Facsímil 05

Agentes y orquestación

Capítulo 11

El bucle con verificador: el LLM que propone y el entorno que mide

686f6c61 · 2026

Capítulo 11: El bucle con verificador: el LLM que propone y el entorno que mide

Entrando en el tema

En el capítulo 2 vimos el ciclo que convierte un modelo en sistema: estado, acción, observación, actualización y parada. En el capítulo 5 ordenamos las arquitecturas que organizan ese ciclo, desde ReAct hasta los sistemas multiagente. Este capítulo coge una de esas arquitecturas y la lleva a su versión más rigurosa: el bucle cerrado donde la observación no es un texto que el agente interpreta, sino **una medida dura que el entorno calcula**.

La idea central es sencilla de enunciar y difícil de respetar: el patrón de agente más fiable es aquel en el que el LLM **propone** una acción, un **verificador** la comprueba y la **mide**, y ese feedback concreto guía el siguiente intento. El modelo es la política que sugiere; el entorno es la verdad que decide y puntúa. Cuando ese reparto está bien hecho, el agente deja de poder alucinar acciones: si propone algo incorrecto, el verificador lo rechaza antes de que cause daño; si propone algo correcto pero mediocre, la medida lo dice con un número.

Conviene separar dos verbos que se confunden. **Generar** es producir texto que suena plausible. **Optimizar** es producir una solución que mejora una cantidad medible. Un LLM por sí solo genera. Un LLM dentro de un bucle con verificador optimiza, porque cada propuesta se enfrenta a una señal externa que no se deja convencer por la elegancia de la redacción. La palabra técnica para ese apoyo en una señal externa comprobable es *grounding* (anclaje): anclar las decisiones del agente en algo que se puede comprobar, no en lo bien que queda el argumento.

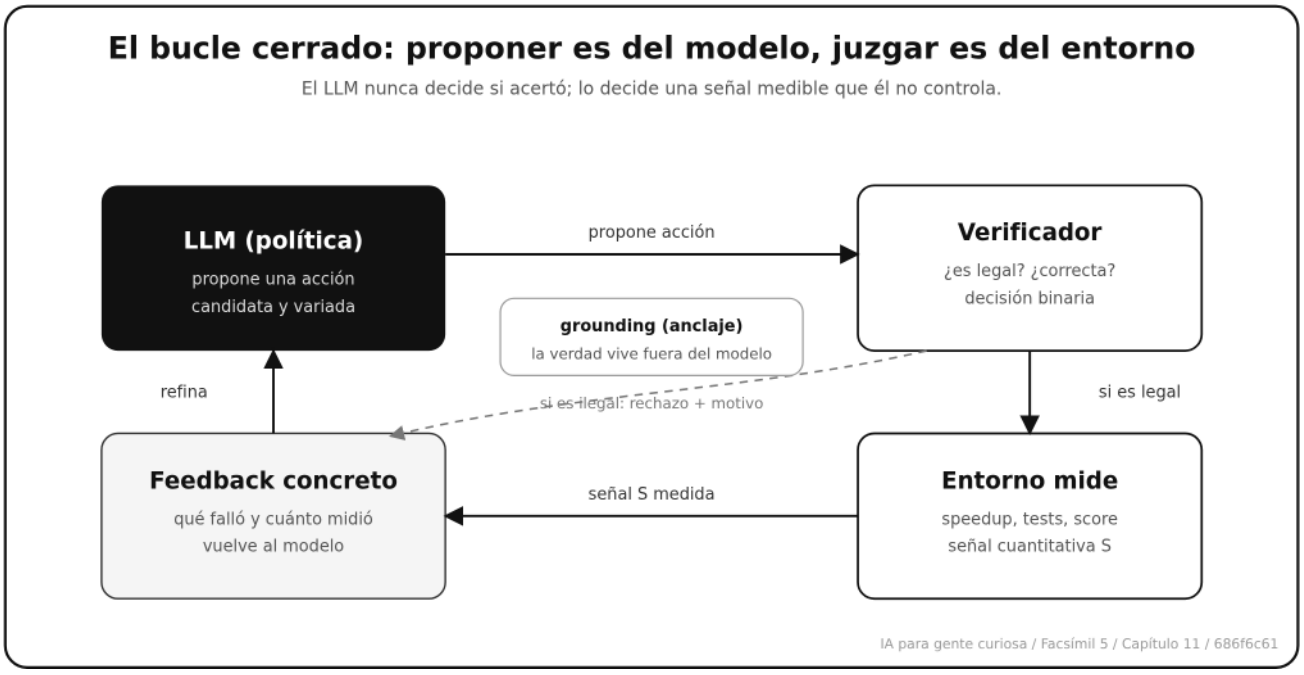
Este capítulo trata, en el fondo, de una sola pregunta de ingeniería: **¿de dónde sale la observación que cierra el bucle?** Si sale del propio modelo opinando sobre su trabajo, el sistema es frágil. Si sale de un entorno que compila, ejecuta, mide o comprueba, el sistema tiene de qué agarrarse.

El patrón: propón, verifica, mide, refina

El bucle tiene cuatro momentos y un reparto de papeles muy claro.

MOMENTO	QUIÉN LO HACE	QUÉ PRODUCE
Propón	El LLM (la política).	Una acción candidata: un cambio, una solución, un programa.
Verifica	El verificador (la verdad).	Una decisión binaria: ¿es válida, legal, correcta?
Mide	El entorno (la verdad).	Una señal cuantitativa: ¿cuánto mejora respecto al objetivo?
Refina	El LLM, con el feedback.	Una nueva propuesta, informada por lo que falló o midió.

La clave es que **verificar** y **medir** no las hace el modelo. El modelo no sabe si su propuesta es correcta; sabe que suena correcta, que es una cosa distinta y a veces opuesta. Por eso el patrón separa dos autoridades: el LLM tiene autoridad para **proponer** (porque genera candidatos buenos y variados), y el entorno tiene autoridad para **juzgar** (porque comprueba contra la realidad). Confundir ambas, dejar que el modelo se autoevalúe sin ancla, es la raíz de la mayoría de agentes que parecen funcionar en la demo y fallan en producción.



Visto así, el bucle es un primo riguroso del ciclo acción-observación del capítulo 2 y un caso particular de las arquitecturas de fiabilidad del capítulo 5 (PEV, dry-run, simulador). La diferencia de grado es importante: aquí la observación no admite interpretación. No es «la página parece respaldar la afirmación», sino «el programa pasa de tardar 1,8 segundos a tardar 0,7». No hay margen para que el modelo se cuente una historia favorable.

Por qué el anclaje gana a la generación libre

Un LLM genera texto que maximiza plausibilidad. Esa es su fuerza y su trampa. En tareas donde plausibilidad y corrección coinciden, generar basta. En tareas donde no coinciden, generar produce respuestas seguras de sí mismas y equivocadas. El anclaje resuelve ese desajuste poniendo entre el modelo y la decisión final un juez que no se deja convencer por la redacción.

Hay tres razones por las que el bucle con verificador es más robusto que la generación libre:

1. **El verificador filtra alucinaciones antes de que cuesten.** Una propuesta ilegal o incorrecta se rechaza en el sitio, sin llegar al usuario ni al sistema de producción. El coste de equivocarse baja de «incidente» a «un intento más en el bucle».
2. **La medida ordena las propuestas buenas.** Entre varias acciones legales, la señal cuantitativa dice cuál mejora más. El modelo no tiene que adivinar; el entorno se lo dice con un número comparable.
3. **El feedback es concreto, no genérico.** «Esta transformación es ilegal porque rompe una dependencia entre las iteraciones 3 y 5» es feedback accionable. «Inténtalo mejor» no lo es. Cuanto más específica es la observación, más útil es el siguiente intento.

La consecuencia práctica es que el patrón convierte un modelo falible en un sistema fiable **sin tener que mejorar el modelo**. El LLM sigue alucinando con la misma frecuencia; lo que cambia es que sus alucinaciones ya no salen del bucle. Esto conecta con una idea que recorre todo el facsímil: la fiabilidad de un agente se diseña en el arnés que lo rodea, no en el modelo que lo anima.

Un caso de estudio real: optimizar bucles guiado por un LLM

El ejemplo más limpio de este patrón viene de la compilación de alto rendimiento. Cuando un programa científico opera sobre matrices grandes (multiplicaciones, convoluciones, simulaciones), el orden en que se recorren los bucles, cómo se trocean en bloques que caben en la caché (*tiling*, tarjado en bloques), si se fusionan dos bucles en uno o si se reparten entre varios núcleos (paralelización) puede cambiar el tiempo de ejecución en un factor enorme. Elegir esas transformaciones se llama *auto-scheduling* (planificación automática de transformaciones), y es un problema de optimización clásico y duro: el espacio de combinaciones es inmenso y muchas combinaciones son ilegales porque cambian el resultado del cálculo.

El trabajo de Merouani, Kara Bernou y Baghdadi sobre *auto-scheduling* con agentes plantea exactamente el bucle de este capítulo.[1] Un LLM, sin entrenamiento específico para la tarea (*zero-shot*, sin ejemplos previos), **propone** una secuencia de transformaciones para un fragmento de código. El compilador poliédrico Tiramisu actúa como verificador y entorno en dos pasos:

- **Verifica la legalidad.** Mediante análisis de dependencias, Tiramisu comprueba si la transformación propuesta preserva el resultado del programa. Una transformación que reordena dos operaciones que dependen entre sí es ilegal y se rechaza, con el motivo. El modelo no puede colar un *schedule* (plan de transformaciones) que «parece» correcto: si rompe una dependencia, el compilador lo detecta.
- **Mide el speedup real.** Si la transformación es legal, se compila y se ejecuta, y se mide cuánto más rápido corre el programa respecto a la versión original. Esa cifra es la observación que cierra el bucle.

Con ese feedback (legal o no, y cuánto acelera), el LLM propone la siguiente secuencia. Es el patrón propón-verifica-mide-refina aplicado a un problema con décadas de literatura.

Los resultados que reporta el trabajo son los que importan, y son estos, sin retoque: en el conjunto de pruebas PolyBench, el sistema alcanza una **media geométrica de speedup de 2,66x en una sola tirada** y de **3,54x en best-of-5** (cinco propuestas, quedándose con la mejor), compitiendo con Pluto, un optimizador poliédrico de referencia.[2] Pluto representa el enfoque clásico: un algoritmo poliédrico que busca transformaciones mediante un modelo matemático del programa, sin LLM.[3]

Lo interesante para nosotros no es el número en sí, sino qué hace que el número sea creíble. Un LLM proponiendo transformaciones a pelo, sin verificador, sería peligroso: una transformación ilegal cambiaría silenciosamente el resultado de un cálculo científico. Lo que vuelve útil al sistema es que **el LLM nunca tiene la última palabra sobre la legalidad ni sobre la velocidad**. Propone; el compilador juzga. El modelo aporta intuición sobre qué combinaciones suelen funcionar (algo que un buscador clásico no tiene); el compilador aporta la garantía de que nada incorrecto se cuela. Cada uno hace lo que sabe hacer.

Tres fórmulas para entender el bucle

El caso anterior se apoya en tres cantidades que conviene mirar de frente, porque aparecen en cualquier bucle con verificador, no solo en compiladores.

Fórmula 1: el speedup

El speedup mide cuántas veces más rápida es la versión optimizada respecto a la original. Es la definición estándar de aceleración en computación de alto rendimiento.[4]

$$S = \frac{T_{orig}}{T_{opt}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
S	Speedup: factor de aceleración.	2,66 significa «2,66 veces más rápido».
T_{orig}	Tiempo de la versión original.	1,8 segundos.
T_{opt}	Tiempo de la versión optimizada.	0,68 segundos.

En palabras: el speedup es el cociente entre lo que tardaba antes y lo que tarda después. Por encima de 1 hay mejora; por debajo de 1, la «optimización» ha empeorado las cosas. Es la señal medible que el verificador devuelve y que el LLM usa para refinar.

Ejemplo numérico: si un bucle tardaba $T_{orig} = 1{,}8$ s y, tras aplicar *tiling* y fusión, tarda $T_{opt} = 0{,}68$ s, entonces $S = 1{,}8/0{,}68 \approx 2{,}65$. El programa va casi 2,65 veces más rápido. Si una propuesta deja el tiempo en $T_{opt} = 2{,}0$ s, entonces $S = 0{,}9$: es legal, pero ha empeorado, y el bucle la descarta.

Fórmula 2: la media geométrica de speedups

Cuando hay varios programas, no se resume su aceleración con una media aritmética, sino con la media geométrica, porque los speedups son razones y no cantidades aditivas.[5]

$$\bar{S}_{geo} = \left(\prod_{i=1}^{n} S_i \right)^{1/n}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\bar{S}_{geo}$	Media geométrica de los speedups.	2,66 en una tirada sobre PolyBench.
S_i	Speedup del programa i.	El speedup de cada *kernel* del conjunto.
n	Número de programas.	Los casos de PolyBench.
$\prod$	Producto de todos los S_i.	$S_1 \cdot S_2 \cdots S_n$.

En palabras: se multiplican todos los speedups y se toma la raíz n-ésima del producto. La media geométrica es la única media que trata igual «acelerar 4x» y «ralentizar a la cuarta parte» (1/4): su producto es 1, factor neutro. La media aritmética, en cambio, se deja arrastrar por un único speedup enorme y miente sobre el comportamiento típico.

Ejemplo numérico: supongamos dos programas con speedups $S_1 = 2$ y $S_2 = 8$. La media aritmética da $(2 + 8)/2 = 5$. La media geométrica da $\sqrt{2 \cdot 8} = \sqrt{16} = 4$. Ahora mira el caso simétrico: un programa que acelera 2x y otro que se ralentiza a la mitad ($S = 0{,}5$). La

aritmética da $(2 + 0{,}5)/2 = 1{,}25$, sugiriendo mejora; la geométrica da $\sqrt{2 \cdot 0{,}5} = \sqrt{1} = 1$, diciendo la verdad: en conjunto, ni mejora ni empeora. Por eso 2,66x como media geométrica es una afirmación honesta sobre el comportamiento típico, no un promedio inflado por un caso afortunado.

Fórmula 3: la mejora de best-of-k

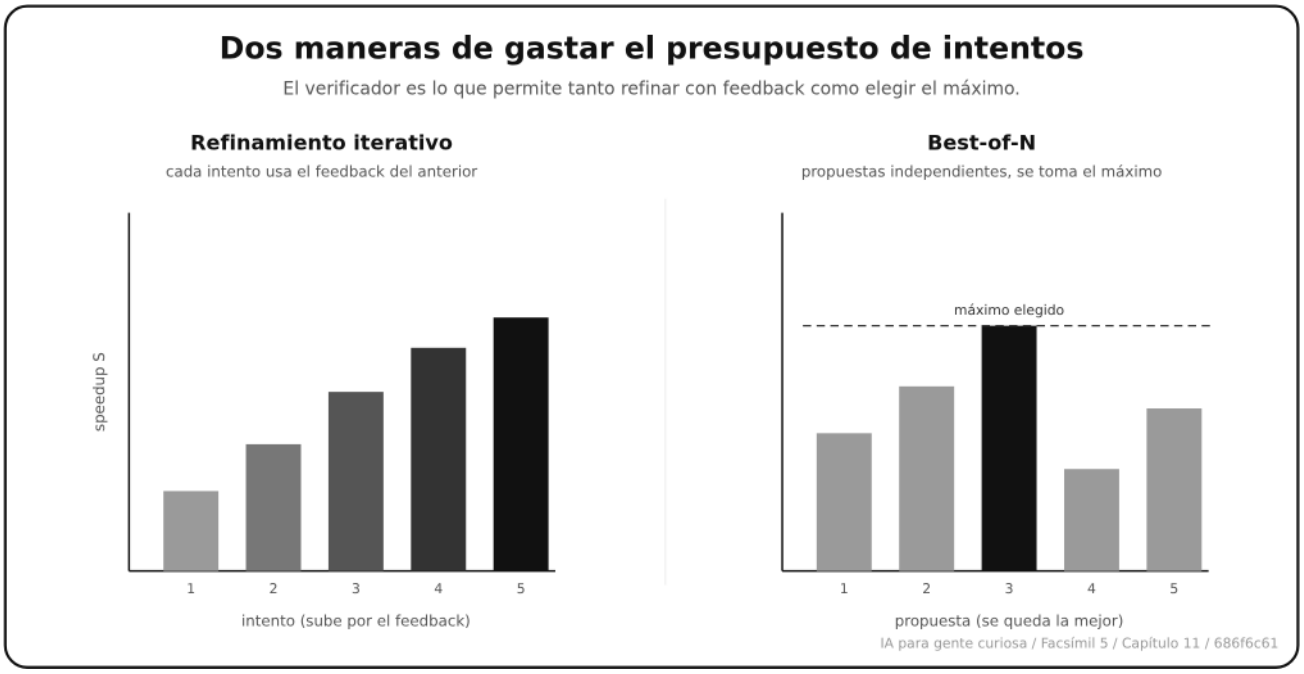
Best-of-N (mejor de N) consiste en generar varias propuestas independientes y quedarse con la que el verificador puntúa más alto. Es una técnica estándar para mejorar la salida de un modelo a costa de más cómputo, y la misma idea que muestrear varios razonamientos y agregar.[6]

$$S_{best\text{-}of\text{-}k} = \max\left(S^{(1)}, S^{(2)}, \ldots, S^{(k)}\right)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$S_{best\text{-}of\text{-}k}$	Speedup de la mejor de k propuestas.	El que se queda tras probar 5.
$S^{(j)}$	Speedup de la propuesta j.	Cada una de las 5 tiradas.
k	Número de propuestas generadas.	5 en best-of-5.
$\max$	Toma el máximo del conjunto.	Se queda con la más rápida.

En palabras: generas k candidatos, los verificas y mides todos, y te quedas con el mejor. Como el máximo de un conjunto nunca es peor que cualquiera de sus elementos, best-of-k siempre iguala o supera a una sola tirada. Aquí está la explicación de por qué 3,54x (best-of-5) supera a 2,66x (una tirada): no es que el modelo mejore, es que tomar el máximo de cinco intentos rescata, en cada programa, la propuesta más afortunada de las cinco. El verificador es lo que hace esto posible: solo se puede «quedarse con la mejor» si hay una señal objetiva que diga cuál es la mejor.

Ejemplo numérico (ilustrativo, no son cifras del trabajo): para un único programa, cinco tiradas del LLM producen speedups de $1{,}8$, $2{,}4$, $3{,}1$, $2{,}0$ y $2{,}9$. Una sola tirada, en promedio, rondaría $2{,}44$. Pero best-of-5 toma $\max(1{,}8, 2{,}4, 3{,}1, 2{,}0, 2{,}9) = 3{,}1$. El coste es cinco veces más llamadas al modelo y cinco compilaciones; la recompensa es quedarse siempre con la mejor exploración. Esa es la negociación de fondo: best-of-N compra calidad con cómputo, y solo merece la pena si el verificador es barato de ejecutar.



Las dos estrategias del dibujo no se excluyen. El refinamiento iterativo aprovecha el feedback concreto para subir intento a intento; best-of-*N* cubre el riesgo de quedarse atascado en una mala primera intuición generando varias semillas. Muchos sistemas reales combinan ambas: varias cadenas de refinamiento en paralelo y, al final, el máximo. En los dos casos, lo que las hace posibles es el mismo componente: un verificador que puntúa.

El patrón más allá del código

El bucle con verificador no es una técnica de compiladores. Es un patrón general que aparece cada vez que alguien pone una medida dura entre el modelo y la decisión. El compilador es solo un verificador especialmente nítido (legalidad por análisis de dependencias, velocidad por reloj). Cambia el verificador y tienes la misma arquitectura en otro dominio.

DOMINIO	EL LLM PROPONE	EL VERIFICADOR COMPRUEBA Y MIDE
Optimización de bucles	Secuencias de transformaciones.	Legalidad (dependencias) y speedup (reloj).
Optimización de prompts	Variantes de un prompt.	Acierto del prompt en un conjunto de ejemplos.
Auto-refinamiento de texto	Una redacción y su revisión.	Una rúbrica o un test sobre la salida.
Agente que escribe código	Una implementación.	La batería de tests (verde o rojo) y la cobertura.
Búsqueda de programas	Funciones candidatas.	La puntuación del programa en el problema.

Cuatro líneas de trabajo merecen nombre propio porque formalizan este patrón:

- **OPRO, «el LLM como optimizador».** En lugar de optimizar código, optimiza el propio prompt. El modelo propone una instrucción, se mide su acierto en un conjunto de ejemplos, y esa puntuación realimenta al modelo para que proponga una instrucción mejor. El LLM hace de optimizador y la métrica de acierto hace de verificador.[7]
- **DSPy.** Lleva la idea a la ingeniería de sistemas con LLM: en vez de escribir prompts a mano, declaras qué quieres y un optimizador compila los prompts y ejemplos buscando los que maximizan una métrica sobre datos. El bucle propón-mide-refina se vuelve infraestructura.[8]
- **Self-Refine.** El modelo genera una salida, produce feedback sobre ella y la revisa, sin entrenamiento adicional. Es el bucle en su forma mínima, y también su forma más frágil: cuando el «verificador» es el propio modelo opinando, hay que vigilar que la crítica sea concreta y, a poder ser, anclada en algo externo (un test, un dato), o solo produces una segunda respuesta igual de segura y cara.[9]
- **FunSearch.** Empareja un LLM que propone programas con un evaluador automático que los puntúa, y mediante búsqueda evolutiva encuentra programas que superan lo conocido en problemas matemáticos. El verificador (la puntuación del programa) es lo que permite quedarse con los buenos y descartar las alucinaciones.[10]

El hilo común es el del capítulo 5: ReAct ya intercalaba razonamiento y acción con observaciones de herramientas.[11] El bucle con verificador es ReAct llevado al extremo del rigor: la observación deja de ser texto que el agente interpreta y pasa a ser una medida que el entorno calcula. Y enlaza con el capítulo 10: allí la evaluación anclaba la decisión de publicar; aquí la evaluación está dentro del propio bucle de resolución, no después.

Una conexión que conviene nombrar

Este patrón es, en el fondo, búsqueda con recompensa medible. Quien haya leído sobre aprendizaje por refuerzo reconocerá las piezas: el LLM es la **política** que elige acciones, el speedup (o el test, o el score) es la **recompensa**, y el bucle es **exploración** del espacio de soluciones. La diferencia, y es enorme, es que aquí **no se entrena nada**. El modelo es *zero-shot*: no actualiza sus pesos con la recompensa. La recompensa solo sirve para filtrar y ordenar propuestas dentro de una misma sesión, no para mejorar el modelo a futuro.

Esa distinción tiene una consecuencia tranquilizadora para quien construye sistemas: no necesitas montar una infraestructura de entrenamiento por refuerzo para aprovechar la idea más valiosa del refuerzo, que es **optimizar contra una señal medible**. Basta con un modelo que proponga, un verificador que mida y un bucle que se quede con lo mejor. El Facsímil 10 desarrolla el refuerzo con entrenamiento; este capítulo muestra que su esqueleto conceptual (política, recompensa, búsqueda) ya rinde sin tocar los pesos, siempre que la señal sea medible y honesta. Y eso último, que la señal sea medible y honesta, es justo lo que el Facsímil 7 trata de anclar.

Límites: cuándo el bucle no salva

El patrón es potente, no mágico. Tiene cuatro límites que conviene tener delante antes de prometer demasiado.

LÍMITE	EN QUÉ CONSISTE
El verificador caro	Si comprobar y medir cuesta tanto como resolver, el bucle no compensa.
El verificador poco fiable	Si la señal es ruidosa o sesgada, refinar contra ella no mejora nada.
El coste del bucle	Muchas llamadas al modelo y muchas verificaciones; best-of-N lo multiplica.
El no determinismo	El mismo punto de partida puede dar trayectorias distintas; cuesta reproducir.

El más insidioso merece su propia palabra. El *reward hacking* (manipulación de la recompensa) ocurre cuando el sistema optimiza la **métrica medida** en lugar del **objetivo real**, explotando un verificador imperfecto. Si premias «el programa pasa los tests» y los tests son débiles, el agente puede escribir código que pasa los tests sin resolver el problema, e incluso borrando el caso que falla. Si premias «el speedup medido» y la medición se hace con una entrada de juguete, el agente puede encontrar una transformación rapidísima en ese caso concreto e inútil en datos reales. El verificador no es neutral: define qué significa «mejor», y el agente perseguirá esa definición hasta sus últimas consecuencias, incluidas las que no querías.

La defensa es la misma que en el capítulo 10: el verificador tiene que medir lo que de verdad importa, con casos representativos, y conviene desconfiar de mejoras espectaculares en una sola métrica. Un speedup de 50x debería disparar una alarma antes que una celebración: lo más probable es que el agente haya encontrado una grieta en la medición, no un milagro de optimización. La señal medible es el cimiento del bucle, y un cimiento que el propio agente puede minar si lo dejamos.

En el día a día

Aunque nunca toques un compilador, este patrón ya está debajo de herramientas que probablemente usas.

- **Un agente de programación que ejecuta los tests.** Cuando un asistente escribe código, lo ejecuta contra una batería de tests y corrige según los fallos, está cerrando el bucle: propone código, el test verifica, el resultado (verde o rojo, con el mensaje de error) realimenta. La diferencia entre un asistente que «sugiere código» y uno que «resuelve la tarea» es casi siempre si tiene o no un verificador ejecutable.
- **La optimización de prompts en una plataforma.** Cuando una herramienta prueba variantes de tu prompt y se queda con la que mejor puntúa en tus ejemplos, está haciendo OPRO sin decírtelo: el LLM propone, tu conjunto de ejemplos verifica.
- **Un corrector de estilo con reglas.** Un agente que reescribe un texto y lo pasa por un validador (longitud, ausencia de palabras prohibidas, formato de citas) y reintenta hasta cumplir, es el bucle con un verificador modesto pero real.
- **Un generador de consultas SQL que comprueba contra el esquema.** Propone una consulta, la valida contra el esquema y, si falla, corrige con el mensaje de error de la base de datos. El esquema es el verificador.

El denominador común: en todos los casos, alguien decidió poner una comprobación dura en el bucle, y eso convirtió un generador en un solucionador.

Por qué debería importarte

Porque distingue los agentes que funcionan de los que solo lo parecen, y te da un criterio para construir y para comprar.

Si vas a **construir** un agente para una tarea, la primera pregunta no debería ser «qué modelo uso», sino «¿qué verificador puedo poner en el bucle?». Si encuentras una señal barata y fiable (un test, un cálculo, un esquema, una métrica), tienes media batalla ganada: el patrón de este capítulo te dará fiabilidad casi gratis. Si no encuentras ningún verificador y dependes de que el modelo se autoevalúe, debes saber que estás en terreno frágil y diseñar con esa fragilidad en mente.

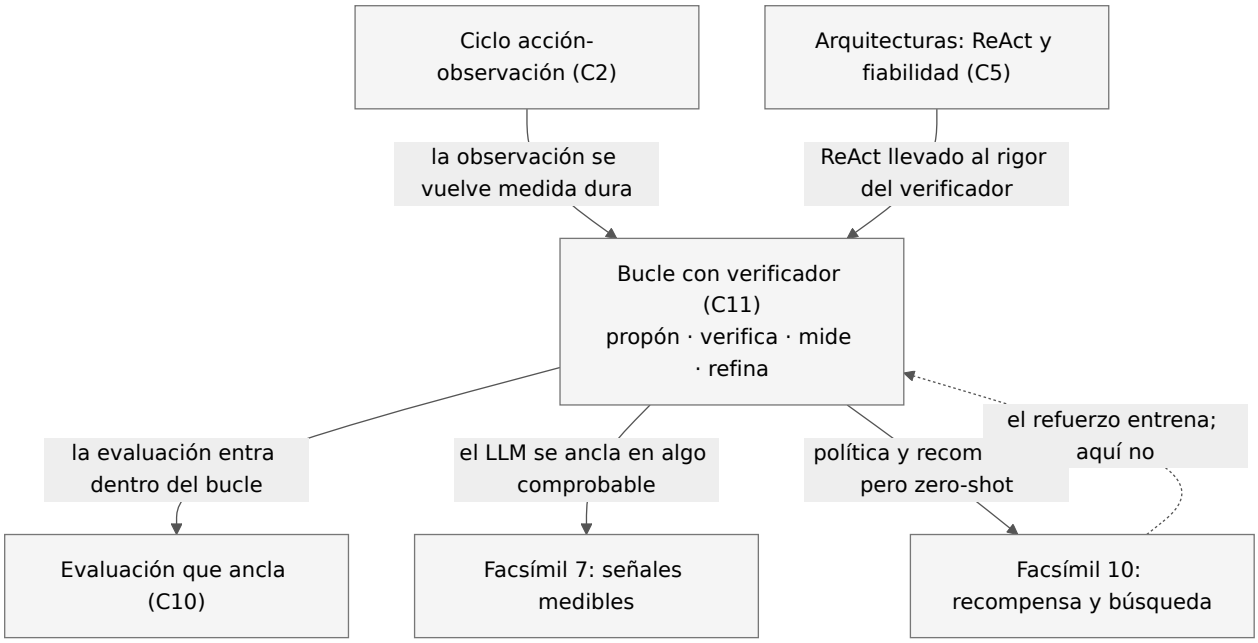
Si vas a **comprar o confiar** en una herramienta agentic, la pregunta reveladora es la misma: «¿contra qué se ancla?». Una herramienta que ejecuta tests, compila, valida contra un esquema o mide un resultado real es de otra categoría que una que solo encadena llamadas a un modelo y confía en su salida. El anclaje no se ve en la demo, pero es lo que decide si el sistema aguanta en producción.

Y como usuario, te da una lente para leer las promesas. Cuando alguien presume de un agente que «razona» y «mejora iterativamente», la pregunta honesta es: mejora, ¿medido cómo? Si la respuesta es «el modelo cree que mejora», desconfía. Si la respuesta es «pasa de tardar 1,8 a tardar 0,7 segundos, comprobado», estás ante un sistema serio.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Dejar que el modelo se autoevalúe sin ancla	La crítica del propio modelo hereda sus mismos sesgos y alucinaciones.	Poner un verificador externo: test, cálculo, esquema, métrica medible.
Confiar en una sola tirada	El no determinismo hace que un buen resultado no se repita.	Medir con repetición y, si compensa, usar best-of-N con el verificador.
Resumir speedups con media aritmética	Un único caso afortunado infla el promedio y miente.	Usar la media geométrica para razones y aceleraciones.
Celebrar una métrica disparada	Casi siempre es reward hacking, no un milagro.	Sospechar de mejoras enormes; revisar si la medición tiene una grieta.
Olvidar el coste del bucle	Best-of-N y el refinamiento multiplican las llamadas y las verificaciones.	Contar coste por tarea aceptada, no por llamada (capítulo 10).
Verificador caro como cuello de botella	Si medir cuesta como resolver, el bucle deja de compensar.	Buscar una señal barata; medir en un caso pequeño antes del completo.

Cómo encaja todo



El bucle con verificador es el punto de encuentro de varias ideas del facsímil. Toma el ciclo acción-observación del capítulo 2 y endurece la observación hasta volverla una medida que no admite interpretación. Coge ReAct y las arquitecturas de fiabilidad del capítulo 5 y les pone una verdad externa en el centro. Comparte con la evaluación del capítulo 10 la disciplina de medir, pero la mueve de «después de resolver» a «mientras se resuelve». Y enseña, sin entrenar nada, el esqueleto del refuerzo que el Facsímil 10 desarrollará con todas sus letras: una política que propone, una recompensa que mide y una búsqueda que se queda con lo mejor. La diferencia, que el modelo no aprende de la recompensa, es justo lo que hace que este patrón sea aplicable hoy, con cualquier modelo, sin infraestructura de entrenamiento.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Bucle con verificador	El agente propone, un verificador comprueba y mide, y el feedback guía el siguiente intento.
Verificador	Componente que decide si una propuesta es válida y la puntúa con una señal medible.
LLM como optimizador	Usar un modelo como política que propone soluciones a un problema de optimización.
Anclaje (grounding)	Apoyar las decisiones en una señal externa comprobable, no en la plausibilidad del texto.
Best-of-N	Generar N propuestas y quedarse con la mejor según el verificador.
Auto-scheduling	Elegir automáticamente las transformaciones de un programa para acelerarlo.
Transformación legal	Cambio que preserva el resultado; el verificador rechaza las ilegales.
Refinamiento iterativo	Mejorar una propuesta paso a paso usando el feedback de cada intento.
Reward hacking	Optimizar la métrica medida en lugar del objetivo real, explotando un verificador imperfecto.

Antes de pasar página

- [ ] ¿Puedo explicar el reparto de papeles: el LLM propone, el verificador y el entorno juzgan?
- [ ] ¿Entiendo por qué el anclaje en una señal medible gana a la generación libre?
- [ ] ¿Sé contar el caso del auto-scheduling con compilador y por qué el LLM nunca decide la legalidad?
- [ ] ¿Sé calcular un speedup y por qué se resume con media geométrica y no aritmética?
- [ ] ¿Entiendo por qué best-of-5 (3,54x) supera a una tirada (2,66x) sin que el modelo mejore?
- [ ] ¿Reconozco el patrón en agentes de código, optimización de prompts, Self-Refine y FunSearch?
- [ ] ¿Sé qué es el reward hacking y por qué una mejora enorme debe levantar sospecha?

- [ ] ¿Puedo elegir, para una tarea mía, un verificador barato y fiable?

En resumen

IDEA FUERZA	DETALLE
Proponer es del modelo; juzgar es del entorno.	El LLM nunca tiene la última palabra sobre legalidad ni sobre la medida.
El anclaje convierte un generador en un solucionador.	La observación deja de interpretarse y pasa a calcularse.
El patrón da fiabilidad sin mejorar el modelo.	Las alucinaciones siguen, pero el verificador no las deja salir del bucle.
Best-of-N compra calidad con cómputo.	Tomar el máximo de varias propuestas iguala o mejora una sola tirada.
El verificador define «mejor»: cuídalo.	Una señal débil invita al reward hacking; mide lo que de verdad importa.

Para saber más

Bondhugula, U., Hartono, A., Ramanujam, J. y Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, 101-113. https://doi.org/10.1145/1375581.1375595

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M. y Potts, C. (2023). *DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines*. https://arxiv.org/abs/2310.03714

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

Merouani, M., Kara Bernou, N. y Baghdadi, R. (2025). *Agentic Auto-Scheduling: Guiding a Polyhedral Compiler with a Large Language Model*. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2025).

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P. y Fawzi, A. (2024). Mathematical discoveries from program search with large language models. *Nature, 625*, 468-475. https://doi.org/10.1038/s41586-023-06924-6

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D. y Chen, X. (2023). *Large Language Models as Optimizers*. https://arxiv.org/abs/2309.03409

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. Merouani, M., Kara Bernou, N. y Baghdadi, R. (2025). *Agentic Auto-Scheduling: Guiding a Polyhedral Compiler with a Large Language Model*. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2025). ↩

2. Merouani, M., Kara Bernou, N. y Baghdadi, R. (2025). *Agentic Auto-Scheduling: Guiding a Polyhedral Compiler with a Large Language Model*. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2025). ↩

3. Bondhugula, U., Hartono, A., Ramanujam, J. y Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, 101-113. https://doi.org/10.1145/1375581.1375595 ↩

4. Bondhugula, U., Hartono, A., Ramanujam, J. y Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, 101-113. https://doi.org/10.1145/1375581.1375595 ↩

5. Merouani, M., Kara Bernou, N. y Baghdadi, R. (2025). *Agentic Auto-Scheduling: Guiding a Polyhedral Compiler with a Large Language Model*. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2025). ↩

6. Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D. y Chen, X. (2023). *Large Language Models as Optimizers*. https://arxiv.org/abs/2309.03409 ↩

7. Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D. y Chen, X. (2023). *Large Language Models as Optimizers*. https://arxiv.org/abs/2309.03409 ↩

8. Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M. y Potts, C. (2023). *DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines*.

https://arxiv.org/abs/2310.03714 ↩

9. 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 ↩

.0. Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P. y Fawzi, A. (2024). Mathematical discoveries from program search with large language models. *Nature, 625*, 468-475. https://doi.org/10.1038/s41586-023-06924-6 ↩

.1. 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 ↩