Facsímil 05

Agentes y orquestación

Capítulo 06

Harness engineering: límites, sensores y trazas

686f6c61 · 2026

Capítulo 06: Harness engineering: límites, sensores y trazas

El arnés que convierte una demo en sistema

En el capítulo anterior elegimos arquitecturas: ReAct, PEV, multiagente, blackboard, dry-run, memoria, meta-controlador. Ahora toca una pregunta menos vistosa y mucho más profesional: **¿qué rodea a esa arquitectura para que no dependa de la buena suerte?**

Un agente puede sonar brillante durante una demo y ser imposible de depurar cuando falla. Puede tocar demasiados archivos, repetir una tool sin información nueva, gastar más de lo previsto, declarar que terminó sin haber probado nada o perder el hilo entre sesiones. El problema no siempre es el modelo. Muchas veces falta el arnés técnico: el conjunto de límites, sensores, trazas, gates y estado operativo que convierte una ejecución en algo revisable.

Martin Fowler usa la expresión *harness engineering* para hablar del entorno que permite usar agentes de código con más control: instrucciones, contexto del repositorio, pruebas, evidencia, límites y revisión.[1] Aquí ampliamos la idea a cualquier agente con tools: código, RAG, datos, navegador, documentos, soporte, operaciones o investigación.

Qué no es un harness

Un harness no es un prompt más largo. Puedes escribir instrucciones excelentes y aun así no tener control sobre coste, permisos, estado, herramientas o verificación.

Tampoco es un dashboard al final. Ver una gráfica después de que algo salió mal ayuda, pero el harness empieza antes: decide qué acciones están permitidas, qué datos entran, qué presupuesto hay, qué debe registrarse y qué condición permite avanzar.

Y no es solo logging. Un log puede ser una pared de texto. Una traza útil tiene estructura: quién hizo qué, con qué entrada, en qué paso, cuánto tardó, cuánto costó, qué observó, qué cambió y por qué se detuvo.

La definición útil

Para este facsímil, un harness es:

> El arnés técnico que envuelve a un agente para limitar lo que puede hacer, medir lo que ocurre, registrar evidencia y decidir si una ejecución puede avanzar.

Anthropic distingue entre workflows predefinidos y agents que deciden dinámicamente su proceso y uso de herramientas.[2] Cuanto más dinámica sea esa decisión, más necesario se vuelve el harness. Si el camino cambia durante la ejecución, necesitamos sensores para verlo y límites para acotarlo.

Un harness tiene un conjunto fijo de componentes. Eso no es una ecuación de la literatura, es una lista de ingeniería, así que lo presentamos como tabla:

COMPONENTE	QUÉ FIJA	EJEMPLO
Objetivo G	La meta verificable.	«Revisar citas y proponer cambios antes de publicar».
Instrucciones I	Reglas estables.	`AGENTS.md`, reglas editoriales, comandos permitidos.
Estado S	Lo operativo.	Paso actual, evidencias, decisiones, bloqueos, coste.
Acciones A	Lo que puede hacer.	Buscar referencia, validar APA, proponer diff, pedir revisión.
Políticas P	Permisos.	Qué tools puede usar, cuándo necesita aprobación.
Presupuesto B	Límites.	Pasos, tokens, tools, tiempo, coste, rutas tocables.
Verificadores V	Gates.	Tests, rúbricas, validadores JSON, revisión de diff.
Recuperación R	Plan B.	Retry, fallback, rollback, handoff, parada controlada.
Traza τ	Observabilidad.	Eventos estructurados de toda la ejecución.

El presupuesto, en cada paso, es un vector de límites que se va consumiendo:

DIMENSIÓN	QUÉ LIMITA	EJEMPLO RESTANTE
Pasos n_t	Iteraciones del bucle.	4 pasos.
Llamadas q_t	Tools restantes.	2 consultas externas.
Tokens k_t	Contexto restante.	18.000 tokens.
Coste c_t	Dinero restante.	0,42 EUR.
Cambio Δ_t	Efecto máximo.	3 archivos, 120 líneas o solo lectura.

Cuando cualquiera de esas dimensiones llega a cero, el bucle para: el presupuesto es una condición de parada, no un adorno.

El **gate** que decide si una ejecución se acepta es una conjunción de criterios de aceptación, una puerta de calidad clásica. Solo pasa si se cumplen todos a la vez:

$$\text{go} = O_{\text{ok}} \wedge T_{\text{ok}} \wedge (C \leq C_{\max}) \wedge P_{\text{ok}} \wedge R_{\text{def}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
O_{ok}	Resultado final válido.	La respuesta cumple el objetivo.
T_{ok}	Trayectoria válida.	Usó tools permitidas y dejó evidencia.
$C \leq C_{\max}$	Coste dentro del límite.	0,75 EUR por tarea.
P_{ok}	Permisos respetados.	No ejecutó fuera de alcance.
R_{def}	Recuperación definida.	Hay rollback, retry o handoff.

En palabras: una ejecución se acepta solo si el resultado es válido, la trayectoria fue limpia, el coste cabe, los permisos se respetaron y hay plan de recuperación. Basta que falle uno para no aceptarla.

La métrica económica que decide producción no es el precio por token, sino el **coste por tarea aceptada** (cost per successful task): el coste total dividido entre las ejecuciones que superan el gate.

$$C_{\text{aceptada}} = \frac{\sum_i C_i}{N_{\text{aceptadas}}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO CONCRETO
C_{aceptada}	Coste por tarea que pasa el gate.	1,40 EUR por revisión aceptada.
$\sum_i C_i$	Coste total de todas las ejecuciones.	Modelo, tools, infraestructura y revisión.
$N_{\text{aceptadas}}$	Ejecuciones que superan el gate.	73 de 100 tareas.

En palabras: un sistema barato por intento puede ser caro de verdad si acepta pocas ejecuciones. El precio por token engaña; el coste por tarea aceptada no, porque reparte entre los intentos el coste de los reintentos, la revisión y las correcciones.

Fecha de corte de herramientas

Fecha de corte: 10 de junio de 2026.
Fuentes consultadas: artículo de Martin Fowler sobre harness engineering, documentación de OpenAI Agents SDK Tracing, OpenTelemetry Tracing API, W3C Trace Context, documentación de pruebas de Anthropic y NIST AI RMF.

Lo estable es el mecanismo: estado operativo, límites, tools pequeñas, trazas estructuradas, evaluaciones repetibles, gates y handoff. Lo cambiante son SDKs, nombres de herramientas, formatos de traza, integraciones de observabilidad, precios y límites de cada proveedor.

Las capas del harness

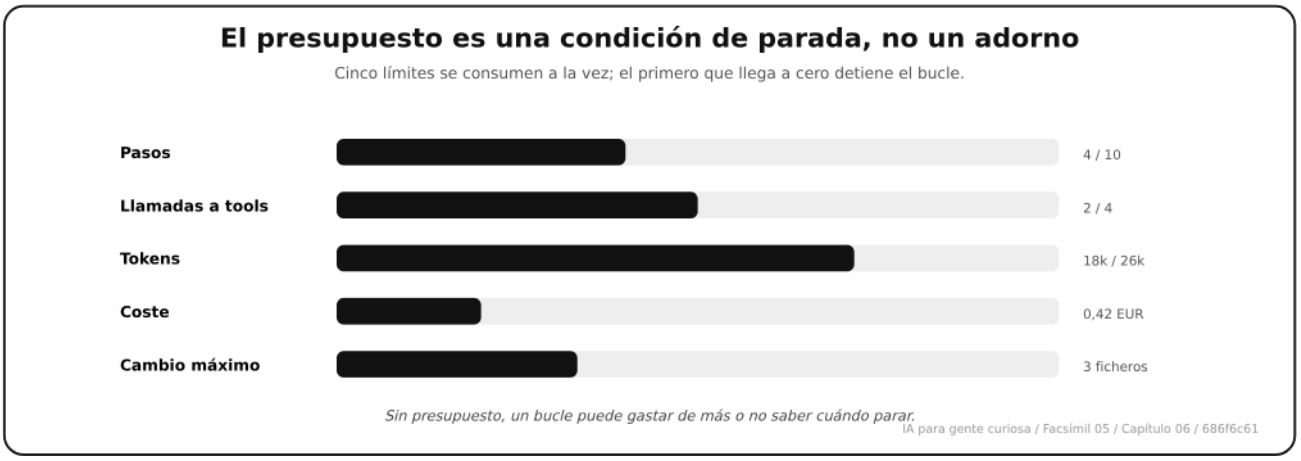
Un harness serio separa capas. Si todo vive en el prompt, no sabes qué cambiar cuando algo falla.

CAPA	PREGUNTA	ARTEFACTO	FALLO TÍPICO SI FALTA
Objetivo	¿Qué resultado cuenta como terminado?	`goal`, `done_when`, criterios de aceptación.	El agente declara éxito con una respuesta bonita.
Instrucciones	¿Cómo se trabaja aquí?	`AGENTS.md`, guía de repo, reglas editoriales.	Cada ejecución interpreta el proyecto de forma distinta.
Estado	¿Qué sabemos ahora?	`run_state.json`, task board, decisiones.	Se repiten pasos o se pierden bloqueos.
Scope	¿Qué puede tocar?	Rutas, dominios, tools, permisos, no-objetivos.	Cambios fuera de alcance o consultas innecesarias.
Sensores	¿Qué señales vuelven del mundo?	Tests, logs, diffs, métricas, resultados de tools.	El agente actúa sin feedback verificable.
Presupuesto	¿Cuánto puede gastar?	Máximo de pasos, tools, tokens, tiempo y coste.	Bucle largo, coste sorpresa o latencia inaceptable.
Gate	¿Puede avanzar?	Rúbrica, tests, umbrales, revisión humana.	El constructor se aprueba a sí mismo.
Handoff	¿Quién puede continuar?	Resumen estructurado, pendientes, evidencia, siguiente paso.	La siguiente sesión empieza desde cero.

OpenAI Agents SDK documenta tracing para registrar ejecuciones de agentes y entender qué ocurrió entre modelo, tools y handoffs.[3] OpenTelemetry, por su parte, define trazas y spans como unidades observables de trabajo con atributos y contexto.[4] El lenguaje cambia entre herramientas; la idea no: una ejecución se entiende por sus eventos.

Anatomía visual de un harness de agentes

El diagrama muestra la idea central con una lectura más de ingeniería: el agente no es una caja en medio del sistema. Hay un plano de control antes de actuar, un plano de ejecución con fronteras explícitas, un store de estado para checkpoints y replay, un gateway para tools, una tubería de observabilidad y un gate que decide con evidencia.



Sensores: cómo ve el sistema lo que hizo

En un agente, un sensor no tiene por qué ser físico. Un test fallido es un sensor. Un diff demasiado grande es un sensor. Un timeout, una captura de pantalla, un código de error, una cita encontrada, una métrica de latencia o una tool que devuelve `not_found` son sensores.

SENSOR	QUÉ MIDE	EJEMPLO
Test	Comportamiento verificable.	`pytest tests/test_citas.py` pasa o falla.
Diff	Alcance del cambio.	2 archivos, 48 líneas, ninguna ruta prohibida.
Tool result	Observación externa.	URL consultada, estado HTTP, campos devueltos.
Métrica	Coste, latencia, calidad o cobertura.	7 pasos, 5 tool calls, 1,2 s p95.
Captura	Estado visual.	Página renderizada sin solapes.
Validador	Forma de salida.	JSON válido, schema completo, campos permitidos.
Revisión	Juicio estructurado.	Rúbrica con criterios y evidencia.

Anthropic recomienda desarrollar tests para aplicaciones con LLM porque la calidad no puede depender de lectura manual ocasional.[5] En agentes, esa idea se amplía: no solo probamos la respuesta final; probamos la trayectoria.

Límites que sí importan

Los límites buenos no están para molestar al agente. Están para hacer explícito el contrato de trabajo.

LÍMITE	QUÉ EVITA	EJEMPLO OPERATIVO
Pasos máximos	Bucle sin progreso.	`max_steps = 8`.
Llamadas a tools	Consultas innecesarias.	`max_tool_calls = 5`.
Tiempo	Esperas inaceptables.	`timeout_s = 30`.
Coste	Sorpresas de factura.	`max_cost = 0.50`.
Tokens/contexto	Prompts enormes y ruido.	Resumir estado cada 6 pasos.
Rutas permitidas	Cambios fuera de alcance.	Solo `docs/` y `tests/`.
Modo de escritura	Cambios antes de revisar.	`dry_run` obligatorio.
Filas o resultados	Respuestas gigantes de tools.	`limit = 20`.
Red	Conexiones no necesarias.	Allowlist de dominios.

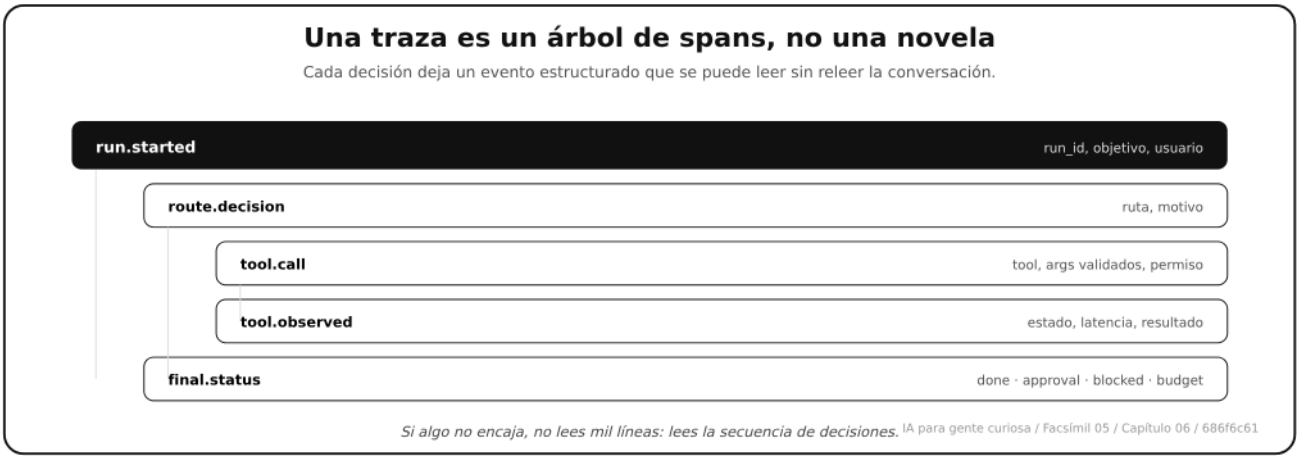
El NIST AI RMF insiste en gestionar sistemas de IA mediante funciones de gobernanza, mapeo, medición y gestión.[6] Traducido a ingeniería de agentes: no basta con que el sistema sea capaz; tiene que operar dentro de límites que alguien pueda explicar.

El cortacircuitos: parar antes de quemar presupuesto

El presupuesto frena un agente que va lento. Pero hay un fallo peor: un agente que entra en bucle, reintenta lo que no funciona o llama sin parar a una tool caída. Para eso, la ingeniería de fiabilidad lleva décadas usando el patrón del **cortacircuitos** (circuit breaker): tras varios fallos seguidos de una dependencia, se «abre» el circuito y se deja de llamar durante un tiempo, devolviendo un error rápido en vez de colgarse.[7]

En un harness de agentes esto se traduce en reglas concretas: si una tool falla tres veces seguidas, se marca como degradada y el agente pasa a su plan de recuperación (otra ruta, abstención o handoff a una persona); si el agente repite la misma acción sin que cambie la observación, se detecta el bucle y se para. Un cortacircuitos no es desconfianza: es la diferencia entre un fallo acotado y una factura desbocada.

Trazas: lo que hay que guardar y lo que no



Una traza útil no es el pensamiento privado del modelo copiado entero. Es una estructura de eventos.

EVENTO	CAMPOS MÍNIMOS	POR QUÉ IMPORTA
`task.received`	`run_id`, objetivo, usuario/tenant, canal.	Sitúa la ejecución.
`architecture.selected`	patrón, motivo, versión de instrucciones.	Explica por qué se eligió el flujo.
`model.called`	proveedor, modelo, temperatura, tokens, prompt version.	Permite comparar cambios.
`tool.called`	tool, argumentos validados o redactados, permiso.	Audita acciones.
`tool.observed`	estado, latencia, tamaño, error, resultado resumido.	Convierte acción en observación.
`budget.updated`	pasos, tools, coste, tiempo restante.	Detecta deriva.
`gate.checked`	criterio, resultado, evidencia.	Explica el go/no-go.
`handoff.created`	estado final, pendientes, siguiente acción.	Permite continuar.

W3C Trace Context define una forma estándar de propagar identificadores de traza entre sistemas distribuidos.[8] No necesitamos implementar todo el estándar en este capítulo, pero sí adoptar la disciplina: cada ejecución debe tener un identificador estable y cada paso debe poder conectarse con el anterior.

La deuda técnica de los sistemas de ML ya se estudió antes del boom de agentes: Sculley y colaboradores explicaron cómo modelos, datos, dependencias y cambios ocultos pueden acumular deuda difícil de ver.[9] Amershi y colaboradores mostraron que construir software con aprendizaje automático introduce retos especiales de datos, evaluación, monitorización y operación.[10] Con agentes ocurre algo parecido, pero más visible: el sistema toma pasos, llama tools y deja efectos. Sin trazas, la deuda se vuelve conversación perdida.

Un ejemplo concreto: revisión académica con tools

Imagina un agente que revisa un capítulo de este facsímil. La tarea no es “mejora el texto” en abstracto. Queremos:

PIEZA	DECISIÓN DE HARNESS
Objetivo	Revisar citas, estilo y coherencia antes de publicar.
Scope	Solo el capítulo activo y `referencias.bib`.
Tools	`buscar_fuente`, `validar_apa`, `proponer_diff`, `ejecutar_build`.
Límites	8 pasos, 6 tools, sin publicar, sin tocar otros facsímiles.
Sensores	Build, diff, enlaces, tabla de citas, ausencia de palabras prohibidas.
Gate	No avanza si falta fuente, si el diff toca rutas no permitidas o si el build falla.
Handoff	Qué se cambió, qué se verificó, qué queda pendiente.

La diferencia con un prompt genérico es enorme. El modelo puede seguir siendo el mismo, pero el problema está encarrilado. Tiene menos espacio para improvisar y más señales para corregirse.

De la demo al sistema: lo que se rompe por el camino

Casi todo agente nace de una demo que funciona. Alguien conecta un modelo a un par de herramientas, le da una tarea bonita y el sistema la resuelve delante de todos. Esa demo es real y es valiosa: demuestra que la idea tiene fondo. El problema es el malentendido que viene después, cuando se confunde «ha funcionado una vez delante de mí» con «funciona». Entre esas dos frases hay un abismo, y el harness es precisamente el puente. Vale la pena recorrer ese abismo despacio, porque cada cosa que se rompe en medio corresponde a una pieza del arnés.

Lo primero que se rompe es la **repetibilidad**. La demo se hizo con una entrada amable; en producción llegan entradas raras, ambiguas, malintencionadas o simplemente distintas. El sistema no determinista, ante variaciones pequeñas, toma caminos distintos, y de pronto la trayectoria que viste no es la que ocurre. Aquí aparece la necesidad del **estado tipado y las observaciones estructuradas**: sin ellas, cada ejecución es irrepetible y no hay nada que depurar. Lo segundo que se rompe es el **coste**. En la demo costó unos céntimos; con tráfico real, el bucle que crece, los reintentos y el contexto acumulado multiplican la factura, y el «coste por llamada» que se midió resulta engañoso frente al «coste por tarea aceptada» que de verdad importa. Aquí aparece el **presupuesto** como condición de parada, no como adorno.

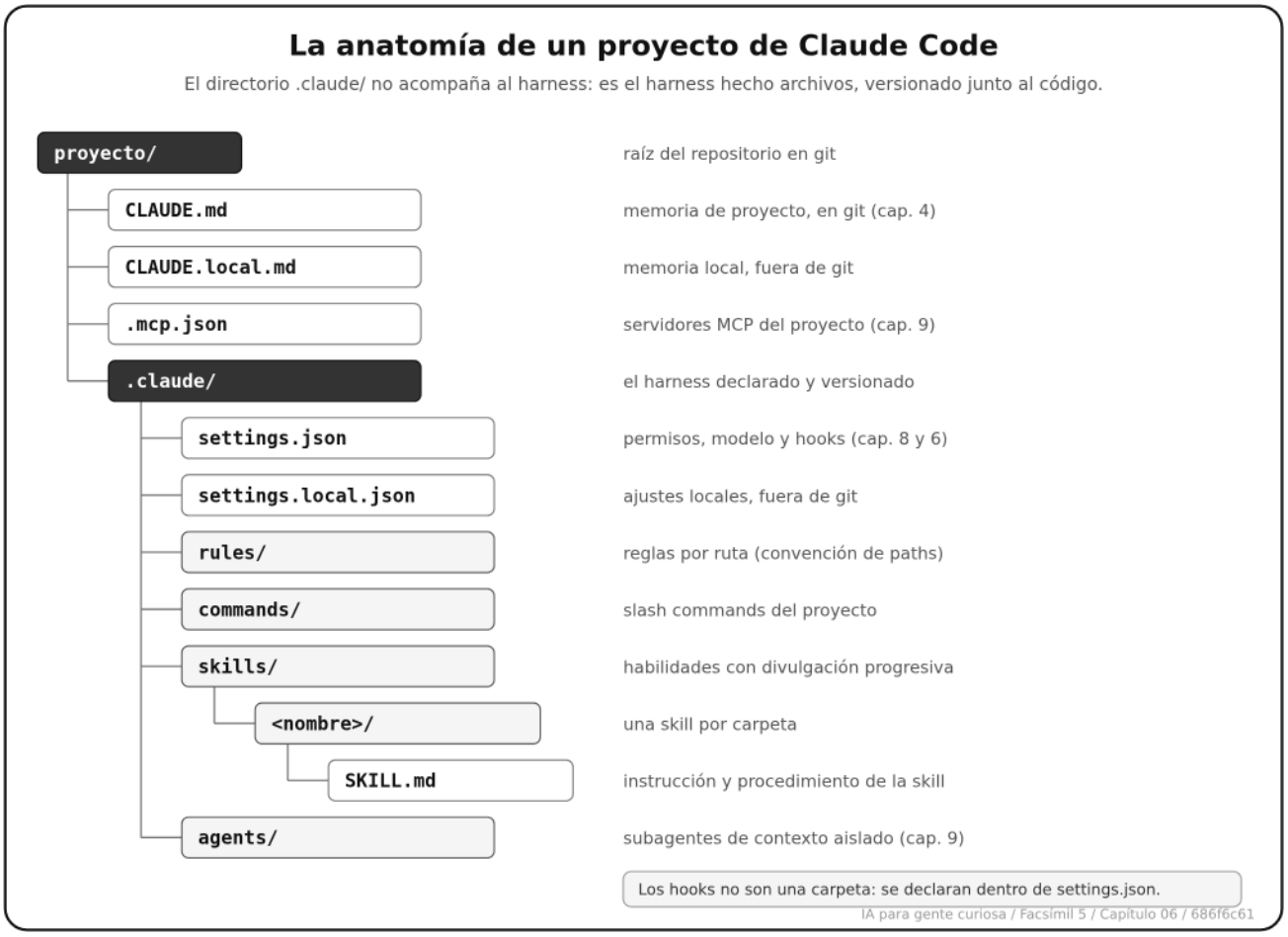
Lo tercero que se rompe es la **frontera de permisos**. En la demo, el agente solo leía; en producción, alguien le pide que «también cree el ticket» o «que envíe el correo», y de repente una acción con efecto externo está en manos de un sistema no determinista. Aquí aparece la **autonomía graduada**: la tool de escritura no se autoriza por estar en la lista, se autoriza paso a paso, con aprobación cuando el efecto lo merece. Lo cuarto que se rompe es la **capacidad de explicar**. Mientras todo va bien, nadie echa de menos las trazas; el día que el sistema falla, sin trazas no hay forma de saber qué tool se llamó, con qué argumentos, qué observó y por qué decidió lo que decidió. Aquí aparece la **observabilidad** como parte del producto, no como un lujo de ingeniería.

Por eso el harness no es burocracia que ralentiza la demo: es la lista exacta de lo que la demo no necesitaba y el sistema sí. Cada capa (estado, presupuesto, permisos, verificadores, recuperación, traza) tapa una de las grietas por las que una demo brillante se desangra en producción. Y hay una forma honesta de saber si tu agente ha cruzado el abismo o sigue siendo una demo disfrazada: pregúntate si puedes ejecutarlo cien veces seguidas, ponerle un límite de gasto, impedir que toque producción sin permiso y, cuando una de esas cien ejecuciones falle, reconstruir exactamente qué pasó. Si la respuesta a las cuatro es sí, tienes un sistema. Si falla alguna, todavía tienes una demo, por buena que se vea.

Caso real: la anatomía de un proyecto de Claude Code

A lo largo del capítulo hemos definido el harness en abstracto: estado, límites, sensores, trazas, gates y handoff. Toca aterrizarlo en algo que puedas abrir, leer y versionar. Claude Code, la herramienta de agentes de código de Anthropic, ofrece un ejemplo limpio: su configuración no vive en una pantalla de ajustes opaca, sino en un puñado de archivos dentro del propio repositorio. Ese directorio `.claude/` no acompaña al harness: **es** el harness hecho archivos. Cada pieza que presentamos en abstracto tiene aquí un fichero concreto. El contexto persistente vive en `CLAUDE.md` (la memoria del capítulo 4); los permisos, en `settings.json` (la frontera del capítulo 8); las herramientas externas, en `.mcp.json`, y los subagentes, en `agents/` (la orquestación del capítulo 9); los sensores y límites deterministas, en los hooks (los de este capítulo 6). En vez de explicar el arnés, lo enseñamos abierto en canal.

Un aviso de método antes de seguir, en la línea de la sección «Fecha de corte de herramientas» de este capítulo: lo que describimos son detalles de Claude Code a fecha de **consulta: 26 de junio de 2026**, y la herramienta evoluciona deprisa. Los nombres de archivo, los campos y las rutas pueden cambiar entre versiones; lo estable es el mecanismo (contexto, permisos, tools, subagentes y automatización declarados como archivos versionados). Para el detalle vigente, la fuente es siempre la documentación oficial.[11]



Recorramos el árbol pieza a pieza. Cada archivo responde a una de las preguntas que el harness deja explícitas, y todos comparten una virtud: viven en el repositorio, así que se revisan, se discuten en una *pull request* y se versionan como cualquier otro código.

CLAUDE.md: la memoria de proyecto

`CLAUDE.md` es la memoria persistente del proyecto. Se carga al inicio de cada sesión y contiene instrucciones estables: cómo se construye, cómo se prueba, qué convenciones se respetan. No es código ejecutable; es contexto que el agente lee antes de actuar, exactamente la capa de instrucciones del capítulo 4.

```
# Build & Test
- Build: `npm run build`
- Test: `npm test` antes de commitear
# Estándares
- Indentación de 2 espacios
- TypeScript en modo strict
```

La memoria tiene jerarquía, y este es el punto que más se malinterpreta: los niveles **se concatenan en orden, no se sobrescriben**. Primero entra la política gestionada del sistema, luego `./CLAUDE.local.md` (local, ignorado por git), después `./CLAUDE.md` o `./.claude/CLAUDE.md` (de proyecto, en git) y por último `~/.claude/CLAUDE.md` (de usuario). Todos se suman; ninguno tapa al anterior. Además, `CLAUDE.md` admite importaciones con `@ruta` (relativas al fichero, hasta cuatro niveles de profundidad), y `CLAUDE.local.md` sigue siendo una recomendación vigente para notas que no quieres subir al repositorio.[12]

.mcp.json: las tools externas del proyecto

`.mcp.json` declara los servidores MCP a nivel de proyecto, compartidos por git con todo el equipo. Es la puerta por la que entran herramientas externas, y por eso pertenece a la capa de orquestación del capítulo 9. Conviene fijar un detalle de ámbito: las configuraciones de MCP local y de usuario no viven aquí, sino en `~/.claude.json`; `.mcp.json` es solo el ámbito de proyecto.

```
{
  "mcpServers": {
    "github": { "type": "http", "url": "https://api.githubcopilot.com/mcp/", "headers": {
"Authorization": "Bearer TOKEN" } }
  }
}
```

La precedencia entre ámbitos va de local a project (este `.mcp.json`), luego user, plugin y conectores. La forma recomendada de crear la entrada de proyecto no es editar el JSON a mano, sino el comando `claude mcp add --transport http github --scope project <url>`, que escribe el bloque correcto por ti.[13]

settings.json y settings.local.json: los permisos

`settings.json` es el contrato de permisos del capítulo 8 hecho fichero: define qué puede y qué no puede hacer el agente, qué modelo usa, qué variables de entorno hereda y qué hooks se disparan. `settings.local.json` es su gemelo local, ignorado por git, para ajustes de tu máquina que no quieres imponer al equipo.

```
{
  "model": "claude-sonnet-4-6",
  "permissions": { "allow": ["Bash(npm *)"], "deny": ["Bash(rm -rf *)", "Read(.env)"] },
  "hooks": { "PostToolUse": [ { "matcher": "Edit|Write", "hooks": [ { "type": "command",
"command": "npx prettier --write" } ] } ] }
}
```

La precedencia aquí es **por capa completa**, no clave a clave: gana la capa de mayor prioridad entera, no una mezcla campo a campo. De mayor a menor: configuración gestionada, banderas de la línea de comandos, `.claude/settings.local.json` (local), `.claude/settings.json` (de proyecto) y `~/.claude/settings.json` (de usuario). Casi todo se recarga en caliente salvo el modelo y el estilo de salida.[14]

commands/: los slash commands

Cada archivo `.claude/commands/*.md` se convierte en un *slash command* (comando con barra) invocable como `/nombre`. Es la forma de empaquetar un procedimiento repetible: un despliegue, una revisión, una limpieza. En el *frontmatter* (la cabecera de metadatos) admite `argument-hint`, `allowed-tools` y `model`; en el cuerpo, `$ARGUMENTS` recibe los argumentos, `@ruta` referencia archivos y `!` ejecuta una orden de consola.

```
---
argument-hint: "[PR_NUMBER]"
allowed-tools: [Read, Bash]
model: sonnet
---
Despliega la PR $ARGUMENTS: valida con `gh pr view $ARGUMENTS` y ejecuta `npm run deploy`.
```

Un matiz importante de versiones: desde la v2.1, *commands* y *skills* están fusionados, de modo que ambos generan el mismo `/comando`. La distinción histórica entre «comandos» y «habilidades» se ha difuminado en la práctica.[15]

agents/: los subagentes

Cada archivo `.claude/agents/*.md` define un subagente: un trabajador con **contexto aislado**, su propio system prompt, sus tools y sus permisos. Es la pieza que hace posible la orquestación del capítulo 9, porque permite delegar una subtarea sin contaminar el hilo principal. En el frontmatter declara `name`, `description` (cuándo conviene delegar en él), `tools`, `model` (`sonnet`, `opus`, `haiku`, `fable` o `inherit`), `permissionMode` y `memory`.

```
---
name: code-reviewer
description: Revisa código en busca de calidad y seguridad
tools: Read, Grep, Glob
model: sonnet
---
Eres un revisor senior. Céntrate en legibilidad, seguridad y rendimiento.
```

La invocación puede ser automática (cuando la `description` encaja con la tarea), explícita (`@nombre`) o por consola (`claude --agent nombre`).[16]

skills//SKILL.md: las habilidades

Una *skill* (habilidad) es una instrucción con su procedimiento, guardada en `.claude/skills/<nombre>/SKILL.md`, que el agente invoca con `/nombre` o detecta como relevante por sí mismo. Su rasgo distintivo es la **divulgación progresiva** (progressive disclosure): solo se carga en contexto cuando hace falta, de modo que tener muchas habilidades no infla el prompt de cada sesión. El frontmatter admite `name`, `description`, `disable-model-invocation`, `context` (`main` o `fork`) y `allowed-tools`, y el directorio puede incluir archivos auxiliares.

Las habilidades existen en tres ámbitos: personal (`~/.claude/skills`), de proyecto (`.claude/skills`) y de plugin. La diferencia operativa con un subagente es nítida: una skill corre **en tu propio contexto**, mientras que un subagente trabaja en un contexto aislado.[17]

hooks: los sensores y límites deterministas

Aquí está el corazón de este capítulo. Los hooks son automatizaciones **deterministas** que se disparan en eventos del ciclo de vida, y por eso encajan con los sensores y límites del harness: no dependen del criterio del modelo, se ejecutan siempre. El detalle que casi ningún infográfico acierta: un *hook* (gancho) no es una carpeta suelta, **se declara dentro de `settings.json`**.

EVENTO	CUÁNDO SE DISPARA	MATCHER TÍPICO
`SessionStart`	Al iniciar, reanudar, limpiar o compactar.	`startup`, `resume`, `clear`, `compact`
`PreToolUse`	Antes de ejecutar una tool.	por herramienta (`Bash`, `Edit`…)
`PostToolUse`	Después de ejecutar una tool.	por herramienta
`Stop`	Cuando el agente termina su turno.	—
`PreCompact`	Antes de comprimir el contexto.	`manual`, `auto`
`ConfigChange`	Al cambiar la configuración.	—
`InstructionsLoaded`	Al cargar las instrucciones.	—

Los hooks pueden ser de tipo `command`, `prompt`, `agent`, `http` o `mcp_tool`. Reciben por entrada estándar un JSON con la forma `{"tool_name":..., "tool_input":{...}}`, y su código de salida decide: salir con `2` bloquea la acción y muestra el `stderr` al agente; salir con `0` la permite. Este es el ejemplo canónico de un límite duro, el cortacircuitos que vimos antes hecho hook: un `PreToolUse` que bloquea cualquier `rm -rf` antes de que llegue a la consola.

```
{
  "hooks": {
    "PreToolUse": [
      { "matcher": "Bash", "hooks": [
        { "type": "command", "command": "bash -c 'in=$(cat); if echo \"$in\" | jq -r
\".tool_input.command\" | grep -q \"rm -rf\"; then echo \"Bloqueado: rm -rf no permitido\"
>&2; exit 2; fi'" }
      ] }
    ]
  }
}
```

Ese fragmento es un sensor y un límite a la vez: observa el comando propuesto y lo veta antes de que tenga efecto, sin pedir permiso al modelo.[18]

rules/: convención frente a lo nativo

La carpeta `.claude/rules/` aparece en muchos esquemas como si fuera un mecanismo del producto, pero es **solo una convención** para organizar reglas. Lo nativo es el frontmatter `paths:` en cualquier `.md`: sin `paths`, el archivo se carga siempre; con `paths`, se carga

únicamente al abrir un archivo que case con el glob indicado (una carga diferida y condicional).

```
---
paths:
  - "src/api/**/*.ts"
---
# Reglas de API
- Formato de error estándar
- Documentar con OpenAPI
```

Así, las reglas de la API solo entran en contexto cuando trabajas en la API, y no pesan el resto del tiempo.[19]

Lo que los infográficos suelen equivocar

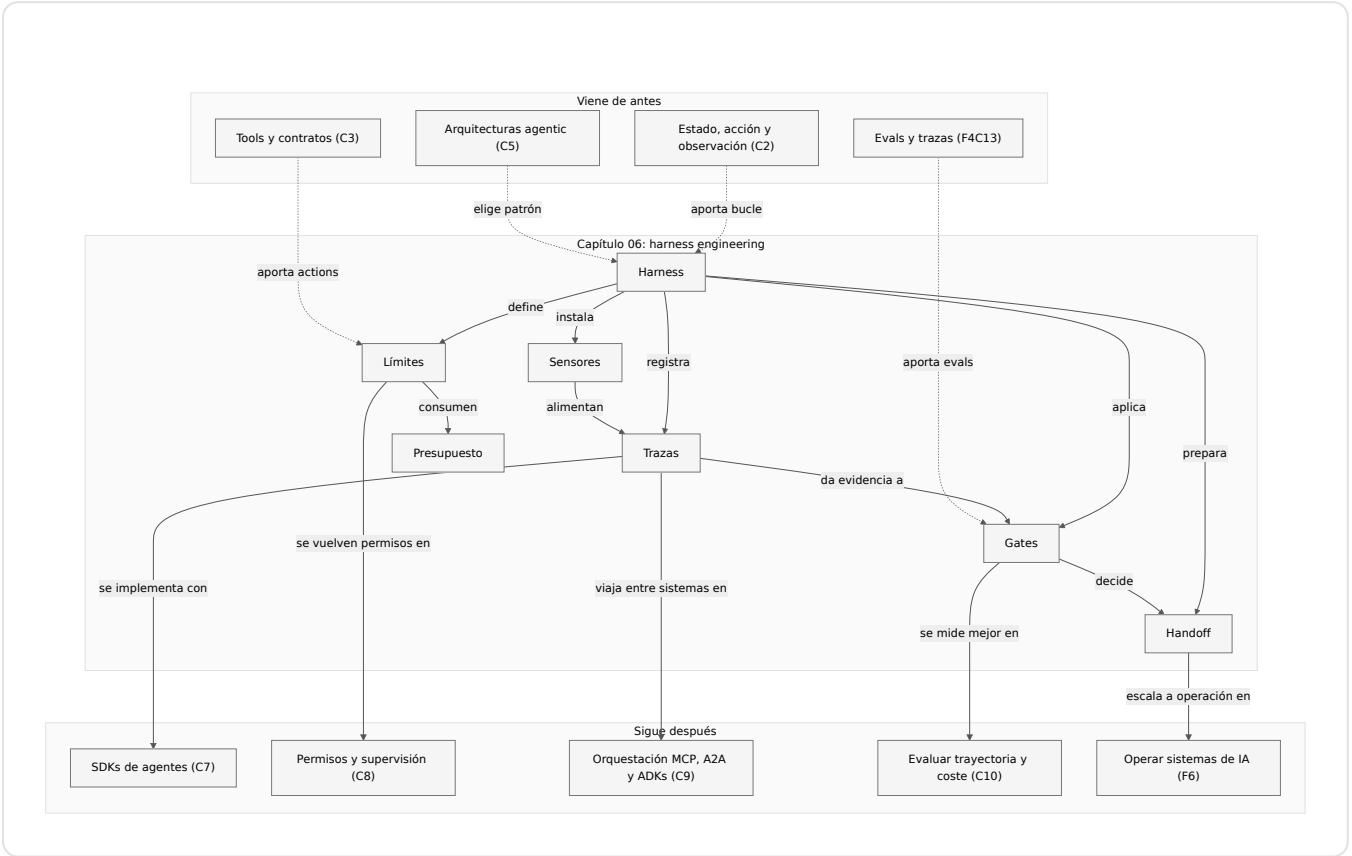
Circulan muchos diagramas de la estructura de Claude Code, y casi todos repiten los mismos ocho errores. Conviene tenerlos a mano.

1. **Los hooks no son una carpeta.** Se declaran dentro de `settings.json`, no en un directorio `hooks/` suelto.
2. **`rules/` es una convención, no un mecanismo nativo.** Lo nativo es el frontmatter `paths:` en cualquier `.md`.
3. **`.mcp.json` es solo del proyecto.** Las configuraciones de MCP local y de usuario viven en `~/.claude.json`, no en `.mcp.json`.
4. **`commands` y `skills` ya no son cosas distintas.** Desde la v2.1 ambos generan el mismo `/comando`.
5. **Los subagentes Explore y Plan no cargan el `CLAUDE.md` completo.** Arrancan con un contexto reducido y aislado, no con toda la memoria de proyecto.
6. **La auto-memoria es un mecanismo aparte.** No es lo mismo que `CLAUDE.md`, aunque ambos aporten contexto.
7. **La precedencia de `settings` es por capa completa.** Gana la capa de mayor prioridad entera, no se mezcla clave a clave.
8. **La jerarquía de `CLAUDE.md` se concatena.** Los niveles se suman en orden; ninguno sobrescribe al anterior.

Visto en conjunto, el `.claude/` desmiente la idea de que un buen agente es cuestión de magia o de un prompt afortunado. Un harness es contexto (`CLAUDE.md`, `rules/`), permisos (`settings.json`), sensores y límites (hooks), tools y subagentes (`.mcp.json`, `agents/`) y procedimientos repetibles (`commands/`, `skills/`), todo declarado, todo legible, todo versionado junto al código. Cuando algo falla, no abres una caja negra: abres un archivo, lees qué

decidiste y lo cambias en una línea. Eso es, en última instancia, lo que distingue una demo de un sistema, y por eso este directorio es la traducción más tangible de todo lo que llevamos visto en el capítulo.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Harness	Arnés técnico que limita, observa, registra y verifica una ejecución.
Sensor	Señal que vuelve del sistema: test, diff, log, métrica, resultado de tool.
Traza	Registro estructurado de eventos de una ejecución.
Span	Unidad de trabajo dentro de una traza.
Gate	Regla que decide si una ejecución avanza, se corrige o se detiene.
Presupuesto operativo	Límite de pasos, coste, tokens, tools, tiempo y alcance.
Coste por tarea aceptada	Coste real dividido entre ejecuciones que pasan el gate.
Handoff	Estado resumido para que otra persona o sistema pueda continuar.
Stop reason	Motivo estructurado de parada: terminado, falta evidencia, falta permiso, presupuesto agotado.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Confundir harness con prompt largo	El prompt no limita coste, permisos ni trazas.	Separar instrucciones, estado, tools, sensores y gates.
Guardar logs sin estructura	Luego nadie puede comparar ejecuciones.	Usar eventos con `run_id`, `tool`, latencia, coste y resultado.
Medir solo respuesta final	No sabes si el camino fue caro, frágil o fuera de alcance.	Evaluar outcome y trayectoria.
No poner límites de presupuesto	El agente puede gastar pasos y tools sin progreso.	Definir máximos y stop reasons.
Dejar que el constructor se apruebe solo	Una salida convincente no equivale a evidencia.	Separar builder, reviewer y gate.
No diseñar handoff	Cada sesión futura reconstruye la historia.	Guardar objetivo, decisiones, evidencia, riesgos y siguiente acción.

Antes de pasar página

- [ ] ¿Sé explicar qué añade un harness que no añade un prompt?
- [ ] ¿Puedo escribir $\mathcal{H} = (G, I, S, A, P, B, V, R, \tau)$ y explicar cada pieza?
- [ ] ¿Sé distinguir sensor, traza, span y gate?
- [ ] ¿Sé definir límites de pasos, tools, coste, tiempo y alcance?
- [ ] ¿Sé qué eventos mínimos debería guardar una ejecución agentic?
- [ ] ¿Puedo explicar por qué coste por tarea aceptada importa más que precio por token?
- [ ] ¿Sé construir un gate que mire resultado y trayectoria?
- [ ] ¿Sé qué debe contener un handoff para que otra persona continúe?
- [ ] ¿He ejecutado el mini harness y leído la traza generada?

En resumen

IDEA FUERZA	DETALLE
Un agente sin harness es difícil de operar.	Puede acertar una vez y ser imposible de depurar después.
Los límites son parte de la arquitectura.	Pasos, tools, coste, tiempo y alcance definen la autonomía real.
Los sensores convierten acciones en observaciones.	Tests, diffs, métricas y resultados de tools alimentan el estado.
Las trazas son memoria operativa.	Permiten explicar qué ocurrió, comparar versiones y corregir fallos.
El gate protege la salida.	No basta con responder: hay que pasar criterios de resultado, trayectoria y coste.
El handoff evita deuda de contexto.	Otra persona o agente debe poder continuar sin reconstruir toda la conversación.

Para saber más

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B. y Zimmermann, T. (2019). Software engineering for machine learning: A case study. *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, 291-300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

Anthropic. (2024). *Building Effective Agents*. https://www.anthropic.com/engineering/building-effective-agents

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

Baylor, D. y otros (2017). TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1387-1395. https://doi.org/10.1145/3097983.3098021

Fowler, M. (2025). *Harness Engineering for Coding Agent Users*. https://martinfowler.com/articles/harness-engineering.html

NIST. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. https://doi.org/10.6028/NIST.AI.100-1

Nygard, M. T. (2018). *Release It! Design and Deploy Production-Ready Software* (2.ª ed.). Pragmatic Bookshelf.

OpenAI. (2026). *Agents SDK: Tracing*. https://openai.github.io/openai-agents-python/tracing/

OpenTelemetry. (2026). *Tracing API*. https://opentelemetry.io/docs/specs/otel/trace/api/

OpenAI. (2026). *AGENTS.md*. https://github.com/openai/agents.md

Sculley, D. y otros (2015). *Hidden Technical Debt in Machine Learning Systems*. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html

W3C. (2021). *Trace Context Level 2*. https://www.w3.org/TR/trace-context-2/

Notas

1. Fowler, M. (2025). *Harness Engineering for Coding Agent Users*. https://martinfowler.com/articles/harness-engineering.html. Consultado el 10 de junio de 2026. ↩

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

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

4. OpenTelemetry. (2026). *Tracing API*. https://opentelemetry.io/docs/specs/otel/trace/api/. Consultado el 10 de junio de 2026. ↩

5. Anthropic. (2025). *Develop Tests for LLM Applications*. https://platform.claude.com/docs/en/build-with-claude/develop-tests. Consultado el 10 de junio de 2026. ↩

6. Tabassi, E. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. NIST AI 100-1. https://doi.org/10.6028/NIST.AI.100-1 ↩

7. Nygard, M. T. (2018). *Release It! Design and Deploy Production-Ready Software* (2.ª ed.). Pragmatic Bookshelf. Describe el patrón circuit breaker para aislar fallos de dependencias y evitar cascadas. ↩

8. W3C. (2021). *Trace Context Level 2*. https://www.w3.org/TR/trace-context-2/. Consultado el 10 de junio de 2026. ↩

9. Sculley, D. y otros (2015). *Hidden Technical Debt in Machine Learning Systems*. Advances in Neural Information Processing Systems, 28. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html ↩

.0. Amershi, S. y otros (2019). Software engineering for machine learning: A case study. *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, 291-300. https://doi.org/10.1109/ICSE-SEIP.2019.00042 ↩

.1. Anthropic. (2026). *Claude Code documentation*. https://code.claude.com/docs. Consultado el 26 de junio de 2026. ↩

.2. Anthropic. (2026). *Manage Claude's memory*. https://code.claude.com/docs/en/memory. Consultado el 26 de junio de 2026. ↩

.3. Anthropic. (2026). *Connect Claude Code to tools via MCP*. https://code.claude.com/docs/en/mcp. Consultado el 26 de junio de 2026. ↩

.4. Anthropic. (2026). *Claude Code settings*. https://code.claude.com/docs/en/settings. Consultado el 26 de junio de 2026. ↩

.5. Anthropic. (2026). *Skills*. https://code.claude.com/docs/en/skills. Consultado el 26 de junio de 2026. ↩

.6. Anthropic. (2026). *Subagents*. https://code.claude.com/docs/en/sub-agents. Consultado el 26 de junio de 2026. ↩

.7. Anthropic. (2026). *Skills*. https://code.claude.com/docs/en/skills. Consultado el 26 de junio de 2026. ↩

.8. Anthropic. (2026). *Get started with hooks*. https://code.claude.com/docs/en/hooks-guide. Consultado el 26 de junio de 2026. ↩

.9. Anthropic. (2026). *Manage Claude's memory*. https://code.claude.com/docs/en/memory. Consultado el 26 de junio de 2026. ↩