Facsímil 05

Agentes y orquestación

Capítulo 10

Evaluar agentes: trayectoria, coste y gates

686f6c61 · 2026

Capítulo 10: Evaluar agentes: trayectoria, coste y gates

Un agente puede acertar por el camino equivocado

En una aplicación clásica, muchas veces basta con comprobar la salida: esta función recibe `x` y devuelve `y`. En un agente, eso se queda corto. Un agente puede dar una respuesta final aceptable después de usar la tool equivocada, consultar demasiadas fuentes, saltarse una aprobación, repetir un paso, gastar demasiado o llegar a una conclusión que no puede reconstruirse.

En el capítulo 06 pusimos harness, límites, sensores y trazas. En el capítulo 08 diseñamos permisos y aprobación humana. En el capítulo 09 construimos routing entre tool local, MCP, A2A y workflows. Ahora toca una pregunta incómoda y necesaria: **¿cómo sabemos que todo eso funciona mejor, y no solo que parece funcionar?**

La evaluación de agentes tiene que mirar tres cosas a la vez: el resultado final, la trayectoria y el coste operativo. Si falta una, podemos engañarnos. Para ingeniería del software, además, hay una cuarta capa: **el cambio**. Una evaluación seria no solo responde “¿funciona esta demo?”, sino “¿puedo cambiar el prompt, el modelo, el router, una tool o el dataset y saber si he mejorado o he roto algo?”.

Qué le pediría a una clase de ingeniería del software

Si este capítulo se convirtiera en práctica universitaria, no lo plantearía como “mide si responde bien”. Lo plantearía como un sistema evaluable, versionado y desplegable.

COMPETENCIA	PREGUNTA DE INGENIERÍA	EVIDENCIA QUE DEBERÍA PRODUCIR EL ALUMNO
Requisitos observables	¿Qué significa “bien” sin depender de una opinión suelta?	Rúbrica, criterios de aceptación y casos con `why_it_exists`.
Diseño de pruebas	¿Qué capas se prueban por separado y cuáles integradas?	Tests de schema, tool contract, trayectoria, escenario y gate.
Trazabilidad	¿Puedes reconstruir por qué el agente decidió algo?	`trace_id`, spans, eventos, argumentos, resultados y versiones.
Reproducibilidad	¿Otra persona puede repetir la evaluación?	Dataset versionado, modelo fijado, prompt versionado, seed si aplica y fixtures.
Control de regresiones	¿Lo que corregiste ayer queda protegido mañana?	Caso nuevo en el dataset y comparación `baseline` contra `candidate`.
Estadística mínima	¿La mejora es señal o ruido?	Tamaño de muestra, intervalo, repetición de runs y tolerancia de cambio.
Operación	¿Qué ocurre si esto llega a producción?	Coste por tarea aceptada, p95 de latencia, rate limits y alertas.
Integración continua	¿Dónde se bloquea un cambio?	Gate de PR, gate nocturno, gate de prepublicación y canary.

Lo importante para el alumno: un agente no se evalúa como una función pura, pero tampoco como una caja negra. Se evalúa como **software no determinista con efectos, dependencias externas y trazas**.

Qué no es evaluar un agente

Evaluar un agente no es leer diez conversaciones bonitas. Eso sirve para intuición inicial, pero no para publicar ni comparar versiones.

Tampoco es pedirle a otro modelo “ponle nota” sin definir criterios. Un evaluador puede ayudar, pero necesita rúbrica, ejemplos, calibración y casos donde sepamos la respuesta. Si no, cambiaremos una caja negra por otra.

Y no es medir solo exactitud. Un sistema que acierta un 90% pero cuesta el triple, tarda 40 segundos, exige revisión manual constante o falla justo en tareas críticas no está listo para un producto serio.

La definición útil

Para este facsímil, evaluar un agente es:

> Ejecutar tareas representativas, capturar la traza completa, puntuar resultado y trayectoria, aplicar gates de coste y permisos, y comparar versiones con criterios repetibles.

Cada ejecución (run) deja un registro con varios campos. No es una ecuación de la literatura, es el registro que guardamos para poder evaluar:

SÍMBOLO	SIGNIFICADO	EJEMPLO
r	Run o ejecución evaluada.	Una petición de alumno resuelta por el agente.
x	Entrada del caso.	“Comprueba una cita y genera referencia APA”.
y	Salida final.	Respuesta, informe, diff, JSON o artefacto.
τ	Trayectoria.	Secuencia de modelo, tool, observación, decisión y parada.
c	Coste.	Euros, tokens, llamadas a tools, revisión humana.
l	Latencia.	Tiempo total y p95 por paso.
g	Gates aplicados.	`quality_gate`, `budget_gate`, `policy_gate`.

Y una puntuación útil es una función de valor aditiva, la decisión multicriterio de siempre: sumar lo que aporta y restar coste y latencia, cada término con su peso.[1]

$$S(r) = w_y S_y + w_\tau S_\tau + w_p S_p + w_o S_o - \lambda C_n - \mu L_n$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$S(r)$	Puntuación total de la ejecución.	0,82 sobre 1.
S_y	Calidad de salida final.	Respuesta correcta y bien citada.
S_τ	Calidad de trayectoria.	Usó tools esperadas, orden razonable y argumentos correctos.
S_p	Cumplimiento de permisos y gates.	Pidió revisión antes de publicar.
S_o	Salud operativa.	Sin retries innecesarios ni loops.
C_n	Coste normalizado.	Coste real dividido por coste máximo.
L_n	Latencia normalizada.	Latencia real dividida por latencia máxima.
w_y, w_τ, w_p, w_o	Pesos de calidad.	En soporte quizá pesa más S_y; en operaciones pesa más S_p.
λ, μ	Penalizaciones.	Penalizar coste y latencia.

La fórmula no pretende esconder juicio humano. Pretende hacerlo explícito. Si tu producto valora trazabilidad, dale peso a S_τ. Si el coste manda, sube λ. Si el riesgo operativo manda, ningún score debería saltarse S_p.

Fecha de corte del estado del arte

Fecha de corte: 10 de junio de 2026.
Fuentes consultadas ese día: documentación oficial de OpenAI sobre agent evals, trace grading y tracing del Agents SDK; documentación oficial de Google ADK sobre evaluación de agentes; documentación de LangChain/LangSmith sobre Agent Evals y evaluación de trayectorias; documentación de OpenTelemetry sobre trazas; documentación de Phoenix y Promptfoo sobre evals de agentes y agentes de código; benchmarks académicos como AgentBench, SWE-bench y ToolBench; referencias de ingeniería de ML sobre evaluación y deuda técnica.

Lo estable es el método: dataset, replay, traza, métricas, gates, comparación de versiones y análisis de regresiones. Lo cambiante son productos, nombres de métricas, dashboards, APIs de eval, modelos evaluadores, precios y benchmarks de moda.

Qué mirar: salida, trayectoria y operación

Google ADK lo formula de forma clara: en agentes no basta con evaluar la respuesta final; también hay que evaluar la trayectoria, es decir, la secuencia de pasos y tools usadas antes de responder.[2] La misma idea aparece en OpenAI: las trazas permiten evaluar llamadas de modelo, tool calls, guardrails y handoffs, y trace grading puntúa esas trazas con criterios estructurados.[3][4]

Podemos organizarlo así:

CAPA	PREGUNTA	MÉTRICA TÍPICA
Salida final	¿Respondió lo correcto?	`answer_score`, `json_valid`, `citation_match`, `artifact_valid`.
Trayectoria	¿Llegó por un camino aceptable?	`tool_order_score`, `arg_match`, `extra_tools`, `missing_steps`.
Permisos	¿Pidió revisión cuando tocaba?	`policy_gate_pass`, `approval_required_match`.
Coste	¿Compensa económicamente?	`cost_per_run`, `cost_per_accepted_task`, `token_budget_pass`.
Latencia	¿Es usable?	`p50`, `p95`, `timeout_rate`.
Robustez	¿Se recupera de fallos esperables?	`retry_success`, `fallback_correct`, `partial_task_handled`.
Trazabilidad	¿Podemos explicar qué pasó?	`trace_completeness`, `missing_span_rate`.

Si solo evalúas salida final, no verás que el agente usó cuatro tools cuando bastaba una. Si solo evalúas coste, no verás que dejó una cita falsa. Si solo evalúas trayectoria, no verás que el texto final no ayuda a nadie.

Evaluar solo la salida confunde dos cosas distintas: que el agente acierte y que acierte por buenas razones. Cruzar resultado y trayectoria deja ver los cuatro casos, y el peligroso es el tercero: acertar por el camino equivocado, porque parece éxito y no se repite.

Pirámide de pruebas para agentes

La forma más útil de pensar esto para ingeniería del software es una pirámide, pero no exactamente la pirámide clásica de unit tests. En agentes hay pruebas de contrato, de trayectoria y de operación.

CAPA	QUÉ PRUEBA	EJEMPLO	FRECUENCIA
Unit	Funciones puras del harness.	Normalizar una URL, calcular coste, validar JSON.	En cada cambio.
Contract	Que una tool respeta schema, permisos y errores esperados.	`search_source(query: str)` devuelve documentos con `url`, `title`, `snippet`.	En cada PR.
Component	Una pieza del agente aislada.	Router elige `biblioteca` y no `publicacion` en modo lectura.	En cada PR.
Trajectory	Secuencia de pasos contra una referencia o rúbrica.	Buscar fuente antes de validar APA.	En PR y nightly.
Scenario	Caso completo multi-turn con tools y estado.	Alumno aporta cita incompleta, agente pregunta, busca, valida y responde.	Nightly o prepublicación.
Regression	Fallos ya corregidos convertidos en casos permanentes.	La versión anterior omitía `validate_apa`.	Siempre.
Shadow	Tráfico real copiado a una versión candidata sin afectar al usuario.	Comparar `agent-v2` y `agent-v3` con el mismo input.	Antes de publicar.
Online sampled eval	Muestra de producción revisada automáticamente y, si hace falta, por personas.	2% de runs con trazas puntuadas.	Continuo.

OpenAI recomienda empezar por trazas cuando aún estás depurando comportamiento y pasar a datasets/eval runs cuando necesitas repetibilidad.[5] Promptfoo lo aterriza muy bien para agentes de código: un agente no transforma `X` en `Y` una sola vez, decide, actúa, observa y repite; por eso hay que evaluar sistema, no solo modelo.[6]

El problema del oráculo

En testing clásico, a veces sabemos exactamente la salida esperada. En agentes, muchas veces no. Dos respuestas pueden ser correctas con redacciones distintas; dos trayectorias pueden ser aceptables con orden diferente; una tool puede devolver datos equivalentes con otro ranking. A eso lo llamamos problema del oráculo: no siempre existe una respuesta única y fácil de comparar.

TIPO DE ORÁCULO	CUÁNDO SIRVE	RIESGO SI LO USAS MAL
Exact match	JSON, IDs, cálculos, rutas, permisos.	Castiga respuestas válidas con formato distinto.
Schema/property check	Salida estructurada, artefactos, contratos.	Puede pasar contenido pobre si el schema es débil.
Golden reference	Casos donde hay respuesta conocida.	Se queda corto para problemas abiertos.
Rúbrica humana	Calidad, utilidad, explicación, criterio profesional.	Cara y menos escalable.
LLM-as-judge	Escalar revisión cualitativa con criterios.	Necesita calibración, ejemplos y control de sesgo.
Pairwise	Comparar `baseline` contra `candidate`.	No dice si ambos son malos.
Metamorphic testing	Propiedades que deben mantenerse al cambiar el input.	Requiere pensar invariantes útiles.

Ejemplo de metamorphic testing: si pido “cita en APA” y luego “cita en APA en castellano”, el idioma puede cambiar, pero la URL, el año y el autor no deberían desaparecer. No busco una frase idéntica; busco una propiedad que debe conservarse.

El dataset: pequeño, vivo y con intención

Un dataset de evaluación de agentes no empieza siendo enorme. Empieza siendo representativo.

TIPO DE CASO	QUÉ CUBRE	EJEMPLO
Golden set	Casos que siempre deben pasar.	Buscar fuente, citar, validar y responder.
Regresión	Fallos ya observados.	Antes omitía revisar permisos al publicar.
Casos límite	Entradas raras pero posibles.	Tool devuelve respuesta vacía o incompleta.
Casos de coste	Peticiones que podrían disparar pasos.	Consulta amplia que invita a llamar varias tools.
Casos de permisos	Acciones con scope distinto.	Alumno puede leer, editor puede publicar.
Casos multi-turn	Conversaciones con información gradual.	Usuario aporta datos en dos turnos.
Casos de recuperación	Error recuperable de tool o timeout.	Primera ruta falla y debe usar fallback válido.

Cada caso debería guardar:

```
{
  "case_id": "f5c10-001",
  "input": "Comprueba esta cita y genera referencia APA.",
  "expected": {
    "final_must_contain": ["autor", "año", "URL"],
    "required_tools": ["search_source", "validate_apa"],
    "forbidden_tools": ["publish_page"],
    "max_cost_eur": 0.05,
    "max_latency_ms": 5000,
    "policy": "read_only"
  },
  "tags": ["referencias", "tool-use", "read-only"],
  "why_it_exists": "Evita publicar una cita sin fuente comprobada."
}
```

El campo `why_it_exists` es más importante de lo que parece. Cuando el dataset crece, ayuda a no borrar casos “raros” que en realidad protegen aprendizaje del equipo.

En una asignatura o proyecto profesional, el dataset debería vivir como código. No basta con una hoja suelta llamada `evals_final_v3.xlsx`.

```yaml
suite_id: f5-agentes-referencias
dataset_version: 2026-06-10.1
owner: equipo-ia
baseline_agent: agent-v2
candidate_agent: agent-v3
default_budget:
  max_cost_eur: 0.08
  max_latency_ms: 6000
cases:
  - case_id: f5c10-001
    tags: [referencias, tool-use, read-only]
    input_file: cases/f5c10-001/input.md
    expected_file: cases/f5c10-001/expected.json
    rubric_file: rubrics/reference_check.yaml
    min_scores:
      final: 0.85
      trajectory: 0.90
      trace: 0.95
    run:
      repeat: 3
      temperature: 0
      sandbox: read_only
```

Esto permite revisar cambios como cualquier otro cambio de software: diff, PR, revisión, historial y rollback. Si el dataset no se versiona, una mejora puede ser solo que hemos cambiado el examen.

Trazas: el material que se evalúa

OpenAI Agents SDK representa una traza como una operación completa de workflow, compuesta por spans; esos spans pueden envolver agentes, generaciones, function tools, guardrails y handoffs.[7] OpenTelemetry describe las trazas como el camino de una petición por una aplicación, formado por spans con nombre, tiempos, atributos, eventos, estado y relaciones padre-hijo.[8]

Para evaluar agentes, una traza mínima debería contener:

EVENTO	CAMPOS MÍNIMOS
`run.started`	`case_id`, `agent_version`, `model`, `prompt_version`, `dataset_version`.
`route.decision`	Ruta elegida, alternativas, motivo, score.
`model.call`	Modelo, tokens, latencia, input_hash, output_hash.
`tool.call`	Tool, argumentos, schema_version, efecto, timeout.
`tool.result`	Estado, resumen, bytes, filas, error recuperable si aplica.
`approval.request`	Acción, recurso, scope, score de riesgo.
`approval.result`	Decisión, input efectivo, persona o rol revisor.
`gate.result`	Gate, umbral, valor medido, pass/fail.
`run.completed`	Salida final, coste total, latencia total, estado final.

No hace falta guardar todo el texto en claro si hay datos sensibles. Puedes guardar hashes, resúmenes, IDs y muestras controladas. Pero si la traza no permite reconstruir la decisión, la evaluación será decorativa.

Para que un alumno de ingeniería lo implemente bien, conviene pensar en términos de OpenTelemetry:

CAMPO	QUÉ APORTA	ERROR TÍPICO
`trace_id`	Une toda la ejecución.	Generar uno distinto por cada tool y perder la historia completa.
`span_id`	Identifica una operación concreta.	Mezclar llamada de modelo, tool y gate en un mismo evento enorme.
`parent_span_id`	Reconstruye jerarquía.	No saber qué tool nació de qué decisión.
`name`	Nombra la operación.	Usar nombres genéricos como `call` o `step`.
`start_time`, `end_time`	Calcula latencia por tramo.	Medir solo latencia total.
`attributes`	Guarda versión, modelo, tool, coste, tokens, policy, dataset.	Esconder datos críticos dentro de texto libre.
`events`	Marca momentos dentro del span.	No distinguir “se pidió aprobación” de “se aprobó”.
`status`	Resultado de la operación.	No separar error recuperado de final correcto.
`links`	Relaciona trazas asíncronas.	Perder trabajos en cola o handoffs entre agentes.

OpenTelemetry recalca que los spans comparten `trace_id`, que `parent_id` permite construir jerarquía, que los exporters envían trazas a un backend y que la propagación de contexto permite correlacionar spans generados en servicios distintos.[9] En agentes esto es oro: si un router llama a un subagente y ese subagente llama a MCP, la evaluación debe poder seguir el hilo.

Métricas de trayectoria

Una trayectoria no es “buena” solo porque termine. Hay que mirar pasos y argumentos.

MÉTRICA	QUÉ MIDE	CUÁNDO USARLA
`tool_sequence_match`	Si las tools aparecen en el orden esperado.	Flujos con orden obligatorio.
`tool_set_match`	Si se usaron las tools esperadas, sin importar orden.	Recuperación de varias fuentes.
`required_tool_recall`	Proporción de tools obligatorias usadas.	Evitar omisiones críticas.
`extra_tool_rate`	Tools no esperadas por caso.	Controlar coste y ruido.
`argument_match`	Coincidencia de argumentos relevantes.	APIs, búsquedas, acciones con scope.
`observation_use`	Si la respuesta final usa observaciones reales.	RAG, navegador, bases de datos.
`stop_quality`	Si paró por condición correcta.	Evitar loops o cierres prematuros.

LangChain documenta evaluadores de trayectoria con modos como `strict`, `unordered`, `subset` y `superset`, además de evaluación con evaluador cuando la trayectoria correcta no es única.[10] La idea práctica es muy buena: no todos los casos necesitan el mismo tipo de comparación.

MODO	QUÉ EXIGE	EJEMPLO
`strict`	Mismo orden y mismas tools.	`lookup_policy` antes de `create_ticket`.
`unordered`	Mismas tools, orden libre.	Buscar normativa y calendario.
`subset`	No llamar tools fuera de la referencia.	Caso de solo lectura.
`superset`	Al menos llamar las tools obligatorias.	Puede consultar fuente extra si no publica.
Rúbrica	Evaluación cualitativa con criterios.	“La trayectoria usa evidencia antes de concluir”.

Evaluar argumentos, no solo nombres de tools

Un error muy común: “ha llamado a la tool correcta, entonces bien”. No necesariamente. La tool correcta con argumentos malos puede ser peor que no llamarla.

CASO	LLAMADA APARENTE	QUÉ HAY QUE EVALUAR
Búsqueda	`search_source(query="paper")`	Query demasiado genérica, fecha, idioma, dominio, número de resultados.
RAG	`retrieve(k=20)`	`k` , filtro, namespace, score mínimo, diversidad, documento usado en la respuesta.
Base de datos	`sql_query("SELECT *")`	Proyección, filtros, límites, coste, permisos, explain plan.
Código	`run_tests(command="npm test")`	Directorio, timeout, salida, cobertura del test ejecutado.
Escritura	`create_ticket(...)`	Campos obligatorios, idempotencia, recurso, owner y efecto persistente.

Para agentes, un contrato de tool debería tener cuatro capas:

CAPA	QUÉ DECLARA
Schema	Tipos, campos obligatorios, enums y límites.
Semántica	Qué significa cada campo y qué invariantes debe respetar.
Efecto	Si lee, escribe, ejecuta, llama red, modifica estado o requiere aprobación.
Observabilidad	Qué span, atributos y eventos debe emitir.

Esto conecta directamente con el capítulo 03: function calling no es solo “pasar JSON”. Es diseñar contratos que luego se puedan evaluar.

Calibrar evaluadores y rúbricas

Un evaluador automático puede ayudar, pero no debería entrar en producción sin control. Phoenix documenta evaluadores con salida estructurada mediante tool calling: el evaluador no devuelve texto libre, sino una etiqueta y una explicación parseables.[11] Esa idea es muy importante: si el evaluador también improvisa formato, la evaluación se vuelve frágil.

CONTROL	CÓMO SE HACE	QUÉ EVITA
Calibration set	30-100 ejemplos puntuados por personas.	Evaluador demasiado generoso o demasiado duro.
Rubric anchors	Ejemplos de 0, 0.5 y 1 para cada criterio.	Escalas ambiguas.
Agreement (kappa de Cohen)	Comparar el evaluador con el criterio humano midiendo el acuerdo corregido por azar.[1][2]	Confiar en un evaluador que no replica el estándar.
Explanation required	Pedir motivo breve y estructurado.	Scores sin diagnóstico.
Blind comparison	Ocultar qué versión es baseline o candidate.	Preferencias por nombre de modelo o versión.
Drift check	Repetir calibración al cambiar evaluador o modelo.	Que el evaluador cambie sin que nos demos cuenta.

No todos los criterios necesitan evaluador. Si puedes validar con parser, test, schema, diff o cálculo, hazlo. El evaluador queda para lo semántico: utilidad, coherencia, suficiencia de evidencia o claridad.

Gates: pasar o no pasar

Un gate es una condición de paso. En agentes, conviene tener gates antes de publicar cambios, antes de activar una versión y durante ejecución.

Un gate de release es una conjunción de criterios de aceptación, una puerta de calidad clásica: el producto de indicadores solo vale 1 si todos se cumplen a la vez:

$$G = \mathbf{1}[S_y \geq \theta_y] \cdot \mathbf{1}[S_\tau \geq \theta_\tau] \cdot \mathbf{1}[C \leq C_{\max}] \cdot \mathbf{1}[L_{95} \leq L_{\max}] \cdot \mathbf{1}[P = 1]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
G	Resultado del gate.	1 pasa, 0 no pasa.
S_y	Score de salida final.	Al menos 0,85.
S_τ	Score de trayectoria.	Al menos 0,90 en tools obligatorias.
C	Coste medio o p95.	Menor o igual a 0,08 EUR por run.
L_{95}	Latencia p95.	Menor o igual a 8 segundos.
P	Cumplimiento de permisos.	1 si no hubo violación de policy.
θ_y, θ_τ	Umbrales mínimos.	Decididos por el producto.

Si cualquier factor vale 0, el gate no pasa. Esto parece duro, pero es sano: no queremos compensar una violación de permisos con una respuesta bonita.

Coste por tarea aceptada

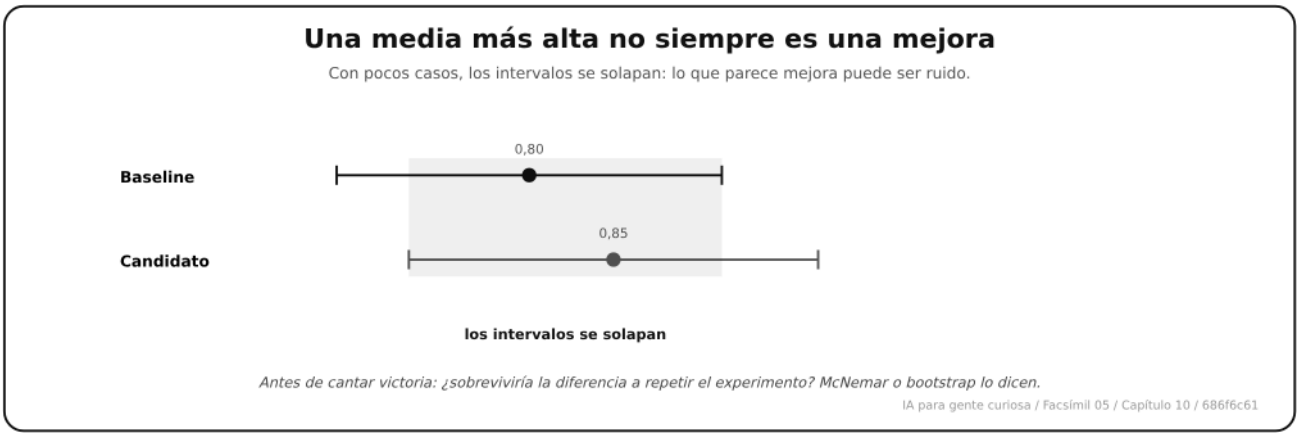
El coste más honesto no es el coste por llamada, sino el **coste por tarea aceptada** (cost per successful task): el coste total, incluida la revisión humana, dividido entre las ejecuciones que pasan el gate.

$$CPA = \frac{\sum_{i=1}^{N} c_i + h_i}{\sum_{i=1}^{N} \mathbf{1}[G_i = 1]}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
CPA	Coste por tarea aceptada.	0,19 EUR por caso que pasa.
N	Número de ejecuciones.	200 runs.
c_i	Coste técnico de la ejecución i.	Modelo, tools, infraestructura.
h_i	Coste de revisión humana o corrección.	Minutos convertidos a euros.
G_i	Gate de la ejecución i.	1 si pasa, 0 si no.

Un agente barato que falla mucho puede salir caro. Si 100 ejecuciones cuestan 5 EUR pero solo 20 pasan, el coste por aceptada es 0,25 EUR. Si otro sistema cuesta 8 EUR y pasan 80, el coste por aceptada baja a 0,10 EUR.

Incertidumbre: no confundas mejora con ruido



Los agentes son sistemas no deterministas. Aunque fijes temperatura, cambian dependencias, tools, latencia, documentos recuperados y, a veces, pequeñas decisiones internas. Por eso una suite de evaluación necesita repetición e intervalo, no solo un porcentaje bonito.

La tasa de paso básica es:

$$\hat{p} = \frac{k}{n}$$

SÍMBOLO	SIGNIFICADO
k	Runs que pasan el gate.
n	Runs totales.
$\hat{p}$	Estimación de tasa de paso.

Pero $\hat{p} = 0{,}90$ no significa lo mismo con 10 casos que con 1.000. Para una estimación más honesta se puede usar un intervalo de Wilson:

$$IC = \frac{\hat{p} + \frac{z^2}{2n} \pm z \sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2}{4n^2}}}{1 + \frac{z^2}{n}}$$

No hace falta memorizar la fórmula. La idea práctica es esta: si la versión nueva pasa 19 de 20 y la antigua 18 de 20, quizá no has descubierto una mejora; quizá solo has visto variación. Si la nueva pasa 860 de 1.000 y la antigua 780 de 1.000, ya tienes más señal.

Para comparar versiones, mira la diferencia de puntuación entre la candidata y la baseline, y decide con tolerancias que separen una mejora real del ruido:

$$\Delta S = S_{candidate} - S_{baseline}$$

Y decide tolerancias:

GATE DE COMPARACIÓN	EJEMPLO
Calidad	$\Delta S_y \geq -0{,}01$: no acepto perder más de 1 punto.
Trayectoria	$\Delta S_\tau \geq 0$: no acepto empeorar tool use.
Coste	$\Delta C_{p95} \leq 0{,}02$: no acepto subir más de 2 céntimos en p95.
Latencia	$\Delta L_{p95} \leq 500ms$: no acepto medio segundo extra sin mejora.
Estabilidad	`flake_rate <= 0.03`: no acepto más de 3% de casos inestables.

Esto enseña una lección clave: evaluar agentes se parece más a hacer ingeniería experimental que a corregir un examen de opción única.

Benchmarks y por qué no bastan

Los benchmarks públicos son útiles para orientarse, pero no sustituyen tu dataset. AgentBench evalúa LLMs como agentes en varios entornos interactivos y muestra que actuar en entornos largos exige razonamiento, decisión y seguimiento de instrucciones más allá de responder texto.[13] ToolLLM/ToolBench se centra en uso de APIs reales y construcción de datos para evaluar llamadas a herramientas.[14] SWE-bench convirtió issues reales de GitHub en tareas de edición de repositorios, mostrando que resolver trabajo de software real exige entender contexto largo, modificar varios archivos y pasar tests.[15]

La lección: usa benchmarks para comparar familias de modelos o enfoques, pero decide con tu tráfico, tus tools, tus permisos y tus costes.

BENCHMARK	QUÉ ENSEÑA	QUÉ NO DECIDE POR TI
AgentBench	Agentes en entornos interactivos.	Tu policy, coste y UX.
ToolBench	Uso de APIs y selección de tools.	Tus schemas, permisos y datos.
SWE-bench	Trabajo de código con repos reales.	Tu producto si no es software engineering.
Evals propias	Tu flujo, tus rutas y tus gates.	Comparación global con el mercado.

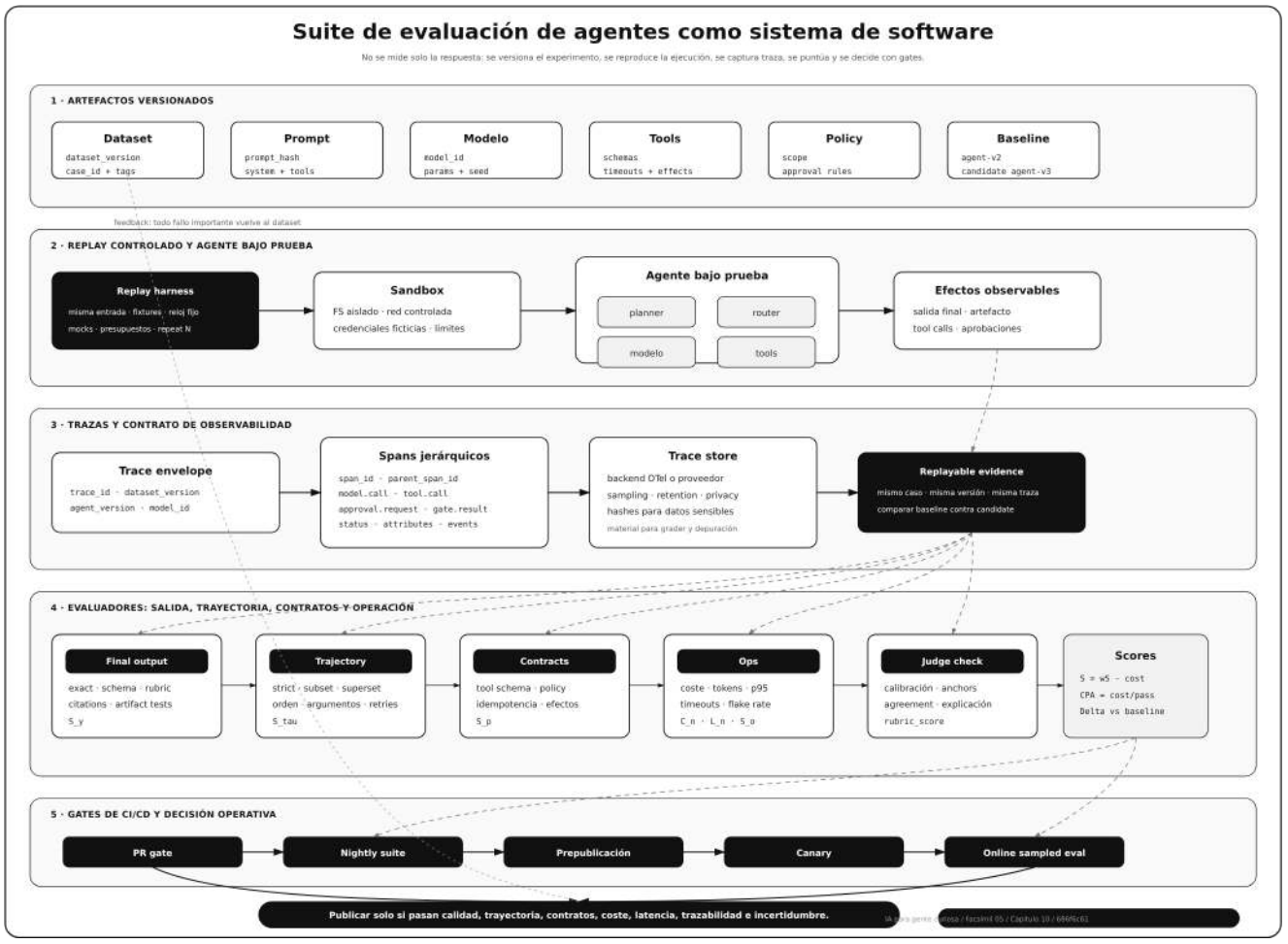
Herramientas que conviene conocer

No hay una única herramienta definitiva. Para un alumno, lo valioso es entender qué pieza del sistema cubre cada una.

HERRAMIENTA O ENFOQUE	QUÉ APORTA	QUÉ NO SUSTITUYE
OpenAI Evals y trace grading	Datasets, graders, runs y evaluación de trazas de workflows.	Tu diseño de casos y rúbricas.
Google ADK Evaluate	Tests de sesión, trayectoria, tool use y respuesta final en agentes ADK.	Observabilidad general si tu sistema vive fuera de ADK.
LangChain/LangSmith evals	Comparación de trayectorias, datasets y seguimiento de runs.	Decisiones de producto sobre coste y riesgo.
Phoenix	Trazas, evaluación con salidas estructuradas y análisis de comportamiento.	Versionado completo del dataset si no lo diseñas.
Promptfoo	Evals configurables, assertions y casos para agentes de código.	Arquitectura de permisos del agente.
OpenTelemetry	Modelo estándar de trazas, spans, exporters y propagación de contexto.	Rúbricas semánticas o criterios de negocio.
`pytest` + scripts propios	Control total, CI sencillo y aprendizaje profundo.	Dashboards y colaboración si el equipo crece.

Mi recomendación didáctica: empezar con scripts propios para entender las piezas, luego conectar una herramienta de trazas/evals cuando el volumen haga incómodo revisar a mano.

Anatomía visual de una suite de evaluación



La figura ya no enseña una cadena bonita, sino una arquitectura: artefactos versionados, replay aislado, agente bajo prueba, trazas, evaluadores separados, estadística, gates y bucle de regresión. Si mezclamos todo en una nota única, no sabremos si falló la respuesta, la tool, el permiso, el coste, la ruta, la observabilidad o la comparación contra baseline.

Cómo diseñaría gates por entorno

No todos los gates viven en el mismo sitio. Un gate de PR debe ser rápido; uno nocturno puede ser más caro; uno de canary mira tráfico real; uno de runtime protege la ejecución concreta.

MOMENTO	ENTRADA	GATE	UMBRAL TÍPICO	QUÉ BLOQUEA
Desarrollo	Unit y contract tests.	`contract_gate`.	100% schemas y tools críticas.	Cambios que rompen contratos.
Pull request	Golden set corto.	`pr_eval_gate`.	Sin regresiones P0/P1.	Prompts o routers que rompen casos básicos.
Nightly	Suite completa repetida.	`stability_gate`.	`flake_rate <= 3%`.	Versiones inestables o dependientes del azar.
Prepublicación	Baseline contra candidate.	`release_gate`.	Mejora o empate dentro de tolerancia.	Versiones más caras, lentas o peores.
Canary	Muestra pequeña de tráfico real.	`canary_gate`.	p95, coste y fallos dentro de SLO.	Despliegue amplio si sube el fallo.
Runtime	Cada ejecución.	`policy_budget_gate`.	Scope y presupuesto válidos.	Acciones fuera de permiso o presupuesto.
Revisión humana	Acciones persistentes.	`approval_gate`.	Decisión explícita y trazable.	Efectos persistentes sin revisión.

La cultura sana no es “bloquear por bloquear”. Es saber qué evidencia falta para avanzar. En un equipo maduro, cada gate tiene dueño, umbral, razón, caducidad y plan de actuación cuando falla.

El acierto frágil: la historia de un éxito que no se repitió

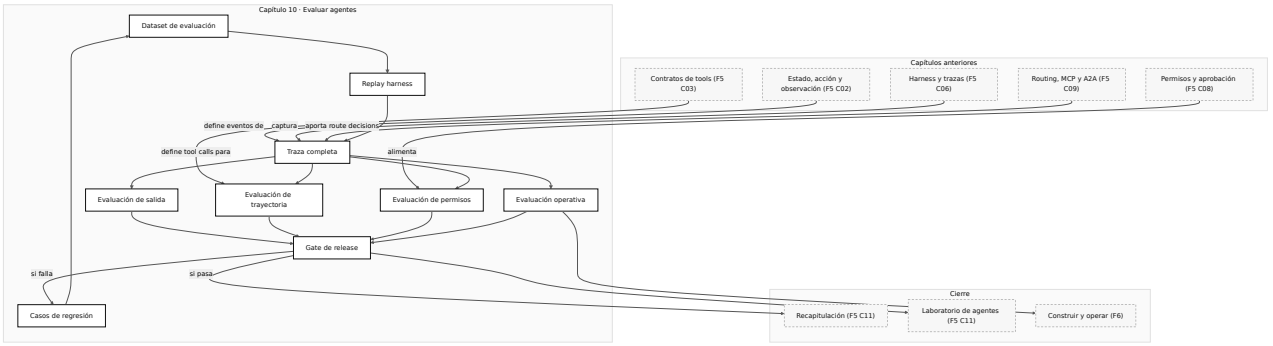
Conviene contar con detalle el caso del cuadrante peligroso, ese en el que el resultado es correcto pero la trayectoria es sucia, porque es el error de evaluación que más caro se paga y el que las métricas de salida no detectan. Un equipo construye un agente de soporte que, ante la pregunta «¿cuántos pagos pendientes tiene este alumno por campus?», responde «Norte: 2, Centro: 1». La respuesta es correcta. En la evaluación, que solo miraba la salida final, el caso cuenta como acierto, el porcentaje sube y el sistema parece listo. Lo lanzan.

En producción, el mismo agente empieza a fallar en preguntas casi idénticas, y nadie entiende por qué, porque «en las pruebas iba bien». La autopsia, que solo es posible porque alguien guardó las trazas, revela la verdad incómoda: el día de la prueba, el agente no consultó la base de datos. Encontró, en un fragmento de contexto que arrastraba de una conversación anterior, la frase «Norte: 2, Centro: 1», y la repitió. Acertó, sí, pero por el camino equivocado: no porque supiera consultar pagos, sino porque la respuesta estaba, por casualidad, flotando en su contexto. Esa habilidad no existe; lo que existía era una coincidencia. Y las coincidencias no se repiten.

Este es el corazón de por qué un agente se evalúa por trayectoria y no solo por resultado. Una respuesta correcta puede llegar por una buena razón (recuperó la evidencia, la verificó, la citó) o por una mala (la adivinó, la copió de un contexto contaminado, tuvo suerte con el ejemplo). Desde fuera, las dos respuestas son idénticas; solo la trayectoria las distingue. Y la diferencia no es académica: la respuesta sólida generaliza a casos nuevos, mientras que el acierto frágil se desmorona en cuanto cambia el ejemplo, justo cuando ya está en producción y el coste de descubrirlo es máximo. Por eso la pregunta de evaluación correcta no es «¿acertó?», sino «¿acertó por una razón que volverá a funcionar?».

De aquí sale la disciplina práctica que distingue una evaluación de agentes seria de una que solo cuenta aciertos. Hay que mirar, en la traza, si las tools que debían usarse se usaron de verdad y con los argumentos correctos, no solo si el nombre de la tool aparece. Hay que incluir en el dataset casos donde la respuesta correcta no esté en el contexto, para que copiar no baste y haga falta recuperar de verdad. Y hay que medir groundedness aparte de la corrección, porque una respuesta puede ser cierta y aun así no estar respaldada por la evidencia que el agente recuperó. Sin esas comprobaciones, una suite de evaluación premia al agente con suerte tanto como al agente competente, y ese sesgo se paga entero el día del despliegue, cuando la suerte se acaba.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN ÚTIL
Run	Ejecución concreta de un caso por una versión de agente.
Trace grading	Evaluación estructurada de una traza completa.
Golden set	Casos pequeños y estables que protegen lo esencial.
Regresión	Algo que antes pasaba y ahora falla.
Trajectory match	Comparación entre pasos reales y pasos esperados.
Rúbrica	Lista de criterios observables para puntuar una respuesta o trayectoria.
Gate	Condición que permite o detiene una versión, acción o ejecución.
Coste por tarea aceptada	Coste total dividido por runs que pasan criterios.
p95	Percentil 95: valor que deja por debajo al 95% de ejecuciones.
Replay harness	Sistema que reproduce casos con versiones controladas.
Dataset version	Identificador del conjunto de casos usado en una evaluación.
Trace completeness	Grado en que la traza contiene los eventos necesarios para explicar la run.
Oracle problem	Dificultad de saber cuál es la salida correcta cuando hay varias respuestas válidas.
Flake rate	Proporción de casos que pasan unas veces y fallan otras sin cambio de código.
Baseline	Versión de referencia contra la que comparas una candidata.
Candidate	Versión nueva que quieres aceptar o descartar.
Calibration set	Casos puntuados por personas para comprobar si un evaluador automático se comporta bien.
Metamorphic testing	Pruebas basadas en propiedades que deben mantenerse al transformar la entrada.

Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Evaluar solo la respuesta final	Es lo más rápido de leer.	Puntuar salida, trayectoria, permisos, coste y latencia.
No guardar dataset versionado	Los casos cambian y ya no comparas lo mismo.	Versionar dataset, prompt, modelo, tools y policy.
Convertir el evaluador en verdad absoluta	Una nota generada parece objetiva.	Usar rúbrica, calibración y casos con respuesta conocida.
Ignorar coste humano	El modelo parece barato pero exige mucha corrección.	Medir coste por tarea aceptada, incluyendo revisión.
No mirar argumentos de tools	La tool correcta puede llamarse con parámetros malos.	Evaluar tool, orden y argumentos relevantes.
Usar un benchmark como decisión final	Da sensación de rigor externo.	Combinar benchmark público con eval propia del producto.
No meter fallos corregidos en regresión	Se repiten problemas viejos.	Cada fallo importante crea un caso nuevo.
No repetir runs	Una ejecución aislada parece concluyente.	Medir `repeat`, `flake_rate` e intervalos.
No separar contrato de semántica	El JSON válido parece suficiente.	Validar schema, significado, efectos y observabilidad.
No definir baseline	La versión nueva se evalúa en el vacío.	Comparar siempre contra una referencia estable.

Antes de pasar página

Antes del cierre del facsímil, deberías poder responder:

PREGUNTA	SI DUDAS, VUELVE A...
¿Por qué un agente puede acertar por el camino equivocado?	`Un agente puede acertar por el camino equivocado`.
¿Qué contiene una run evaluable?	`La definición útil`.
¿Qué diferencia hay entre salida final, trayectoria y operación?	`Qué mirar: salida, trayectoria y operación`.
¿Cómo se parece una suite de agentes a una suite de ingeniería del software?	`Pirámide de pruebas para agentes`.
¿Qué haces cuando no hay una salida única esperada?	`El problema del oráculo`.
¿Qué casos debería tener un dataset inicial?	`El dataset: pequeño, vivo y con intención`.
¿Qué eventos mínimos necesita una traza?	`Trazas: el material que se evalúa`.
¿Qué métricas usarías para evaluar tool calls?	`Métricas de trayectoria`.
¿Por qué hay que mirar argumentos de tools?	`Evaluar argumentos, no solo nombres de tools`.
¿Cómo controlas que un evaluador automático no sea una caja negra nueva?	`Calibrar evaluadores y rúbricas`.
¿Por qué un gate no debería compensar permisos con buena respuesta?	`Gates: pasar o no pasar`.
¿Cómo calculas coste por tarea aceptada?	`Coste por tarea aceptada`.
¿Cómo evitas confundir una mejora real con variación?	`Incertidumbre: no confundas mejora con ruido`.
¿Qué aportan benchmarks como AgentBench o SWE-bench?	`Benchmarks y por qué no bastan`.
¿Qué herramienta usarías según la pieza que quieras evaluar?	`Herramientas que conviene conocer`.
¿Cómo construirías un evaluador mínimo sin depender de proveedor?	Practícalo en el cuaderno del facsímil.

Para saber más

- Amershi, S. y otros (2019). *Software Engineering for Machine Learning: A Case Study*. https://doi.org/10.1109/ICSE-SEIP.2019.00042
- Arize Phoenix. (2026). *LLM Evals*. https://arize.com/docs/phoenix/evaluation/llm-evals
- Baylor, D. y otros (2017). *TFX: A TensorFlow-Based Production-Scale Machine Learning Platform*. https://doi.org/10.1145/3097983.3098021
- Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1), 37-46. https://doi.org/10.1177/001316446002000104
- Google. (2026). *Agent Development Kit: Why Evaluate Agents*. https://adk.dev/evaluate/
- Jimenez, C. E. y otros (2024). *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?*. https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html
- Keeney, R. L., & Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley.
- LangChain. (2026). *Agent Evals*. https://docs.langchain.com/oss/python/langchain/test/evals
- Liu, X. y otros (2024). *AgentBench: Evaluating LLMs as Agents*. https://doi.org/10.48550/arXiv.2308.03688
- OpenAI. (2026). *Agents SDK: Tracing*. https://openai.github.io/openai-agents-python/tracing/
- OpenAI. (2026). *Evaluate agent workflows*. https://developers.openai.com/api/docs/guides/agent-evals
- OpenAI. (2026). *Trace grading*. https://developers.openai.com/api/docs/guides/trace-grading
- OpenTelemetry. (2026). *Traces*. https://opentelemetry.io/docs/concepts/signals/traces/
- Promptfoo. (2026). *Evaluate Coding Agents*. https://www.promptfoo.dev/docs/guides/evaluate-coding-agents/
- Qin, Y. y otros (2023). *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-World APIs*. https://doi.org/10.48550/arXiv.2307.16789

En resumen

IDEA	QUÉ TE LLEVAS
Un agente se evalúa por más que su respuesta.	Hay que mirar salida final, trayectoria, permisos, coste, latencia y trazabilidad.
Una suite de evaluación es software.	Dataset, prompt, modelo, tools, policy y baseline deben estar versionados.
Las trazas son el material de evaluación.	Sin eventos estructurados no puedes saber dónde falló ni comparar versiones.
Los gates convierten métricas en decisión.	Una versión pasa solo si cumple calidad, trayectoria, coste y policy.
El oráculo no siempre es exacto.	Combina exact match, schemas, propiedades, rúbricas, evaluadores calibrados y revisión humana.
El coste real se mide por tarea aceptada.	Un sistema barato por llamada puede salir caro si falla mucho o exige corrección.
La estadística importa.	Repite runs, mide inestabilidad y compara candidate contra baseline con tolerancias.
Cada fallo importante alimenta el dataset.	La evaluación mejora cuando las regresiones se convierten en casos permanentes.

Notas

1. Keeney, R. L. y Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley. Formaliza la función de valor aditiva ponderada. ↩

2. Google. (2026). *Agent Development Kit: Why Evaluate Agents*. https://adk.dev/evaluate/. Consultado el 10 de junio de 2026. ↩

3. OpenAI. (2026). *Evaluate agent workflows*. https://developers.openai.com/api/docs/guides/agent-evals. Consultado el 10 de junio de 2026. ↩

4. OpenAI. (2026). *Trace grading*. https://developers.openai.com/api/docs/guides/trace-grading. Consultado el 10 de junio de 2026. ↩

5. OpenAI. (2026). *Evaluate agent workflows*. https://developers.openai.com/api/docs/guides/agent-evals. Consultado el 10 de junio de 2026. ↩

6. Promptfoo. (2026). *Evaluate Coding Agents*. https://www.promptfoo.dev/docs/guides/evaluate-coding-agents/. Consultado el 10 de junio de 2026. ↩

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

8. OpenTelemetry. (2026). *Traces*. https://opentelemetry.io/docs/concepts/signals/traces/. Consultado el 10 de junio de 2026. ↩

9. OpenTelemetry. (2026). *Traces*. https://opentelemetry.io/docs/concepts/signals/traces/. Consultado el 10 de junio de 2026. ↩

.0. LangChain. (2026). *Agent Evals*. https://docs.langchain.com/oss/python/langchain/test/evals. Consultado el 10 de junio de 2026. ↩

.1. Arize Phoenix. (2026). *LLM Evals*. https://arize.com/docs/phoenix/evaluation/llm-evals. Consultado el 10 de junio de 2026. ↩

.2. Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1), 37-46. https://doi.org/10.1177/001316446002000104 Define el coeficiente kappa, que mide el acuerdo entre evaluadores corrigiendo las coincidencias por azar. ↩

.3. Liu, X. y otros (2024). *AgentBench: Evaluating LLMs as Agents*. International Conference on Learning Representations. https://doi.org/10.48550/arXiv.2308.03688. Consultado el 10 de junio de 2026. ↩

.4. Qin, Y. y otros (2023). *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-World APIs*. https://doi.org/10.48550/arXiv.2307.16789. Consultado el 10 de junio de 2026. ↩

.5. Jimenez, C. E. y otros (2024). *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?*. International Conference on Learning Representations. https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html. Consultado el 10 de junio de 2026. ↩