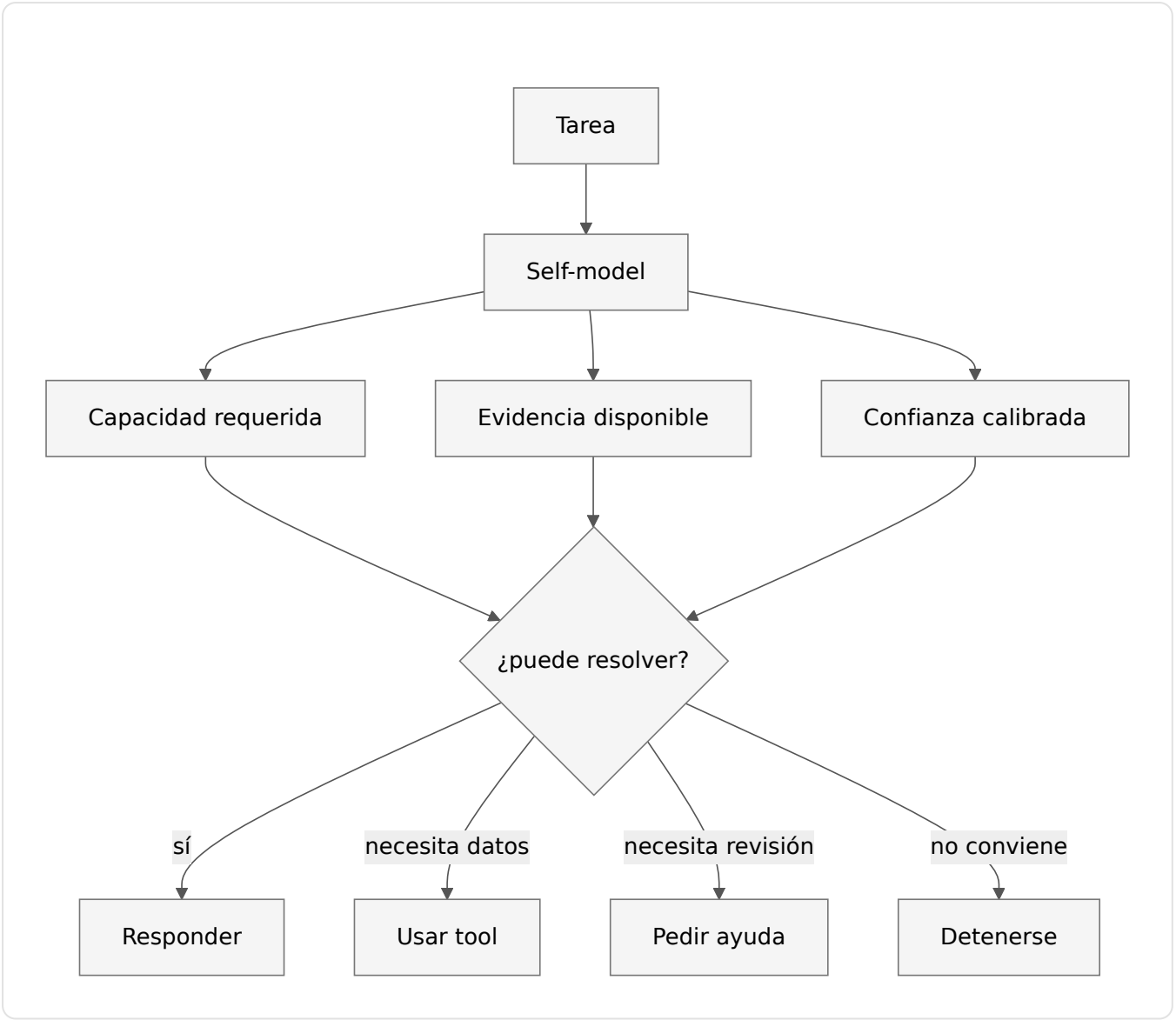


17. Arquitectura Reflexive Metacognitive

Reflexive Metacognitive añade un modelo de las propias capacidades del sistema: qué sabe hacer, qué no sabe, qué herramienta necesita, cuándo debe pedir revisión y cuándo debe parar. Bien diseñada, no es modestia verbal; es control operativo. El notebook `17_reflexive_metacognitive.ipynb` lo demuestra con un asistente de triaje médico-informativo que primero consulta su self-model antes de responder, usar una tool o escalar.

El estado mínimo contiene capacidad requerida, confianza calibrada, evidencia disponible, acciones permitidas y salida posible: responder, usar tool, pedir ayuda o detenerse. Encaja en asesoramiento técnico, triaje profesional, revisión de fuentes o decisiones donde reconocer límites es parte de la calidad.

La diferencia con Reflection es importante. Reflection revisa una salida. Metacognición decide si el sistema está en condiciones de actuar. Si solo añade frases tipo “podría estar equivocado” sin cambiar la política, no aporta arquitectura.



Hay una línea clara con lo visto antes. Chain-of-thought mostró que pedir razonamiento intermedio puede mejorar tareas que requieren varios pasos.[20] Self-consistency añadió una idea que conecta con ensemble: muestrear varios razonamientos y agregar la respuesta puede ser más robusto que confiar en una sola trayectoria.[21] ReAct formaliza la alternancia entre razonamiento y acción con observaciones de herramientas.[22] Toolformer mostró que el uso de herramientas puede aprenderse como parte del comportamiento del modelo, aunque en ingeniería solemos envolverlo con schemas y validadores.[23] Tree of Thoughts empuja la idea de explorar varios caminos de razonamiento antes de comprometerse con una respuesta.[24]

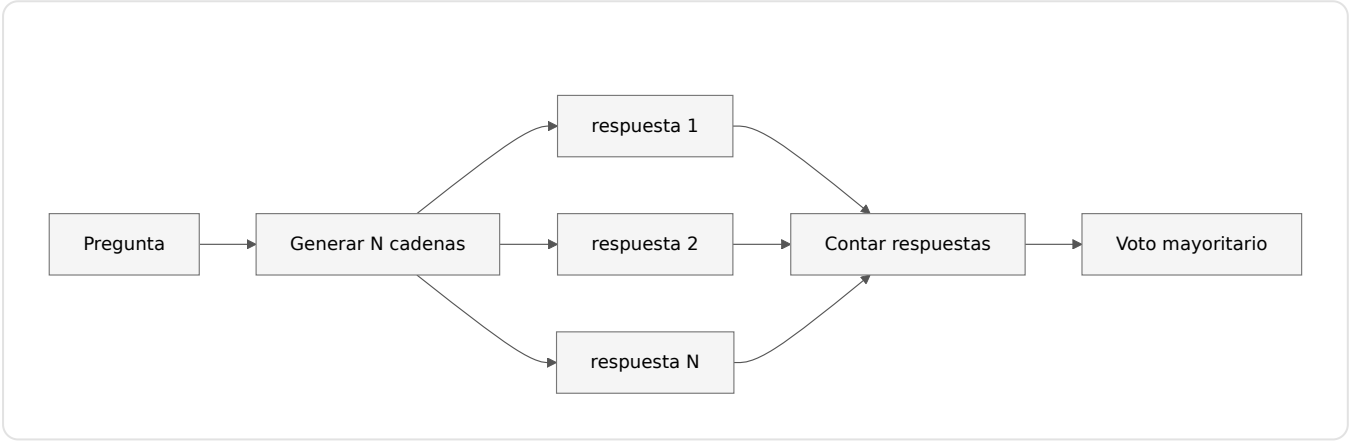
18. Arquitectura Self-Consistency

Self-Consistency parte de una idea simple: en vez de pedir una sola cadena de razonamiento, pide muchas y quédate con la respuesta a la que llegan la mayoría. Si un problema admite varios caminos correctos, esos caminos tienden a coincidir en la respuesta final, mientras que

los errores se dispersan; el voto mayoritario filtra el ruido. Es la versión de razonamiento de algo que ya conoces de la estadística: promediar muestras independientes reduce la varianza.

El estado mínimo guarda las N cadenas generadas y el recuento de respuestas finales. La política no es del LLM, sino de Python: extraer la respuesta de cada cadena y contar. La métrica relevante es si la respuesta mayoritaria mejora frente a una sola pasada, y a qué coste, porque generar N caminos multiplica por N el gasto.

El método se introdujo como una mejora directa del *chain of thought*, y funciona mejor cuanto más se beneficia la tarea de explorar varias rutas, como en problemas de aritmética o de sentido común.[25] La trampa: si la tarea tiene una sola vía o el modelo se equivoca de forma sistemática, votar más caminos solo confirma el mismo error más caro.

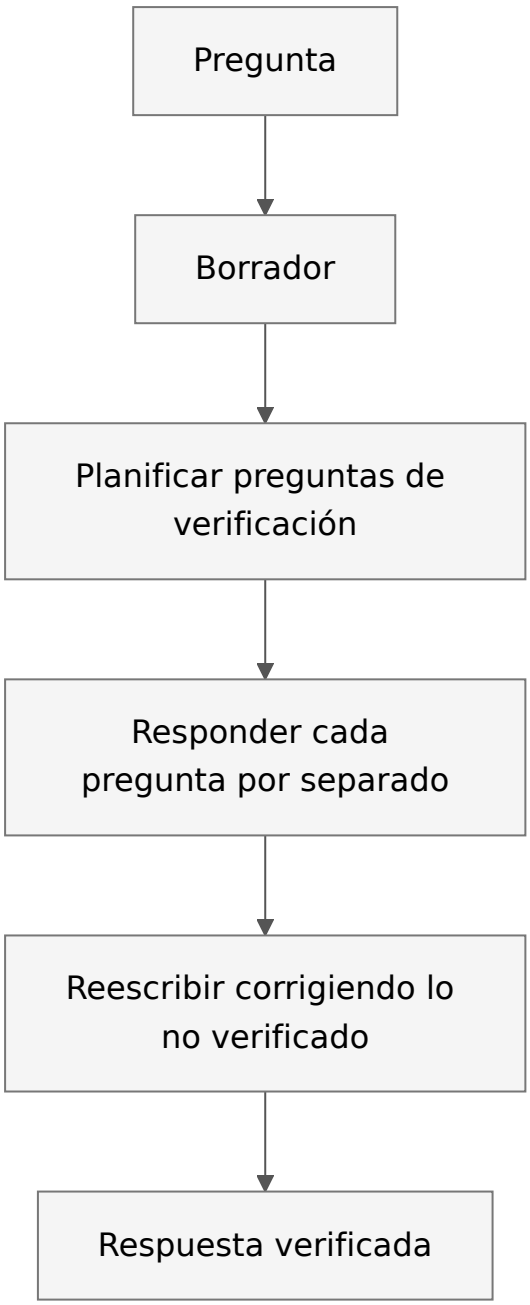


19. Arquitectura Chain-of-Verification

Chain-of-Verification (CoVe) ataca un fallo concreto: las afirmaciones plausibles pero falsas. En vez de confiar en la primera respuesta, el agente redacta un borrador, genera preguntas de verificación sobre cada afirmación que contiene, las responde **de forma independiente** (sin ver el borrador, para no contagiarse de su sesgo) y reescribe la respuesta final corrigiendo lo que no se sostiene. Es una forma disciplinada de que el modelo se revise a sí mismo por partes.

El estado mínimo contiene el borrador, la lista de preguntas de verificación, sus respuestas aisladas y la versión final. La política decide qué afirmaciones merecen verificación; la métrica es la reducción de afirmaciones falsas, no que el texto suene más seguro.

El paper mostró que esta separación entre afirmar y verificar reduce las alucinaciones en listas, preguntas de respuesta cerrada y texto largo.[26] El cuidado clave es la independencia: si las preguntas de verificación se responden mirando el borrador, el agente se limita a ratificar sus propios errores.

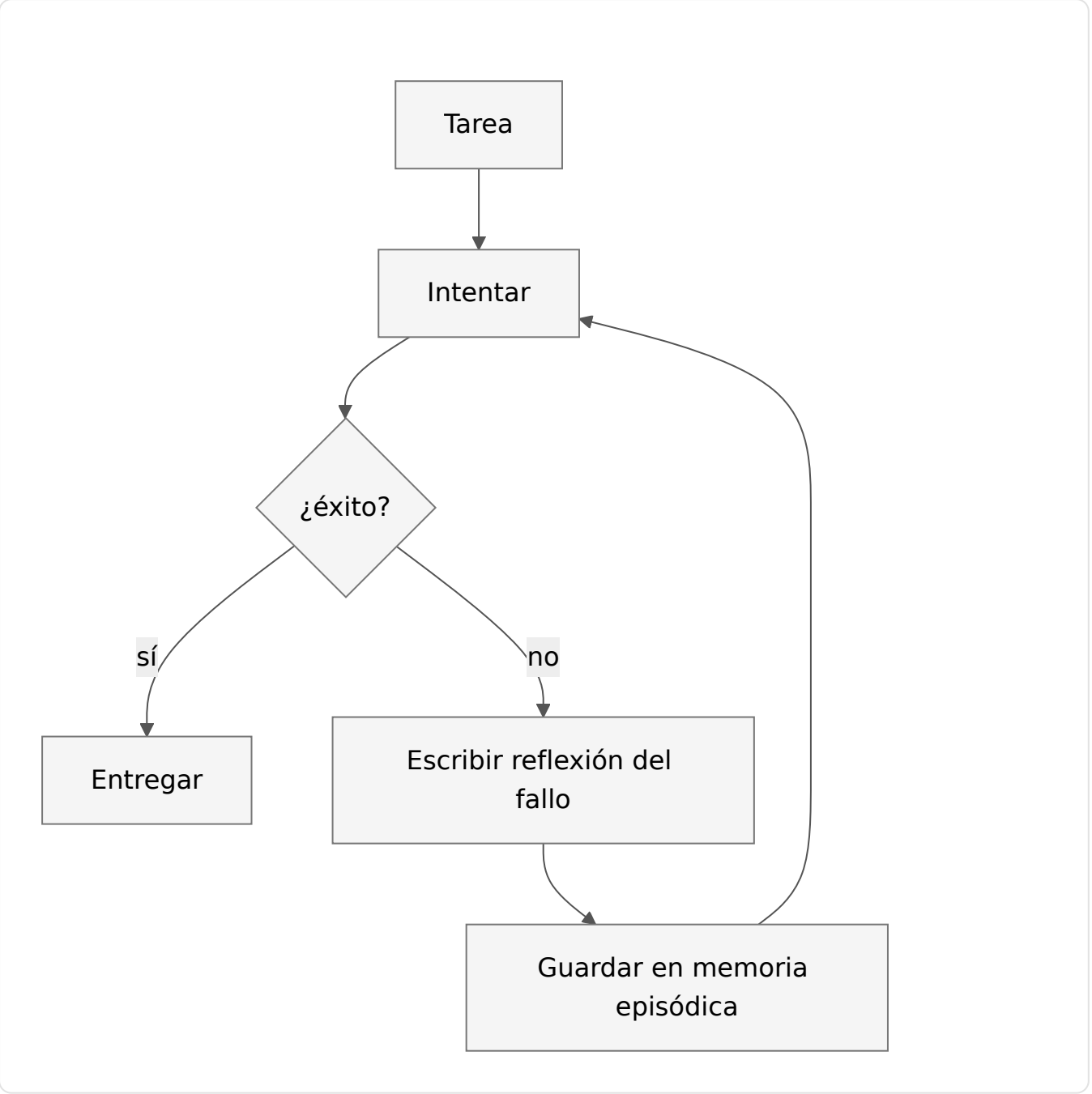


20. Arquitectura Reflexion

Reflexion lleva la idea de Reflection un paso más allá: en vez de criticar y revisar dentro de una sola tarea, el agente convierte cada fracaso en una **nota verbal** que guarda en memoria y consulta en el siguiente intento. Tras fallar, escribe en lenguaje natural qué salió mal y qué hará distinto, y ese texto pasa a formar parte del contexto de la próxima vuelta. Es aprendizaje por refuerzo, pero la señal no son gradientes, sino frases.

El estado mínimo añade, a lo de Reflection, una **memoria episódica** de reflexiones acumuladas a lo largo de los intentos. La política decide cuándo reintentar y qué reflexiones arrastrar; la métrica es si el rendimiento mejora intento a intento, no en una sola pasada.

Se formalizó como agentes que mejoran decisiones futuras a partir de retroalimentación verbal, sin reentrenar el modelo.[27] La trampa: la memoria de reflexiones puede crecer sin límite y contaminar el contexto; conviene resumirla y quedarse con las lecciones que de verdad cambian la conducta.

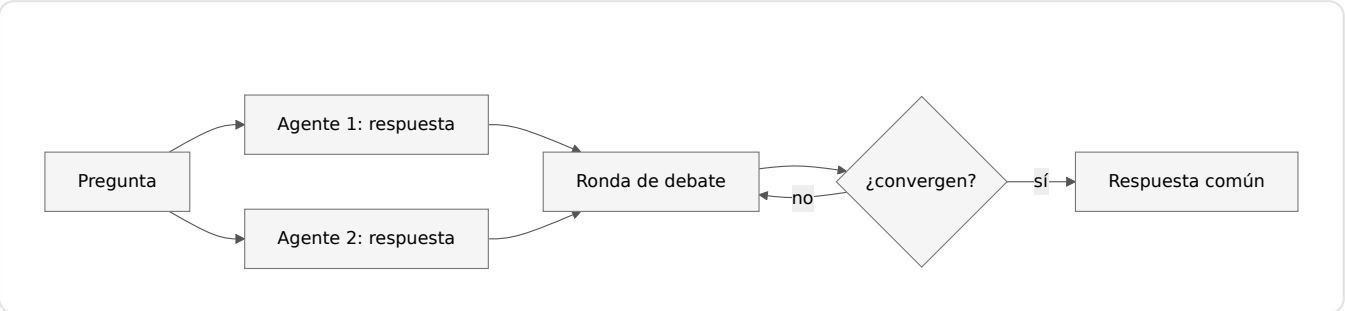


21. Arquitectura Debate

En Debate, varias instancias del modelo proponen su respuesta y luego **debaten** durante varias rondas: cada una ve las respuestas de las demás y revisa la suya, hasta converger en una respuesta común. La intuición es que un error aislado suele no resistir el escrutinio de otros razonadores, mientras que una respuesta correcta tiende a reforzarse cuando se confronta. Es la versión multiagente de pedir una segunda y tercera opinión.

El estado mínimo guarda las respuestas de cada agente en cada ronda y el criterio de convergencia. La política fija cuántos agentes y cuántas rondas; la métrica es si el debate mejora la factualidad y el razonamiento frente a un solo modelo, y si el coste de N agentes por K rondas se justifica.

El trabajo original mostró mejoras en razonamiento matemático y estratégico y una reducción de respuestas falaces.[28] El cuidado: si todos los agentes comparten el mismo sesgo, el debate converge con seguridad hacia el mismo error; la diversidad de prompts o de modelos es lo que da valor.

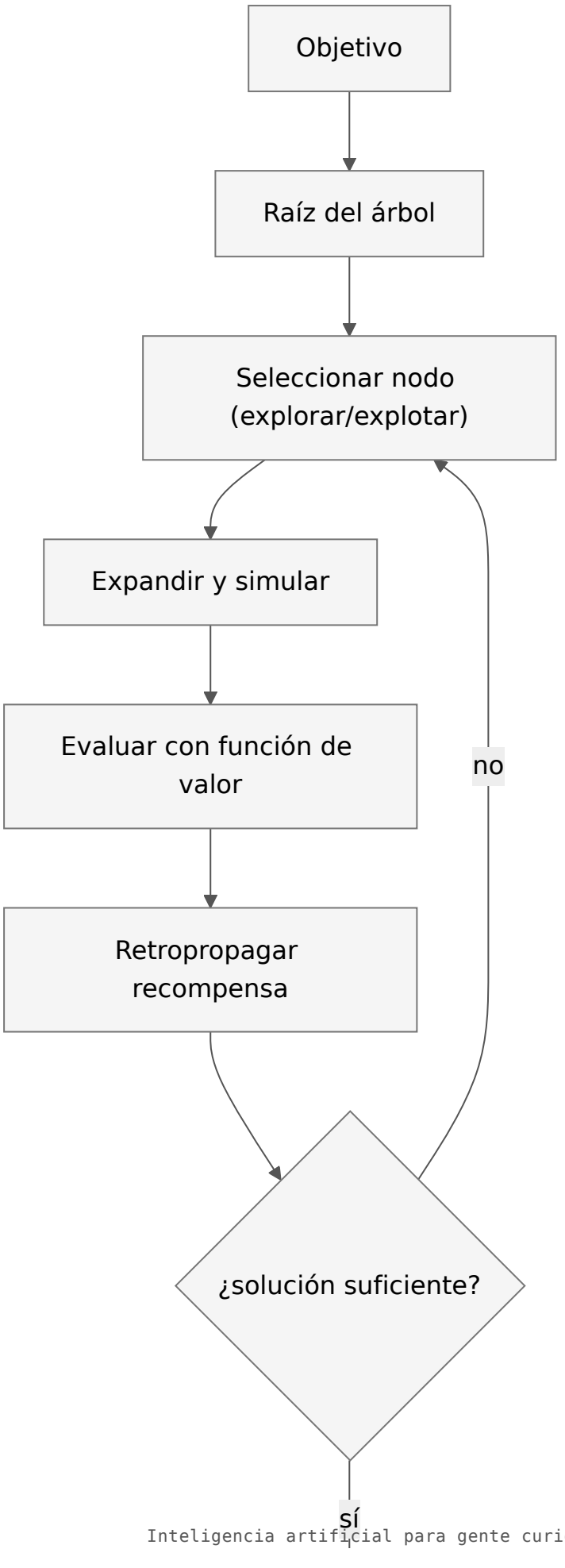


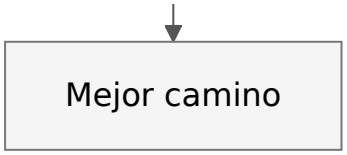
22. Arquitectura LATS

LATS (*Language Agent Tree Search*) une razonamiento, acción y planificación bajo una búsqueda en árbol. En lugar de seguir una única trayectoria como ReAct, mantiene un **árbol de posibles caminos** y lo explora con Montecarlo (MCTS): expande nodos prometedores, simula hacia delante, evalúa con una función de valor del propio modelo y retropropaga la recompensa, además de incorporar reflexiones cuando un camino falla. Es la arquitectura más cara del catálogo y la que más se acerca a un agente que de verdad planifica.

El estado mínimo es el árbol de nodos (cada uno con su estado, su valor estimado y sus visitas), más el historial de reflexiones. La política es la del MCTS: equilibrar explotar lo bueno conocido y explorar lo incierto. La métrica es si la búsqueda encuentra soluciones que una sola trayectoria no alcanza, y si el coste en llamadas lo compensa.

El marco se propuso como la primera unificación general de razonar, actuar y planificar, con resultados en programación, preguntas multietapa y navegación web.[29] La trampa es evidente: el árbol explota en coste; sin una función de valor decente y una poda agresiva, gastas muchísimo para una mejora marginal.


Objetivo
Raíz del árbol
Seleccionar nodo (explorar/explotar)
Expandir y simular
Evaluar con función de valor
Retropropagar recompensa
¿solución suficiente?
no
sí

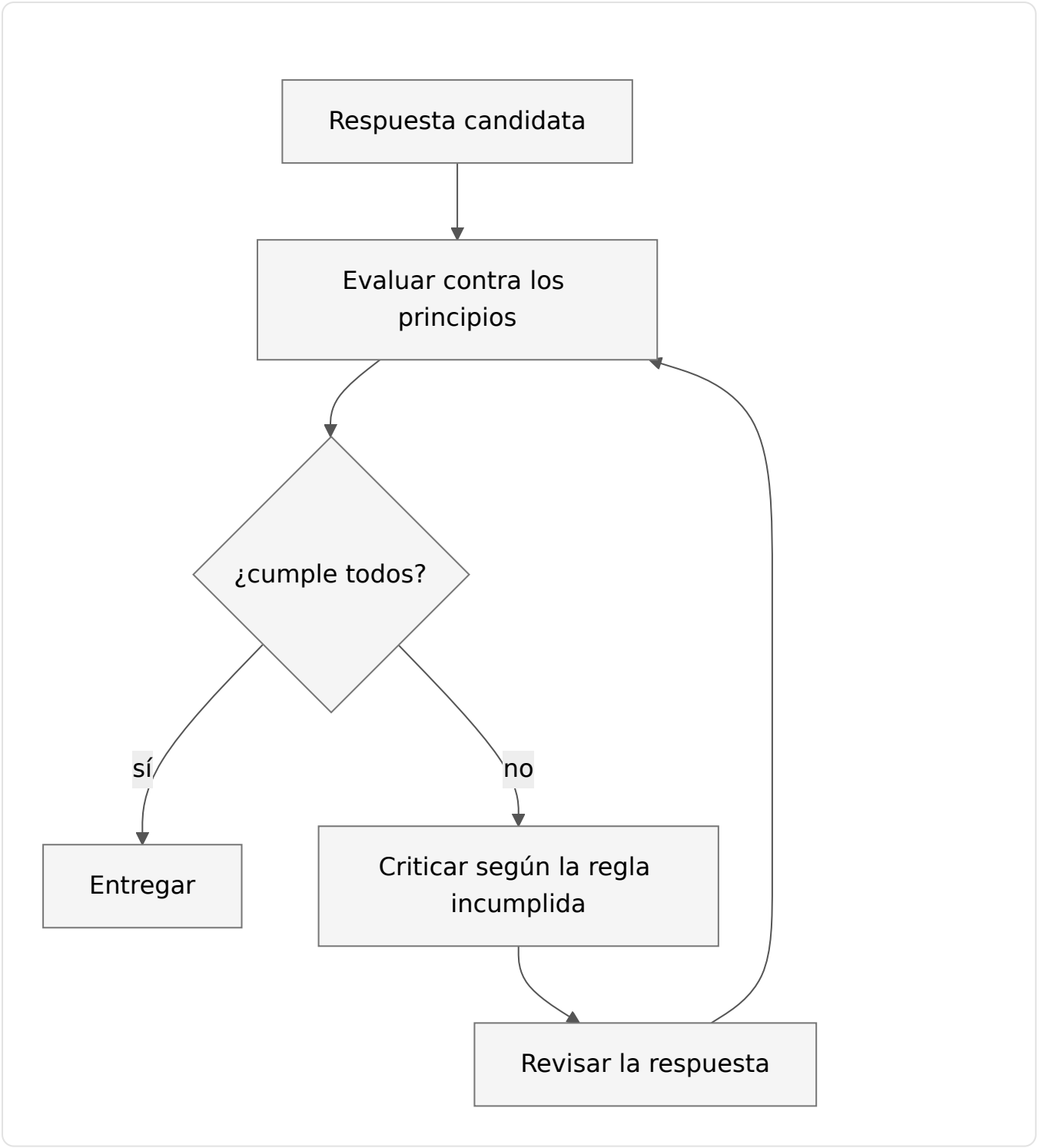


23. Arquitectura Constitutional AI

Constitutional AI sustituye al supervisor humano por un conjunto explícito de **principios**, una «constitución», frente a los que el propio modelo evalúa y revisa sus salidas. Ante una respuesta, otra pasada del modelo la critica según las reglas («¿incumple este principio?») y, si hace falta, la reescribe para cumplirlas. La gracia es que las reglas son visibles y editables, en lugar de quedar implícitas en miles de ejemplos de entrenamiento.

El estado mínimo contiene la respuesta candidata, el conjunto de principios y el resultado de evaluar cada uno (cumple o no, y por qué). La política decide si se entrega, se revisa o se bloquea; la métrica es la tasa de cumplimiento de los principios sin degradar la utilidad.

El método se introdujo en Anthropic como una forma de alinear modelos a partir de retroalimentación del propio modelo guiada por una constitución, en vez de solo etiquetas humanas.[30] Conecta de lleno con el capítulo de permisos y supervisión: una constitución es, en esencia, una capa de reglas declarativas sobre un modelo generativo. El cuidado: unos principios vagos producen revisiones vagas; las reglas deben ser concretas y comprobables.

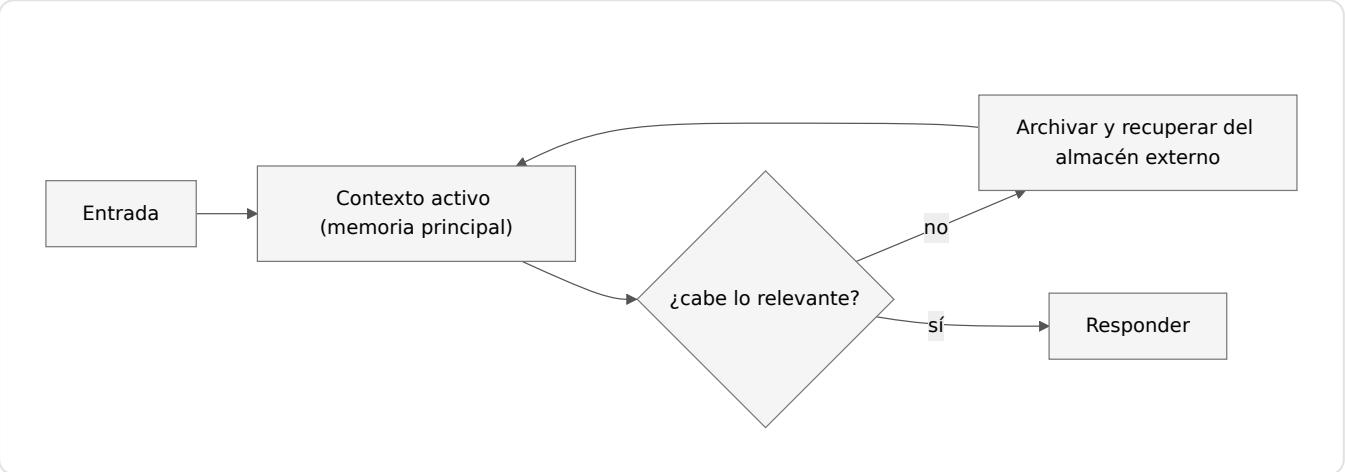


24. Arquitectura MemGPT

MemGPT trata la ventana de contexto como la memoria principal de un ordenador y gestiona el resto como si fuera disco: lo que no cabe en el contexto vive fuera, y el propio agente decide qué traer y qué archivar, paginando información entre ambos niveles. Así, una conversación o un documento mucho mayores que la ventana siguen siendo manejables, porque el agente mueve datos dentro y fuera según los necesita.

El estado mínimo distingue dos niveles: el contexto activo (rápido y limitado) y el almacén externo (amplio y lento), más las operaciones para mover información entre ellos. La política es la del propio agente: cuándo recuperar, cuándo resumir y archivar. La métrica es si mantiene la información relevante accesible sin saturar el contexto.

Se presentó con la metáfora explícita de los sistemas operativos, con niveles de memoria e interrupciones para gestionar el flujo.[31] Enlaza con el capítulo 04: es una forma concreta de la compactación y el *handoff* que ya viste, elevada a sistema. El cuidado: cada operación de paginar cuesta llamadas y latencia; si el agente pagina sin criterio, gasta más de lo que ahorra.

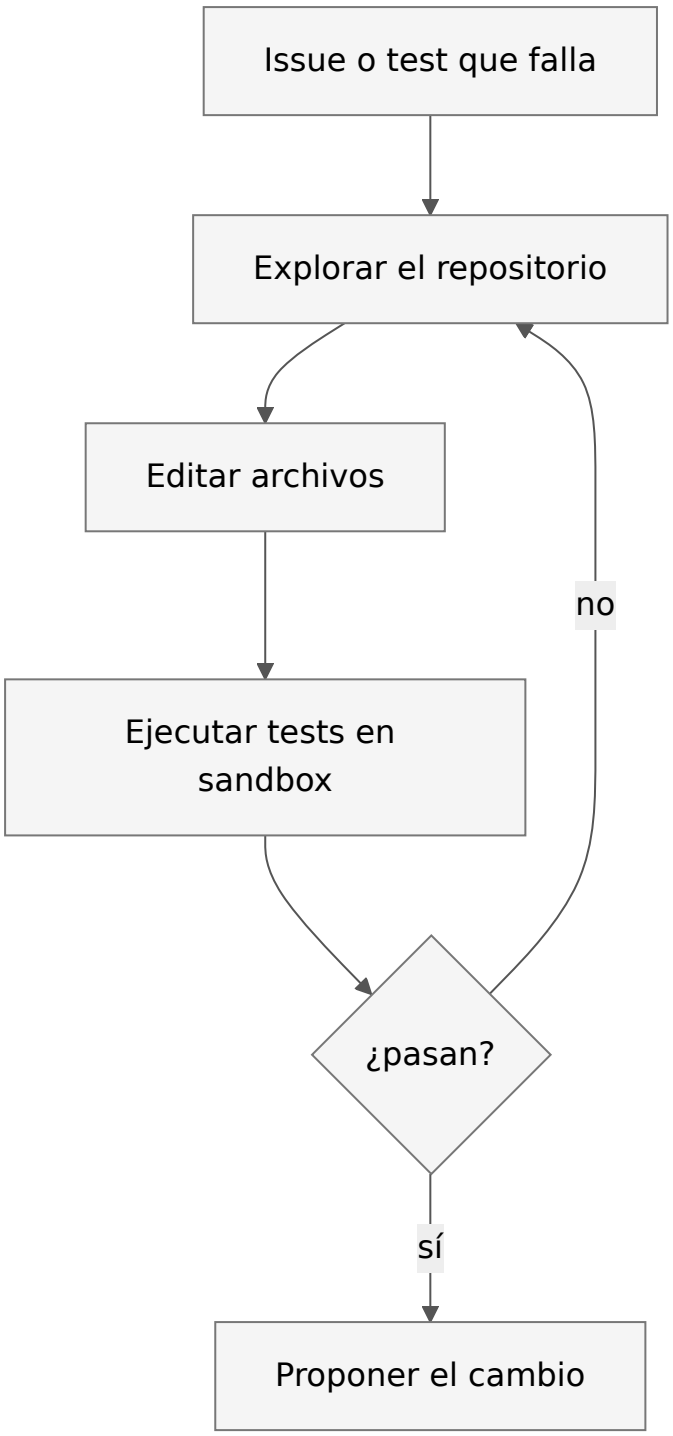


25. Arquitectura SWE-Agent

SWE-Agent es el patrón de los agentes que escriben código de verdad: dado un problema en un repositorio (un *issue* de GitHub, un test que falla), el agente navega los archivos, edita, ejecuta y vuelve a intentar dentro de un entorno aislado. Su aportación no fue un modelo nuevo, sino el diseño de la **interfaz agente-ordenador** (*agent-computer interface*): comandos pensados para que un modelo opere un sistema de archivos con pocos errores, en vez de darle una terminal cruda.

El estado mínimo contiene el repositorio, el objetivo, el historial de comandos y observaciones, y el resultado de los tests. La política decide qué archivo abrir, qué editar y cuándo ejecutar; la métrica es objetiva y dura: ¿pasa el test, se resuelve el *issue*? Es la arquitectura que mejor encaja con la idea de darle al agente una forma de verificar su trabajo.

El trabajo mostró que el diseño de la interfaz importa tanto como el modelo, y popularizó la evaluación sobre problemas reales de ingeniería de software.[32] Es el primo de investigación de herramientas como Claude Code: un agente con acceso a archivos, comandos y tests, bajo permisos. El cuidado: sin un entorno aislado y sin tests como juez, un agente que edita y ejecuta es tan peligroso como capaz.

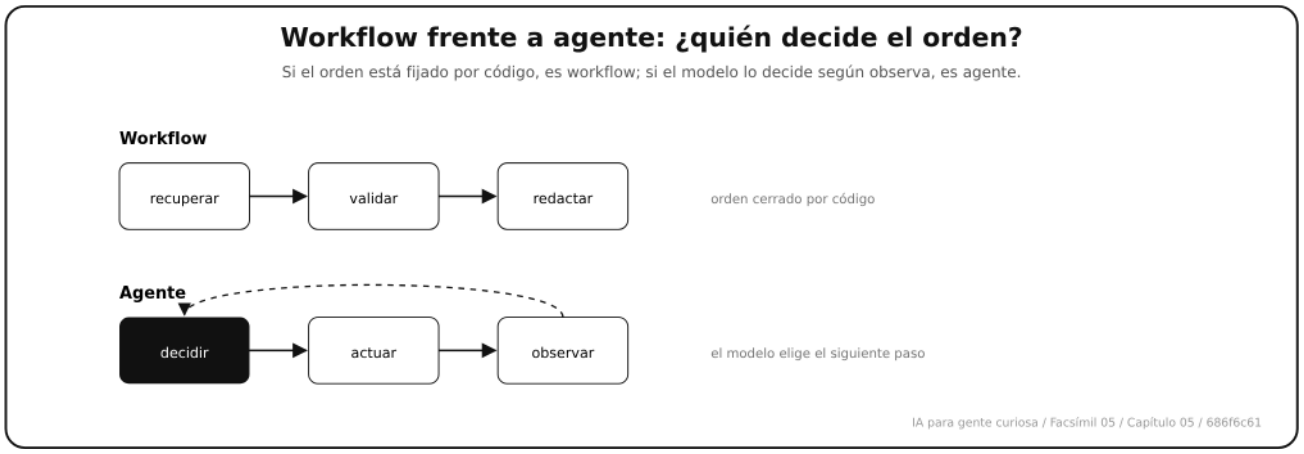


Notebooks y Colab

El repositorio está en GitHub y cada notebook se puede abrir en Colab con una URL directa. Esto es útil para clase: el alumno lee la explicación, ejecuta el patrón y luego lo traduce a su propio caso.

#	PATRÓN	NOTEBOOK	COLAB
01	Reflection	01_reflection.ipynb	Abrir
02	Tool use	02_tool_use.ipynb	Abrir
03	ReAct	03_ReAct.ipynb	Abrir
04	Planning	04_planning.ipynb	Abrir
05	Multi-agent	05_multi_agent.ipynb	Abrir
06	PEV	06_PEV.ipynb	Abrir
07	Blackboard	07_blackboard.ipynb	Abrir
08	Episodic + Semantic Memory	08_episodic_with_semantic.ipynb	Abrir
09	Tree of Thoughts	09_tree_of_thoughts.ipynb	Abrir
10	Mental Loop	10_mental_loop.ipynb	Abrir
11	Meta-controller	11_meta_controller.ipynb	Abrir
12	Graph memory	12_graph.ipynb	Abrir
13	Ensemble	13_ensemble.ipynb	Abrir
14	Dry-run harness	14_dry_run.ipynb	Abrir
15	Feedback loop	15_RLHF.ipynb	Abrir
16	Cellular automata	16_cellular_automata.ipynb	Abrir
17	Reflexive metacognitive	17_reflexive_metacognitive.ipynb	Abrir

Al abrir un Colab, fíjate en tres cosas antes de tocar código: qué estado se guarda, qué tools se declaran y qué métrica comprueba si el patrón funcionó. Si solo ves prompts largos y ninguna traza, todavía no tienes una arquitectura operable.

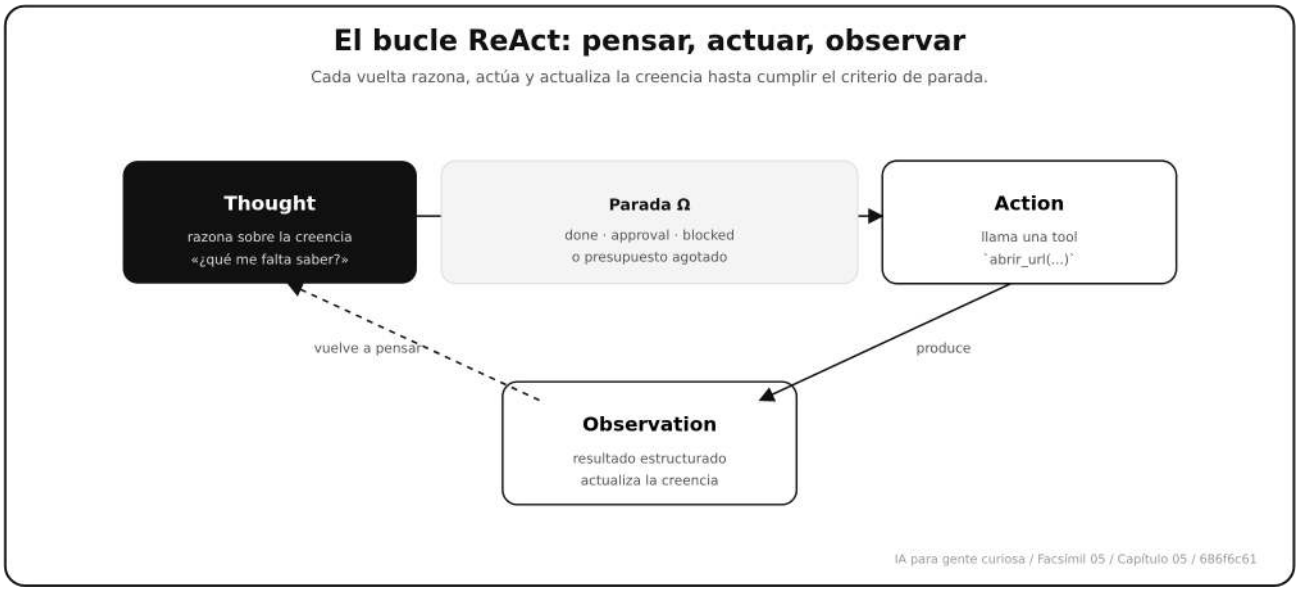


El bucle ReAct, paso a paso

ReAct es la arquitectura fundacional de las que vienen después: intercala pensamiento y acción.[33] Verlo en un trazo concreto vale más que la definición. El agente verifica una cita:

PASO	TIPO	CONTENIDO
1	Thought	«Necesito comprobar si la URL respalda la afirmación.»
2	Action	`abrir_url("https://...")`
3	Observation	«La página dice: *Agents have instructions, tools, handoffs…*»
4	Thought	«La afirmación coincide con el texto recuperado.»
5	Action	`responder("supported", evidencia)`

Cada Thought es una decisión sobre la creencia; cada Action consulta o cambia el mundo; cada Observation actualiza la creencia. Es el POMDP del capítulo 2 hecho explícito en texto, y por eso ReAct sirve de base para Reflexion (añade autocrítica), Plan-and-Solve (añade plan previo) y Tree of Thoughts (explora varias ramas).



Cómo elegir entre arquitecturas

La colección se puede leer como cinco familias:

FAMILIA	ARQUITECTURAS	DECISIÓN PRÁCTICA
Patrones fundacionales	Reflection, Tool use, ReAct, Planning.	Úsalos cuando todavía puedes resolver con un agente único.
Colaboración	Multi-agent, Blackboard, Meta-controller, Ensemble.	Úsalos cuando hay responsabilidades separables.
Memoria y razonamiento	Episodic + Semantic, Tree of Thoughts, Graph memory.	Úsalos cuando el problema exige recordar o explorar caminos.
Fiabilidad	PEV, Simulator, Dry-run, Reflexive metacognitive.	Úsalos cuando actuar sin verificar sale caro.
Aprendizaje y emergencia	Feedback loop, Cellular automata.	Úsalos cuando necesitas adaptar comportamiento a partir de señales.

Anthropic separa workflows y agents: los workflows siguen rutas predefinidas; los agents dejan que el modelo dirija dinámicamente el proceso y el uso de herramientas.[34] Esta distinción ayuda a ordenar la tabla. Planning puede ser workflow si el plan está cerrado. ReAct se acerca a agent cuando el siguiente paso depende de la observación. Multiagent puede ser workflow si los roles se ejecutan siempre igual. La arquitectura no la define el nombre: la define dónde está la decisión.

OpenAI Agents SDK ofrece una capa práctica para agentes con herramientas, handoffs, guardrails y trazas.[35] La documentación de tracing del SDK refuerza algo que aquí será constante: no evalúas solo la respuesta final, evalúas la trayectoria.[36] LangGraph, usado por el repositorio de Khan, encaja con estos patrones porque permite grafos con estado, ciclos y persistencia mediante checkpoints.[37]

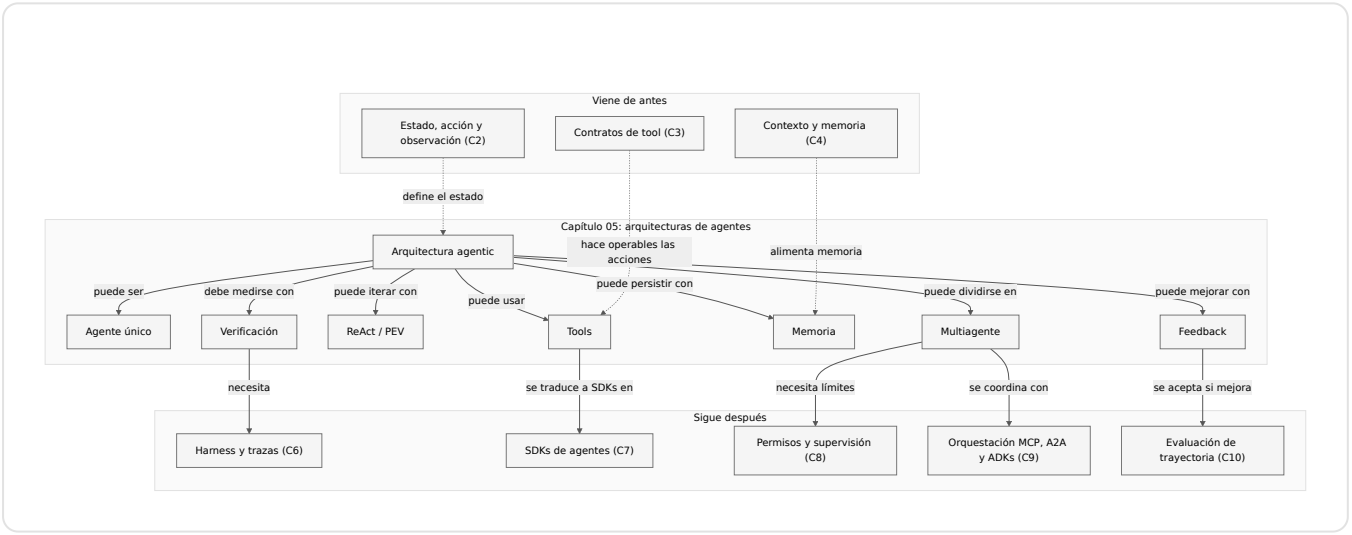
El coste oculto de complicar: cuándo no subir de escalón

La fórmula de selección de arquitectura tiene una asimetría que conviene mirar de frente, porque es la fuente de la mayoría de los proyectos de agentes que se atascan. Los términos positivos (la utilidad de un patrón) se ven en la demo: un sistema multiagente que se reparte el trabajo impresiona, un agente que planifica antes de actuar parece más inteligente, un ensemble que vota da respuestas más robustas. Los términos negativos (coste, latencia y, sobre todo, dificultad de verificación) no se ven en la demo: aparecen tres meses después, cuando hay que depurar por qué el sistema falló un martes a las tres de la tarde y la traza es una maraña de mensajes entre cinco agentes.

Esa asimetría sesga las decisiones hacia complicar de más. Conviene contar el caso típico, porque se repite. Un equipo tiene una tarea que un workflow fijo resolvería: recuperar unos documentos, validar un dato y redactar una respuesta, siempre en ese orden. Pero «workflow» suena poco ambicioso, así que se monta un sistema de agentes que deciden dinámicamente el siguiente paso. Funciona en la demo. En producción, empieza a tomar caminos distintos ante entradas casi idénticas, porque eso es lo que hace un sistema no determinista; el coste por tarea se dispara por las vueltas de más; y cuando algo falla, nadie sabe si fue el modelo, la herramienta o la coordinación, porque la trayectoria cambia cada vez. El equipo acaba añadiendo restricciones, gates y reglas hasta que, sin darse cuenta, ha reconstruido a mano el workflow fijo que evitó al principio, pero más caro y más frágil.

La lección no es que las arquitecturas complejas sean malas; es que su complejidad tiene que ganarse el sitio en la cuenta. Un agente con decisión dinámica solo compensa cuando el siguiente paso depende de verdad de lo observado, y no se conoce de antemano. Un sistema multiagente solo compensa cuando las subtareas son genuinamente independientes y se benefician de contextos separados, no cuando lo único que aporta es dividir un prompt en cinco. Un ensemble solo compensa cuando el coste de un error supera con holgura el coste de varias ejecuciones. En todos los casos, la pregunta correcta no es «¿esta arquitectura es más potente?», sino «¿la utilidad extra que aporta para *esta* tarea supera el coste, la latencia y la dificultad de verificación que añade?». Y como la utilidad se ve y los costes se esconden, la disciplina sana es la contraria a la intuición: empezar por el escalón más bajo que pueda funcionar, medirlo, y subir solo cuando la medición (no el entusiasmo) demuestre que el escalón actual se queda corto.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Arquitectura agentic	Patrón que organiza estado, tools, memoria, verificación y parada.
Reflection	Generar, criticar y revisar antes de entregar.
Tool use	Delegar una parte del trabajo a una función externa.
ReAct	Intercalar razonamiento, acción y observación.
Planning	Descomponer la tarea antes de ejecutarla.
PEV	Planificar, ejecutar y verificar.
Blackboard	Espacio compartido donde varios agentes escriben evidencias.
Meta-controlador	Router que elige especialista o flujo.
Ensemble	Varios agentes producen salidas y un agregador sintetiza.
Dry-run	Simulación previa antes de aplicar un cambio.
Self-consistency	Generar N cadenas de razonamiento y quedarse con la respuesta mayoritaria.
Chain-of-verification	Redactar, generar preguntas de verificación, responderlas aisladas y reescribir.
Reflexion	Guardar en memoria una nota verbal de cada fallo para mejorar en el siguiente intento.
Debate	Varios agentes proponen y debaten en rondas hasta converger.
LATS	Búsqueda en árbol (MCTS) sobre caminos de razonamiento y acción, con función de valor.
Constitutional AI	Evaluar y revisar las salidas contra un conjunto explícito de principios.
MemGPT	Gestionar la memoria por niveles, paginando entre el contexto y un almacén externo.
SWE-Agent	Agente que edita y ejecuta código en un entorno aislado, juzgado por los tests.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Elegir la arquitectura por moda	Un patrón complejo puede ocultar que bastaba una tool.	Empezar por la tarea y calcular coste, latencia y verificación.
Confundir multiagente con calidad	Más agentes pueden producir más ruido si no hay roles claros.	Definir responsabilidad, entrada y salida de cada especialista.
Usar memoria sin caducidad	La memoria vieja se vuelve una fuente falsa de certeza.	Guardar fuente, fecha, versión y criterio de recuperación.
Hacer ReAct sin parada	El bucle puede repetir observaciones sin aportar evidencia nueva.	Definir límite de pasos y criterio Ω.
Simular sin comprobar el simulador	Un dry-run malo tranquiliza sin proteger.	Comparar simulación con resultados reales en casos pequeños.
Copiar notebooks sin harness	El notebook enseña el patrón, pero no opera producción.	Añadir trazas, métricas, permisos, tests y rollback.

Antes de pasar página

- [ ] ¿Puedo recorrer el árbol de decisión y justificar por qué una hoja aplica o no aplica?
- [ ] ¿Puedo explicar qué problema resuelve cada una de las arquitecturas del catálogo?
- [ ] ¿Sé diferenciar Reflection, ReAct, Planning y PEV?
- [ ] ¿Sé cuándo usar un meta-controlador frente a un ensemble?
- [ ] ¿Entiendo por qué memoria episódica y semántica no son lo mismo?
- [ ] ¿Puedo justificar un dry-run con coste, evidencia y criterio de aprobación?
- [ ] ¿Sé abrir un notebook en Colab y localizar estado, tools y métrica?
- [ ] ¿Puedo traducir un patrón del notebook a $G, S, A, O, \pi, T, \Omega, B$?
- [ ] ¿Sé qué tendría que añadir para llevarlo a producción?

En resumen

IDEA FUERZA	DETALLE
La arquitectura es una decisión de control.	Decide cómo se reparte el bucle entre plan, tools, memoria y verificación.
No hay patrón superior en abstracto.	Hay patrones más o menos adecuados para cada perfil de tarea.
Los notebooks son material de trabajo.	Úsalos para aprender el patrón, no para copiar producción sin harness.
La complejidad se paga.	Multiagente, memoria y simulación aumentan coste y trazabilidad necesaria.
El siguiente capítulo baja a operación.	Harness, límites, sensores y trazas convierten patrones en sistemas medibles.

Para saber más

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

Anthropic. (2025). *Develop Tests for LLM Applications*. Documentación oficial.

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

Bonet, B. y Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence, 129*(1-2), 5-33.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A. C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S. y Zimmermann, A. (2021). Knowledge graphs. *ACM Computing Surveys, 54*(4). https://doi.org/10.1145/3447772

Keeney, R. L. y Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley.

Khan, F. (2026). *All Agentic Architectures*. Repositorio GitHub.

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

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Welleck, S., Majumder, B. P., Gupta, S., Yazdanbakhsh, A. y Clark, P. (2023). *Self-Refine: Iterative Refinement with Self-Feedback*. https://arxiv.org/abs/2303.17651

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. y Wilkins, D. (1998). *PDDL: The Planning Domain Definition Language Version 1.2*. Technical Report CVC TR-98-003/DCS TR-1165.

Nii, H. P. (1986). Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine, 7*(2), 38-53.

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

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

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J. y Lowe, R. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems, 35*, 27730-27744. https://arxiv.org/abs/2203.02155

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*. https://arxiv.org/abs/2304.03442

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

Shinn, N., Cassano, F., Labash, A., Gopinath, A., Narasimhan, K. y Yao, S. (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning*. https://arxiv.org/abs/2303.11366

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A. y Zhou, D. (2023). *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. *International Conference on Learning Representations*. https://arxiv.org/abs/2203.11171

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q. y Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems, 35*, 24824-24837. https://arxiv.org/abs/2201.11903

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y. y Narasimhan, K. (2023). *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. https://arxiv.org/abs/2305.10601

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. Khan, F. (2026). *All Agentic Architectures*. Repositorio GitHub. https://github.com/FareedKhan-dev/all-agentic-architectures. Consultado el 10 de junio de 2026. ↩

2. Keeney, R. L. y Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley. Formaliza la función de valor aditiva ponderada para decidir entre alternativas con varios criterios. ↩

3. Khan, F. (2026). *All Agentic Architectures*. Repositorio GitHub. https://github.com/FareedKhan-dev/all-agentic-architectures. Consultado el 28 de junio de 2026. ↩

4. Shinn, N. y otros (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning*. https://arxiv.org/abs/2303.11366 ↩

5. Madaan, A. y otros (2023). *Self-Refine: Iterative Refinement with Self-Feedback*. https://arxiv.org/abs/2303.17651 ↩

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

7. Anthropic. (2026). *How to implement tool use*. https://docs.anthropic.com/en/docs/agents-and-tools/tool-use/implement-tool-use. Consultado el 10 de junio de 2026. ↩

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

9. McDermott, D. y otros (1998). *PDDL: The Planning Domain Definition Language Version 1.2*. Technical Report CVC TR-98-003/DCS TR-1165. ↩

10. Bonet, B. y Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence, 129*(1-2), 5-33. ↩

11. Anthropic. (2025). *Develop Tests for LLM Applications*. https://platform.claude.com/docs/en/build-with-claude/develop-tests. ↩

12. Nii, H. P. (1986). Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine, 7*(2), 38-53. ↩

13. Park, J. S. y otros (2023). *Generative Agents: Interactive Simulacra of Human Behavior*. https://arxiv.org/abs/2304.03442 ↩

14. Yao, S. y otros (2023). *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. https://arxiv.org/abs/2305.10601 ↩

15. OpenAI. (2026). *Agents SDK*. https://developers.openai.com/api/docs/guides/agents. Consultado el 10 de junio de 2026. ↩

16. Hogan, A. y otros (2021). Knowledge graphs. *ACM Computing Surveys, 54*(4). https://doi.org/10.1145/3447772 ↩

17. Wang, X. y otros (2023). *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. *International Conference on Learning Representations*. https://arxiv.org/abs/2203.11171 ↩

18. OpenAI. (2026). *Agents SDK: Tracing*. https://openai.github.io/openai-agents-python/tracing/. Consultado el 10 de junio de 2026. ↩

19. Ouyang, L. y otros (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems, 35*, 27730-27744. https://arxiv.org/abs/2203.02155 ↩

20. Wei, J. y otros (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems, 35*, 24824-24837. https://arxiv.org/abs/2201.11903 ↩

21. Wang, X. y otros (2023). *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. *International Conference on Learning Representations*. https://arxiv.org/abs/2203.11171 ↩

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

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

24. Yao, S. y otros (2023). *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. https://arxiv.org/abs/2305.10601 ↩

25. Wang, X. y otros (2023). *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. International Conference on Learning Representations. https://arxiv.org/abs/2203.11171 ↩

26. Dhuliawala, S. y otros (2023). *Chain-of-Verification Reduces Hallucination in Large Language Models*. https://arxiv.org/abs/2309.11495 ↩

27. Shinn, N. y otros (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning*. https://arxiv.org/abs/2303.11366 ↩

28. Du, Y. y otros (2023). *Improving Factuality and Reasoning in Language Models through Multiagent Debate*. https://arxiv.org/abs/2305.14325 ↩

29. Zhou, A. y otros (2023). *Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models*. https://arxiv.org/abs/2310.04406 ↩

30. Bai, Y. y otros (2022). *Constitutional AI: Harmlessness from AI Feedback*. https://arxiv.org/abs/2212.08073 ↩

31. Packer, C. y otros (2023). *MemGPT: Towards LLMs as Operating Systems*. https://arxiv.org/abs/2310.08560 ↩

32. Yang, J. y otros (2024). *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. https://arxiv.org/abs/2405.15793 ↩

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

34. Anthropic. (2024). *Building Effective Agents*. https://www.anthropic.com/engineering/building-effective-agents. Consultado el 10 de junio de 2026. ↩

35. OpenAI. (2026). *Agents SDK*. https://developers.openai.com/api/docs/guides/agents. Consultado el 10 de junio de 2026. ↩

36. OpenAI. (2026). *Agents SDK: Tracing*. https://openai.github.io/openai-agents-python/tracing/. Consultado el 10 de junio de 2026. ↩

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