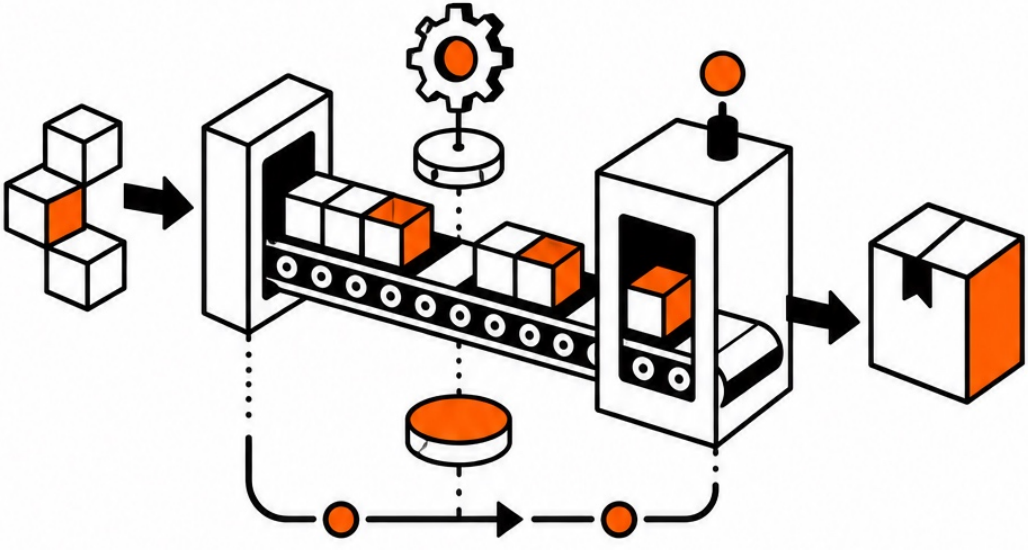
Principios del Software

FACTORY MODEL



Javier Rayón
librosdeia.com

Contenido

1	El nuevo factory model de siempre	2
2	Cuatro capas, cuatro preguntas	7
3	El loop: objetivo y condición de parada	10
4	El harness: guías, sensores y límites	13
5	El grafo: precedencia, ramas y reanudación	16
6	La spec: el criterio por escrito antes de construir	19
7	La cadena completa de la fábrica	22
8	Diagnóstico y orden de lectura	26

Principios del Software Factory Model

Loop, harness, graph y spec-driven: las cuatro capas de la ingeniería con IA

© 2026 Javier Rayón. Todos los derechos reservados.

Publicado por librosdeia.com

Guía de distribución gratuita. Se permite compartir el fichero completo y sin modificar. Ninguna parte de esta publicación puede reproducirse de forma parcial, alterada o dentro de otra obra sin la autorización previa por escrito del autor.

Los nombres de productos y servicios mencionados son marcas registradas de sus respectivos propietarios. Esta guía no está afiliada a Anthropic, OpenAI ni ninguna otra empresa mencionada, ni patrocinada por ellas.

Los principios recogidos aquí resumen los cuatro volúmenes de la serie AI Engineering del mismo autor. Las atribuciones y fechas corresponden a las fuentes citadas en esos volúmenes, verificadas a septiembre de 2026. La información se ofrece sin garantía de ningún tipo y el autor no será responsable de los daños derivados de su uso.

Primera edición: septiembre de 2026.

SECCIÓN 1

El nuevo factory model de siempre

Una planta de montaje de coches tiene resuelto desde hace décadas el problema que la fábrica de software hereda: producir en serie sin que una persona sostenga cada herramienta. Sus piezas se corresponden una a una con las cuatro capas de esta guía, y esa correspondencia ayuda a ver por qué existe cada capa.

En cada estación, un robot hace una sola operación y la repite hasta que la medida de la pieza entra en tolerancia. A su alrededor están las pinzas y plantillas que le permiten trabajar, la jaula que limita hasta dónde llega el brazo y los sensores que miden la pieza al salir. La cinta une las estaciones en un orden fijo, con una sublínea de puertas que se incorpora en un punto concreto, un control de calidad donde una persona firma y una vía de retrabajo que devuelve una pieza a una estación anterior. Y en el atril de cada estación están el plano del coche y la orden de fabricación.

Capa	En la planta	Lo que decide
Loop	El robot de cada estación	Cuándo la pieza está hecha
Harness	Pinzas, plantillas, jaula de seguridad y sensores de la estación	Con qué cuenta el robot y hasta dónde llega
Grafo	El trazado de la cinta: sublíneas, puntos de unión, control de calidad y vía de retrabajo	Qué estación viene después de cuál
Spec	El plano del coche y la orden de fabricación	Qué se monta y con qué tolerancias

La persona trabaja fuera de la línea: diseña el trazado, ajusta el utillaje de una estación cuando su robot falla y revisa lo que sale por el final. Ese reparto, mecánica para la fábrica y juicio para la persona, se mantiene en las cuatro capas.

La analogía sirve para situar los cuatro conceptos, y falla en dos puntos en los que el software se aparta de la fabricación de coches. El robot de la planta es determinista y el agente no lo es, así que las comprobaciones del harness, que en la planta son una

garantía de calidad, en software son lo que dice si la pieza sirve. Y la planta fabrica miles de unidades iguales mientras la fábrica de software resuelve una tarea distinta cada vez, de modo que la spec cuenta en cada tarea, además de al diseñar la línea.

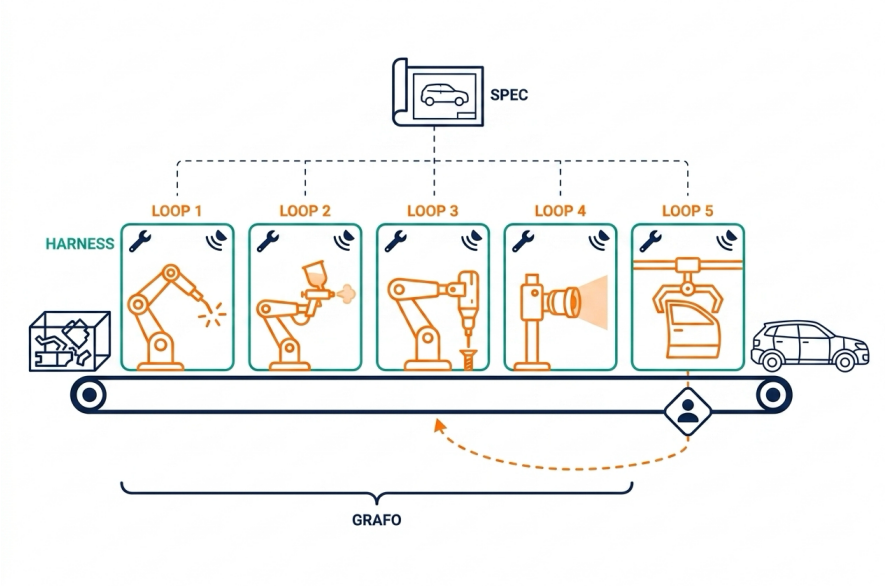


Figura 1.1: La misma fábrica vista como una planta de coches: cada capa tiene su pieza en la línea.

1.1 De asistente a fábrica

Un agente de código rinde bien mientras alguien está delante. Le pides un cambio, lee el repositorio, lo implementa, ejecuta los tests y abre el pull request. El problema aparece cuando el volumen de trabajo crece más deprisa que el tiempo que puedes pasar en la sesión. Las incidencias entran a un ritmo que ninguna conversación absorbe, y el agente, por bueno que sea, sigue esperando a que alguien le diga qué hacer a continuación.

La primera reacción suele ser mejorar el prompt, añadir instrucciones al fichero de memoria del proyecto y probar otro modelo. Todas mejoran algo durante unos días, de modo que parece que el diagnóstico era correcto, y el límite reaparece en cuanto cambia la carga. Lo que hace falta es otra forma de organizar el trabajo.

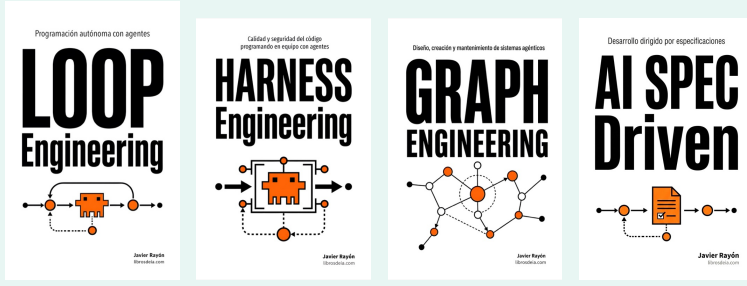
El **asistente** es la forma de trabajar de la que parte casi todo el mundo. Una persona escribe software y la IA le ayuda a escribirlo más rápido, con la persona al mando de cada turno. Sirve para la mayor parte del día a día y deja de escalar en el punto descrito al empezar este apartado.

La **fábrica** funciona al revés. Un sistema produce software mientras la persona lo diseña, lo alimenta y lo supervisa. El trabajo entra por un lado, en forma de incidencias,

fallos de integración continua o peticiones, atraviesa una cadena que lo reparte, lo implementa, lo verifica y lo anota, y sale por el otro como pull request en verde esperando revisión, sin que nadie dirija cada turno.

	Asistente	Fábrica
Quién está al mando de cada turno	La persona	El sistema
Objeto que se diseña	El prompt de cada tarea	La cadena que resuelve una clase de tareas
Cuándo se mejora	Dentro de la sesión, leyendo la respuesta	Entre ejecuciones, leyendo lo que el sistema hizo sin nadie delante
Qué limita el caudal	El tiempo de sesión	La capacidad de revisar con criterio lo que sale

A la segunda forma se la llama **software factory model**, y en esta guía se abrevia como factory model o fábrica. La idea aparece en *Loop Engineering*, el primer volumen de la serie AI Engineering, y los otros tres volúmenes la construyen pieza a pieza.



La serie AI Engineering. Cuatro volúmenes, uno por capa: Loop Engineering (programación autónoma con agentes), Harness Engineering (calidad y seguridad del código programando en equipo con agentes), Graph Engineering (diseño, creación y mantenimiento de sistemas agénticos) y AI Spec-driven (desarrollo dirigido por especificaciones). Se leen por separado. Disponibles en librosdeia.com.

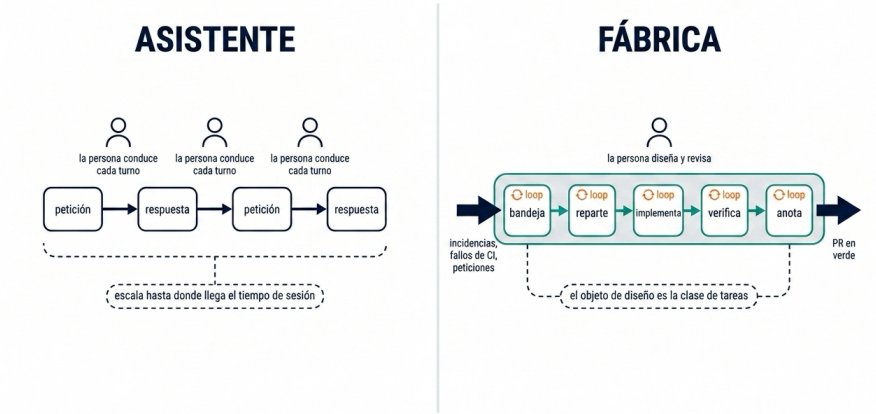


Figura 1.2: El asistente escala hasta donde llega el tiempo de sesión; la fábrica cambia el objeto de diseño por la clase entera de tareas.

1.2 Qué tiene dentro una fábrica

Una fábrica de software con agentes se compone de muchos loops pequeños encadenados, cada uno con su entorno alrededor, dispuestos sobre un mapa de qué paso puede seguir a cuál, y gobernados por una descripción escrita de qué hay que construir. Las cuatro responden a preguntas distintas.

- El **loop** fija cuándo el trabajo está hecho y da la forma de comprobarlo.
- El **harness** establece con qué cuenta el agente para trabajar: instrucciones, herramientas, permisos, comprobaciones y límites. Cada hueco que deja se rellena con una conjetura.
- El **grafo** determina qué puede ocurrir después de cada paso: qué espera a qué, qué corre en paralelo, quién aprueba y desde dónde se reanuda. Así el orden entre pasos deja de salir de lo que el modelo genere en cada turno.
- La **spec** define qué hay que construir. Sin ella, la fábrica produce a toda velocidad algo impecablemente equivocado.

En el factory model el objeto de diseño pasa a ser la clase entera de tareas. Montas una vez la cadena que resuelve, por ejemplo, la actualización semanal de dependencias, y a partir de ahí cada dependencia entra por esa cadena sin que intervengas en cada una. Tu trabajo de cada lunes queda en leer los pull requests que la fábrica produjo y vaciar la bandeja de lo que quedó sin resolver.

> **Una cadena larga exige más verificación.** Una fábrica multiplica lo que entra, incluido el error. Si la etapa que reparte clasifica mal, las siguientes trabajan a conciencia sobre la tarea equivocada y entregan un pull request correcto para una tarea equivocada. Por eso diseñar una fábrica exige más criterio que escribir un buen prompt.

Las siete secciones siguientes recorren las cuatro capas, la cadena completa y un procedimiento para diagnosticar en qué capa está fallando un sistema propio.

SECCIÓN 2

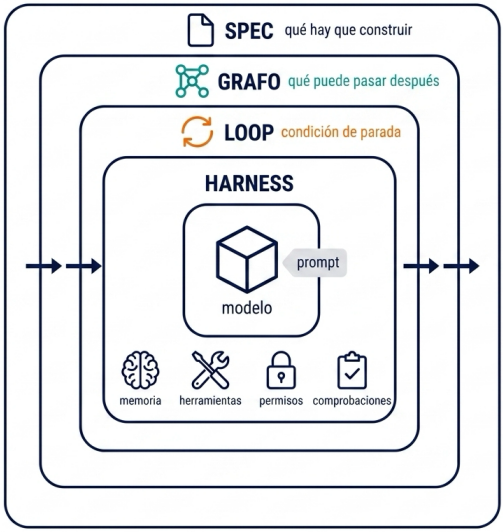
Cuatro capas, cuatro preguntas

Un sistema de agentes puede fallar en varios sitios distintos y todos producen el mismo síntoma: el agente entrega algo que no sirve. Puede fallar el entorno que le montaste, el loop cuya condición de parada debía detectar que el trabajo estaba hecho, el orden entre los pasos que nunca declaraste o la descripción de lo que había que construir. Desde fuera los cuatro se parecen, así que la capa responsable se acaba buscando por intuición.

Las cuatro capas tienen nombres fijados en el oficio, y cada una responde a una pregunta que las otras no responden.

Capa	Pregunta que responde	Cuando falla, lo que ves es…	Volumen
Loop	¿Cuándo está hecho el trabajo?	Casi funciona, pero de forma irregular	*Loop Engineering*
Harness	¿Con qué cuenta el agente para trabajar?	El agente no puede operar	*Harness Engineering*
Grafo	¿Qué puede ocurrir después de cada paso?	Nadie puede explicar el sistema entero	*Graph Engineering*
Spec	¿Qué hay que construir?	Trabajo correcto sobre el problema equivocado	*AI Spec-driven*

Por debajo de las cuatro está el **prompt**, que determina qué devuelve el modelo en una respuesta concreta. Es la capa más pequeña y la que menos rinde cuando se pule, porque un prompt se itera dentro de una sesión y sus mejoras se pierden al cerrarla. Las cuatro capas de la tabla se iteran entre ejecuciones y se acumulan.



capa	pregunta que responde
loop	¿cuándo está hecho?
harness	¿con qué cuenta el agente?
grafo	¿qué puede ocurrir después?
spec	¿qué hay que construir?

Figura 2.1: Las cuatro capas rodean al mismo modelo; cada una tiene su pregunta propia.

Prompt engineering y context engineering. Al trabajo sobre esa capa de abajo se le ha llamado prompt engineering y, cuando pasó a incluir qué ficheros, herramientas y memoria entran en la ventana, context engineering. Llamarlo ingeniería es excesivo: se limita a un conjunto de buenas prácticas para redactar la petición y elegir qué entra en el contexto, y esas prácticas hacen falta igual dentro de cualquiera de las cuatro capas. Lo que se diseña, se mide y se acumula está en las capas de la tabla.

2.1 Tres causas de confusión entre capas

La primera causa es de nomenclatura. Las cuatro capas actúan sobre los mismos ficheros del mismo repositorio, y todo lo que ocurre ahí se llama «el agente». Cuando algo sale mal, el sujeto de la frase es siempre el mismo, así que el fallo termina atribuido al modelo por descarte.

La segunda es que la palabra «bucle» nombra tres cosas distintas. Está el ciclo interno de un turno, en el que el modelo genera una acción, la herramienta la ejecuta y el resultado vuelve al contexto antes del paso siguiente. Está el ciclo de trabajo con condición de parada, que es el loop de esta guía. Y está el ciclo entre pasos, el que devuelve el trabajo a un paso anterior con el diagnóstico adjunto, que pertenece al grafo. Quien dice «tengo un problema con el bucle» puede referirse a cualquiera de los tres, y el arreglo es distinto en cada caso.

La tercera es la mejora parcial. Supongamos que un agente entrega una implementación incompleta y reaccionas ampliando las instrucciones del repositorio. Los resultados mejoran esa semana, el equipo cierra el asunto y el fallo vuelve con la siguiente subida de carga. Has gastado una semana y sigues sin saber qué lo causaba.

2.2 El encaje de las capas en un paso de la cadena

Toma el paso «implementar» de una cadena de entrega. Según el grafo, se ejecuta una vez por rama, con tres ramas a la vez, en aislamiento, y que el trabajo sigue hacia «verificar». Dentro del paso hay un loop: el agente lee la especificación, escribe código, ejecuta las pruebas, corrige y vuelve a ejecutar hasta que la condición de parada se cumple. Por debajo está el harness, que pone el directorio aislado, el comando de pruebas cuya salida hace de evidencia, la credencial acotada y la telemetría. Y por encima, la spec, que es lo que el agente lee al empezar y contra lo que el verificador compara al terminar.

Es decir, el grafo corre dentro del harness, los loops viven dentro de los nodos del grafo, y la spec gobierna qué entra en la cadena y qué cuenta como salida válida. Cuando ese encaje se rompe, el síntoma aparece a un nivel distinto de donde está la causa: una rama de «implementar» que no termina bloquea al paso «verificar», y lo que ves desde fuera es un grafo parado en un nodo que no tiene ningún problema.

2.3 Origen de los cuatro términos

Los términos se fijaron en la práctica durante 2026, fuera de la documentación de producto de los agentes de código, lo que explica parte del desorden con el que se usan.

- **Harness.** Mitchell Hashimoto usa *harness engineering* en febrero de 2026, en «My AI Adoption Journey»: cada error que descubres se paga diseñando lo que impide que vuelva a ocurrir. El 11 de febrero, OpenAI publica «Harness engineering: leveraging Codex in an agent-first world», de Ryan Lopopolo, con el diagnóstico que da nombre a la capa: el agente era capaz y el entorno estaba mal especificado.
- **Loop.** Addy Osmani fija *loop engineering* el 7 de junio de 2026, a partir de una frase de Boris Cherny, responsable de Claude Code en Anthropic, sobre haber dejado de pasar prompts para dedicarse a escribir los loops que los pasan.
- **Grafo.** *Graph engineering* aparece a mediados de julio de 2026, y LangChain lo recoge en su balance del 22 de julio, donde sitúa al loop como una versión simple del grafo.
- **Spec-driven.** El desarrollo dirigido por especificaciones tiene herramientas propias, como Spec Kit de GitHub y Kiro de AWS, y se consolida como alternativa al *vibe coding*.

La serie AI Engineering ordena las cuatro en un mismo marco y da a cada una un volumen.

SECCIÓN 3

El loop: objetivo y condición de parada

El agente escribe «hecho» al final de la respuesta y a la mañana siguiente descubres que el test que iba a probarlo llevaba una semana desactivado. Ese es un fallo de loop habitual, y tiene una versión más cara: los reintentos sin techo, con el sistema insistiendo toda la noche sobre el mismo error y una factura que ha crecido sin que nadie sepa señalar en qué punto.

Un **loop** es un objetivo recursivo con una condición de parada verificable. Lo que lo distingue de una ráfaga de prompts es que persigue un fin hasta que algo comprobable confirma que ese fin se ha alcanzado. Cuando falta un criterio de parada que se pueda medir, lo que tienes es una deriva que consume tokens sin saber cuándo detenerse.

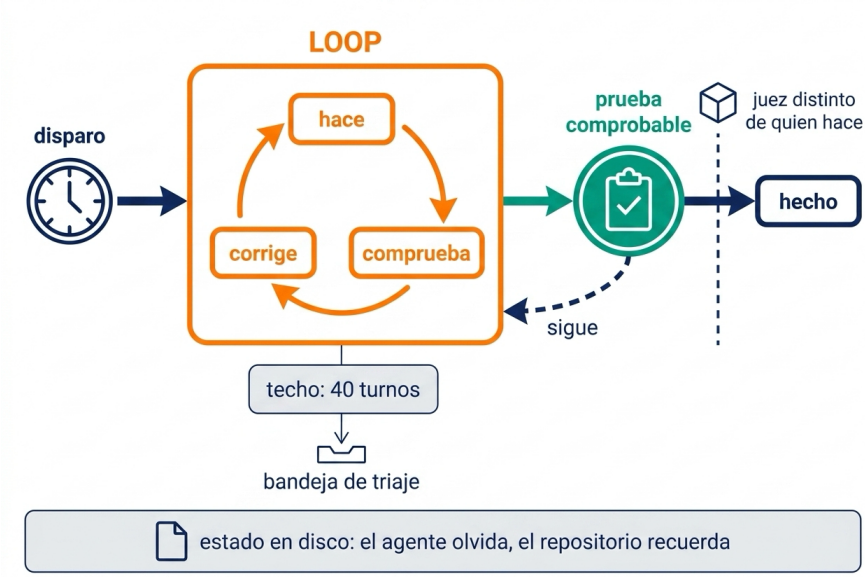


Figura 3.1: Un loop: el disparo, el ciclo de hacer y comprobar, la prueba comprobable juzgada por otro, el techo y el estado en disco.

3.1 Cuatro decisiones tomadas de antemano

Un loop bien montado lleva escritas cuatro cosas antes de arrancar.

- **Qué lo dispara.** Un fallo de integración continua, una incidencia nueva, un reloj que lo lanza cada noche o una persona desde la terminal. Ese disparo con cadencia es lo que separa a un loop de una ejecución aislada.
- **Qué prueba que ha terminado.** Un test verde, un esquema que valida, una revisión aprobada. Algo que se pueda comprobar ejecutándolo, y no el «ya está» del propio agente.
- **Quién juzga esa prueba.** Alguien distinto de quien hizo el trabajo. El modelo que ha escrito el código tiende a confirmar su propio trabajo al revisarlo, así que la revisión se encarga a un agente distinto, con otras instrucciones y, a ser posible, con otro modelo.
- **Cuántas vueltas se admiten y qué pasa al agotarlas.** Un número, y un destino para el trabajo que no se ha resuelto.

La tercera decisión es la más rentable de las cuatro. Separar a quien hace de quien verifica es lo que permite marcharse mientras el loop trabaja, porque su «hecho» pasa a significar algo.

3.2 El estado se guarda en disco

Un agente no conserva nada entre una ejecución y la siguiente. Por eso la memoria del loop reside fuera de la conversación, en un fichero de estado o en un tablero que registra lo hecho y lo pendiente: el estado persiste en el repositorio, y no en el agente.

Lo mismo vale para la intención. Cada sesión arranca en frío y el agente rellena con una conjetura cualquier hueco que dejes, de modo que las convenciones del proyecto se escriben una sola vez en una *skill* que se lee en cada ejecución.

3.3 Un ejemplo: el triaje de fallos de CI

Supongamos un loop que cada mañana revisa los fallos de integración continua de la víspera. La versión ingenua le pasa un prompt largo: revisa los fallos, distingue los reales de los tests inestables y propón un arreglo para los reales. Funciona a medias. Unos días el agente no encuentra los logs; otros clasifica como fallo real un test inestable porque no tiene delante el historial de tests inestables del equipo.

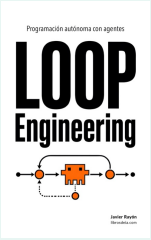
La versión con loop deja la frase como está y escribe las cuatro decisiones:

```
# ilustrativo
disparo: cada día a las 7:00
parada: "cada fallo de la víspera está clasificado como inestable
         o real, y los reales tienen ticket abierto"
```

```
juez: un segundo agente, con instrucciones de verificador
techo: 40 turnos; lo que no se resuelva va a la bandeja de triaje
```

El prompt puede ser ahora de una línea. La condición de parada es comprobable, porque cada fallo acaba clasificado con ticket o en la bandeja con motivo. Y lo que el loop registra sobre tests inestables se acumula en un fichero que crece, en lugar de perderse con la conversación.

«Hecho» sigue siendo una afirmación. Que el loop declare terminada una tarea no equivale a que esté bien resuelta, por mucho que un verificador lo respalde. Confirmar que el código funciona sigue siendo tu responsabilidad, y tu capacidad de revisión es el límite real: ningún agente adicional en paralelo amplía el número de pull requests que una persona puede leer con criterio.



Loop Engineering. Programación autónoma con agentes. Primer volumen de la serie: desarrolla esta capa en diez capítulos, desde el ciclo interno del agente hasta el gobierno de loops desatendidos, y su último capítulo presenta la idea de fábrica que esta guía resume. Disponible en librosdeia.com.

SECCIÓN 4

El harness: guías, sensores y límites

Le pides a un agente que ejecute los tests. Como en su contexto no consta que el comando real de tu repositorio sea `make test`, prueba con `npm test`, no encuentra el script y da la suite por rota. El modelo ha hecho algo sensato con lo que tenía delante. Lo que faltaba era el dato. Eso es un fallo de harness, y en la práctica son los más frecuentes.

El **harness**, o arnés, es todo lo que rodea a la llamada al modelo y que tú controlas: las instrucciones que lee al arrancar, las herramientas que puede invocar, los permisos que deciden qué hace solo y qué se detiene a pedir, las comprobaciones que le devuelven señal y los límites que no puede cruzar. El modelo es el mismo para todos los que lo alquilan. El arnés es propio de cada equipo, se compone de piezas deterministas y es lo que se acumula con el tiempo.

4.1 Cada error se convierte en ingeniería

La regla fundacional de la disciplina la formuló Mitchell Hashimoto: cada vez que el agente comete un error, inviertes el tiempo en ingeniería para que ese error no pueda repetirse. Un fallo que solo corriges es coste. Un fallo que conviertes en regla, linter, test o hook es inversión.

En la práctica, eso cambia lo que es un fichero de memoria del proyecto: cada línea existe porque un fallo concreto la motivó, y una línea que no previene nada es ruido que compite por contexto.

> **La regla de las dos correcciones.** Si has corregido lo mismo dos veces en revisión, la tercera es un linter o una línea de memoria. Se elige el linter cuando la regla es comprobable mecánicamente y la línea de memoria cuando pide juicio.

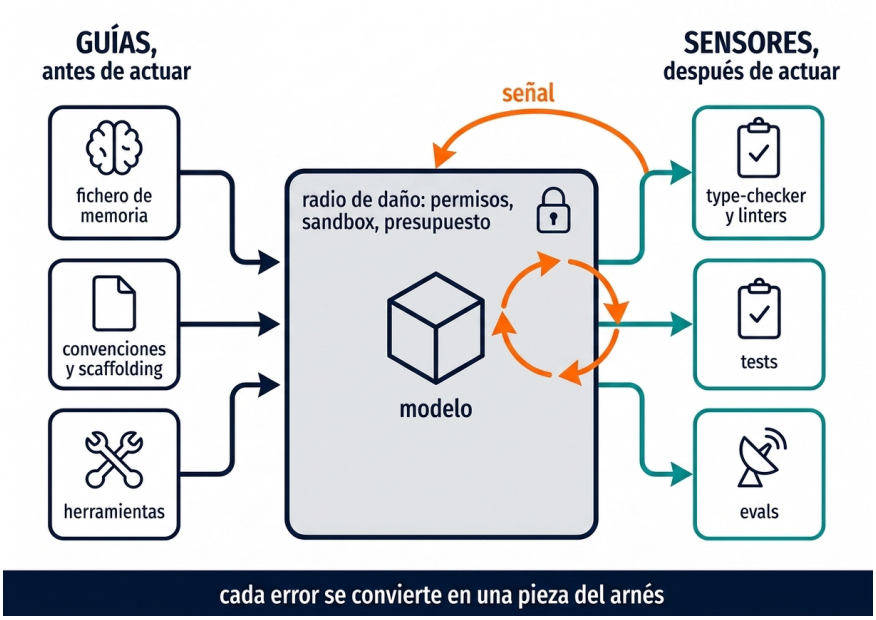


Figura 4.1: Las guías orientan antes de actuar, los sensores devuelven señal después, y el radio de daño se decide alrededor.

4.2 Guías y sensores: orientar antes, medir después

Las **guías** orientan antes de generar: la documentación de arquitectura, las convenciones escritas, el scaffolding que fija la estructura para que el agente solo rellene. Los **sensores** devuelven señal después: los linters, el type-checker, los tests, los validadores de arquitectura, los revisores basados en modelo. Sin guías, el arnés detecta el mismo fallo una y otra vez; sin sensores, nadie comprueba que las reglas escritas se cumplen.

Los sensores se ordenan por coste. Un type-checker responde en segundos y no se equivoca sobre lo que comprueba; un revisor basado en un modelo tarda, cuesta y acierta a veces. El arnés coloca sus controles de más determinista y barato a más inferencial y caro. Y el código del repositorio condiciona cuánto detecta cada uno: tipado estricto, límites de módulo claros y builds reproducibles multiplican lo que cada control detecta. Preparar el repositorio es el primer paso del arnés.

Cuando quien programa es un agente, los tests cambian de función: pasan a ser la señal que el agente recibe en cada iteración y que determina si continúa o corrige. Eso pide suites rápidas, mensajes de fallo que expliquen y una cobertura que no dependa de quien escribió el código, porque el agente que lo escribió tiende a escribir tests que lo confirman. Por encima de los tests están los **evals**: tareas representativas con criterio de éxito comprobable, que corres contra cada cambio del arnés, del prompt o del modelo. Sin esa suite no hay forma de saber si un cambio en el arnés, en el prompt o en el modelo ha empeorado el resultado.

4.3 El radio de daño se decide antes

Todo lo anterior asume que el agente se equivocará. Permisos, aislamiento de proceso, red restringida, secretos fuera de su alcance y techos de presupuesto definen cuánto puede romper cuando lo haga. Se dimensionan según lo que el error puede costar, y crecen con la autonomía: lo que apruebas turno a turno puede ejecutarse en tu máquina; lo que corre solo de noche va en un sandbox con red restringida.

Lo que el agente verifica en su máquina es cortesía. Lo que el pipeline de integración continua verifica es contrato, porque se ejecuta fuera del alcance del agente y este no puede alterar su resultado. El arnés es también el único sitio donde se separa lo que escribiste tú de lo que venía dentro de una incidencia, un README de dependencia o una respuesta de una herramienta, porque el modelo lee una única secuencia y no distingue el origen de cada fragmento. Todo input es instrucción hasta que se demuestre lo contrario, y esa separación se hace fuera del modelo.

4.4 Lo que el arnés no decide

El arnés externaliza la mecánica de verificar. El criterio sobre qué construir, qué aceptar y qué riesgo asumir sigue siendo tuyo, y un arnés perfecto al servicio de un criterio ausente solo produce basura verificada. Y un arnés que solo entiende quien lo escribió se degrada; para evitarlo, se versiona con el código y cada regla lleva al lado el fallo que la motivó.



Harness Engineering. Calidad y seguridad del código programando en equipo con agentes. Segundo volumen de la serie: desarrolla esta capa en quince capítulos, de la anatomía del arnés a la fábrica montada de punta a punta, pasando por tests, evals, permisos, hooks, integración continua, observabilidad e input no confiable. Es la continuación práctica de Loop Engineering: donde aquel describe la idea, este la implementa. Disponible en librosdeia.com.

SECCIÓN 5

El grafo: precedencia, ramas y reanudación

Una incidencia entra en el sistema con cinco requisitos enumerados, y el paso que implementa abre un pull request que compila, pasa la integración continua en verde y cubre dos de los cinco. El agente tenía el repositorio, las instrucciones y acceso al gestor de incidencias, así que el arnés queda descartado. La condición de parada existía. Lo que fallaba era que el paso que redactó la especificación escribió los cinco requisitos en un campo que la transición hacia el paso de implementación no transportaba. El agente recibió el título de la incidencia y los dos requisitos que cabían en él, y rellenó los otros tres con conjeturas.

Ese fallo pertenece al **grafo**, la capa que decide a dónde va el trabajo cuando un paso termina. Puede ir al siguiente especialista, volver atrás porque la revisión lo ha rechazado, pararse a esperar la aprobación de una persona o no ir a ningún sitio porque ya está hecho. Alguien tiene que tomar esa decisión, y si no la has escrito, el modelo genera sobre la marcha una transición que nadie ha validado.

5.1 Seis decisiones que no caben en las otras capas

- **Precedencia.** Qué paso espera a cuál. El despliegue no arranca hasta que la revisión está aprobada.
- **Concurrencia.** Qué trabajo es independiente de verdad. La documentación y los tests pueden correr a la vez; dos agentes editando el mismo módulo, no.
- **Espera.** Dónde se vuelven a juntar las ramas que abriste, y qué haces con la que falla o con la que tarda diez veces más que las demás.
- **Firma.** Qué paso no se ejecuta sin que una persona lo apruebe. Normalmente, el que no se puede deshacer.
- **Retorno.** A dónde vuelve el trabajo cuando la verificación lo rechaza, que casi nunca es al principio.
- **Reanudación.** Desde qué punto se retoma una ejecución que se cayó a mitad, para no repetir lo que ya estaba hecho.

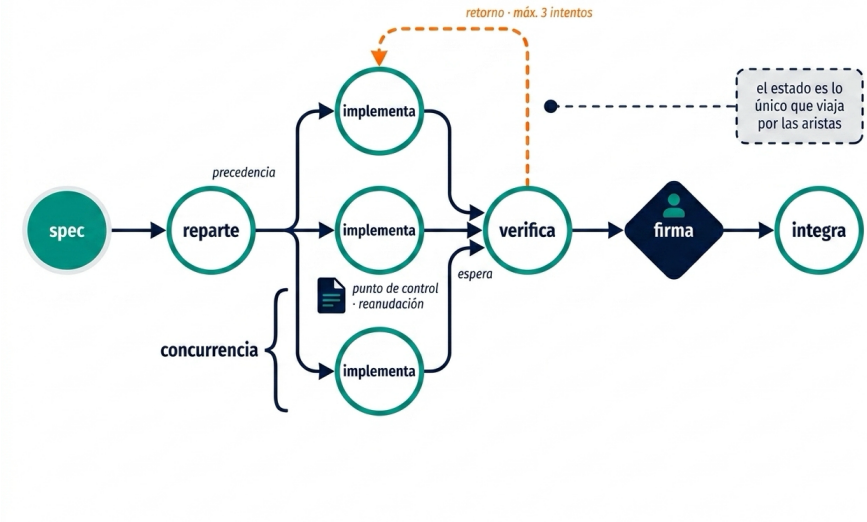


Figura 5.1: Un grafo de entrega con las seis decisiones señaladas: precedencia, concurrencia, espera, firma, retorno y reanudación.

El vocabulario para escribir esas decisiones cabe en tres palabras. Los **nodos** son los pasos: una función determinista, una llamada al modelo, un subagente especialista o una persona. Las **aristas** son las transiciones permitidas. El **estado** es lo único que viaja entre ellos, y se declara como un contrato: qué campo escribe cada nodo y qué campo lee.

> **El fallo que se atribuye al modelo.** Casi todo fallo de un sistema con varios agentes que parece «el modelo se ha despistado» es una arista que no transportaba el dato. Antes de tocar un prompt, comprueba qué campo leía el paso y qué paso lo escribía.

5.2 Cuándo se justifica un grafo

Ramificación real, aprobaciones, relevos entre especialistas, trabajo en paralelo y caminos de recuperación justifican un grafo. Un agente con cuatro herramientas y una condición de parada decente, no. Un grafo prematuro se reconoce por el fichero: seis nodos y cuatro campos de estado para un proceso que resolvía un solo agente. Al mes, cada cambio obliga a editar la topología y el equipo empieza a hacer las cosas por fuera para ahorrarse el trámite.

Cuando el grafo sí se justifica, cada nodo se elige por su naturaleza. Lo que se puede comprobar con código va en una función determinista; lo que exige juicio semántico va

en el modelo; lo que concentra consecuencia va en una persona. Poner un modelo donde bastaba un condicional es un error frecuente y de los que más cuestan corregir.

5.3 Ciclos con techo y sistemas que se reanudan

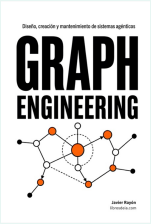
Los grafos de producción tienen ciclos. Reintentar, pedir aclaración, volver a validar y esperar a una persona son bucles legítimos, y por eso un grafo acíclico puro no describe un sistema real. La regla es que todo ciclo lleva techo, contador y destino al agotarse. De las tres cosas, la que más se olvida es la tercera: poner un máximo de tres intentos sin decir a dónde va el trabajo que no se resolvió deja el sistema parado en un sitio que nadie declaró.

El enrutado, igual que la parada del loop, se decide con evidencia. La arista condicional se dispara con un test que pasa, un esquema que valida, una puntuación de eval o una firma humana. Enrutar porque «el nodo dice que está listo» reproduce en la topología el peor vicio del loop.

Y un sistema que no se reanuda es una demostración. Los procesos se caen, las credenciales expiran y las aprobaciones tardan tres días. Los puntos de control, la idempotencia de cada nodo con efectos externos y la reanudación desde el último estado válido son lo que separa un grafo operable de un guion largo. Por la misma razón, un nodo humano lleva plazo y política de vencimiento.

5.4 Lo que el grafo necesita del arnés y del loop

El grafo corre dentro del arnés y necesita de él identidad por nodo, presupuesto por rama y trazas correlacionadas, porque lo que no se atribuye por nodo no se gobierna. De cada nodo necesita un loop con condición de parada, porque un paso sin ella bloquea a todos los que dependen de él. La estructura sustituye vigilancia, y no criterio: qué merece pasar por el grafo sigue siendo tuyo.



Graph Engineering. Diseño, creación y mantenimiento de sistemas agénticos con Claude Code. Tercer volumen de la serie: desarrolla esta capa en doce capítulos, de las fronteras de nodo y el esquema de estado a la durabilidad, los gates humanos y el coste por nodo. Incluye graphspec, la herramienta open source que declara, dibuja y valida la topología desde un fichero YAML. Disponible en librosdeia.com.

SECCIÓN 6

La spec: el criterio por escrito antes de construir

Treinta pantallas aprobadas por encima en una semana dejan treinta decisiones que nadie tomó del todo. Cada una llegó en dos minutos, compilaba y se parecía bastante a lo pedido, así que se aprobó y se pasó a la siguiente. Al final de la semana, el correo de confirmación se envía dos veces porque un agente añadió un reintento que nadie pidió, y el formulario pide el teléfono porque otro lo copió de un ejemplo que tenía en el contexto.

A esa forma de trabajar se la llama *vibe coding*. Alguien mantiene la intención en la cabeza, la suelta a trozos en la conversación y aprueba turno a turno. Funciona mientras el trabajo es pequeño y la persona está delante, y deja de funcionar en cuanto el agente corre horas sin supervisión, porque cada hueco de criterio que encuentra lo rellena con una conjetura plausible. En una fábrica, ese modo de trabajar ni siquiera es una opción: nadie está delante para aprobar cada turno.

6.1 El criterio pasa al principio

La alternativa lleva el criterio al principio. Antes de que el agente escriba nada, las decisiones que condicionan el resultado se escriben en un documento que el agente recibe como entrada y contra el que se comprueba el resultado después. Eso es el desarrollo dirigido por especificaciones, y hacerlo con IA de por medio tiene su propio método.

Mientras lo caro fue escribir el código, el criterio se aplicaba tecleando, decisión a decisión, y el documento previo era una aproximación que ya se corregiría. Con un agente que escribe el código, el documento previo pasa a ser lo caro y lo determinante. Un agente puede generar diez versiones de la misma pantalla en el tiempo que tardas en explicar cuál querías. Lo que escasea es la explicación.

La IA automatiza la generación. Lo que no automatiza es la decisión, porque le falta el contexto: que ese servicio se cae los lunes, que el equipo no domina esa tecnología, que

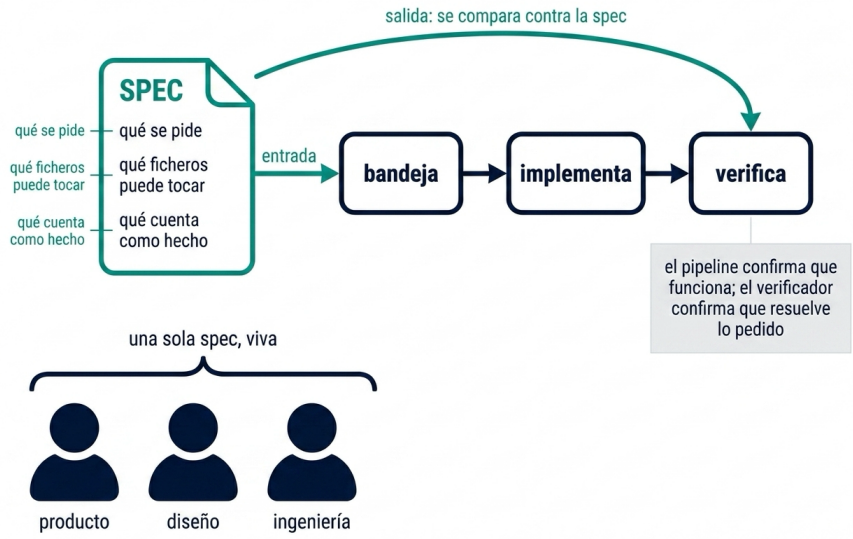


Figura 6.1: La spec entra en la bandeja con la tarea y vuelve a aparecer a la salida, cuando el verificador compara el resultado contra ella.

a este usuario le importa más la velocidad que las funciones. Esa información vive en tu cabeza y en las conversaciones del equipo, y cuando falta, el modelo rellena el hueco con una salida tan fluida como la acertada, e imposible de distinguir de ella sin la spec.

6.2 Tres propiedades de una spec para agentes

- **Es una sola.** Los criterios de producto, diseño e ingeniería conviven en un artefacto y sus contradicciones se resuelven antes de llegar a la máquina, en lugar de repartirse entre tres documentos que obligan al agente a elegir.
- **Es ejecutable por un agente.** Está escrita con el nivel de precisión que necesita alguien que no va a preguntar. Cada decisión que condiciona el resultado aparece, y lo que se deja abierto se deja abierto a propósito.
- **Está viva.** Cambia cuando el producto cambia, con el registro de qué se decidió y por qué. Una spec que describe la versión del primer día deja de gobernar nada en la tercera semana.

Mantener eso a mano no es viable. La parte pesada, que consiste en redactar, auditar huecos, detectar contradicciones y propagar un cambio por todo el documento, la hace la propia IA, con prompts y reglas.

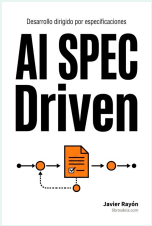
6.3 La spec dentro de la fábrica

En la cadena de una fábrica, la spec aparece en dos sitios. A la entrada, porque cada tarea que entra en la bandeja necesita una spec mínima.

La spec mínima de una tarea. Qué se pide, qué ficheros es razonable que toque y qué criterio de éxito comprobará el verificador más adelante. Una tarea sin el tercer dato es una idea a medio formular que alguien tiene que concretar antes de meterla en la bandeja.

Y a la salida, porque el pipeline confirma que el código funciona según sus propios sensores, y eso es distinto de confirmar que resuelve la tarea que pedía la spec. Si la tarea pedía validar fechas sin añadir dependencias nuevas, el pipeline confirma que los tests pasan. El verificador comprueba que la solución no metió una librería externa y que el pull request no tocó ningún módulo que la spec no mencionaba.

Es una capa que gana peso con cada versión del modelo, porque cuanto mejor ejecuta la máquina, más caro sale haberle pedido lo que no era.



AI Spec-driven. Desarrollo dirigido por especificaciones para ingenieros, product managers y product designers. Cuarto volumen de la serie: desarrolla esta capa en quince capítulos, con la anatomía de una spec viva, cómo se arranca desde cero, cómo se funden los tres criterios, cómo se audita hasta dejarla completa y consistente, cómo se mantiene, y cómo se dirige y se revisa al agente desde ella. El libro no contiene código; contiene prompts y reglas. Disponible en librosdeia.com.

SECCIÓN 7

La cadena completa de la fábrica

Puedes tener las cuatro capas construidas y aun así no atreverte a dejar el sistema solo ocho horas, porque nadie las ha conectado todavía. Esta sección monta la cadena entera y señala, en cada eslabón, qué capa lo sostiene. Es la misma cadena que *Loop Engineering* describe como idea y que *Harness Engineering* construye pieza a pieza en su último capítulo.

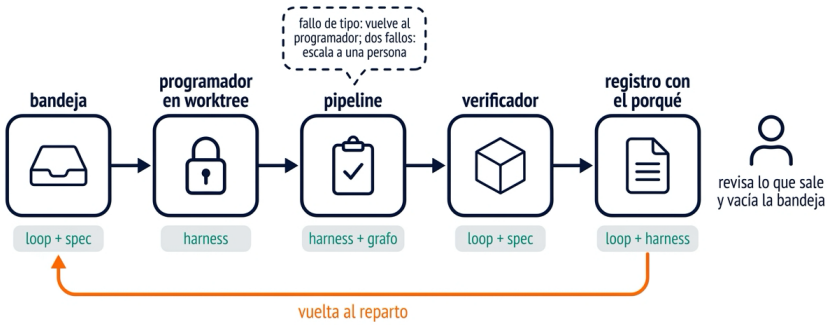


Figura 7.1: La cadena completa: reparto, implementación aislada, pipeline, verificación contra la spec y registro, con la capa responsable de cada eslabón.

7.1 Una bandeja para repartir el trabajo

Sin un mecanismo que reparta trabajo, cada tarea nueva depende de que alguien se acuerde de lanzarla, y eso ya no es una fábrica. La bandeja es un backlog priorizado donde cada entrada trae, además de la descripción, un criterio de qué cuenta como terminado. Un loop la recorre, saca la siguiente tarea que cumple sus condiciones de entrada y la asigna a un agente que la implementa. Cada entrada necesita la spec mínima de la sección anterior, y esa exigencia a la entrada es lo que permite que el resto de la cadena funcione sin nadie mirando. Capa responsable: el loop reparte; la spec fija qué entra.

7.2 Un programador en un directorio aislado

El primer agente de la cadena trabaja en un directorio de trabajo propio, aislado según el perfil de permisos y sandbox que corresponde a una tarea desatendida. Ese aislamiento es lo que limita el daño cuando el agente falla. Dentro, el programador tiene el arnés alrededor: el repositorio preparado para ser gobernado, la memoria del proyecto con las convenciones que siempre hacen falta y los tests que le devuelven señal en cada iteración. En la práctica, un programador sin ese entorno produce un diff parecido al de uno que sí lo tiene. La diferencia aparece tres pasos después, cuando ese diff pasa por controles que no atienden a explicaciones. Capa responsable: el harness.

7.3 El pipeline como autoridad

El programador termina y abre un pull request. Lo que ejecutó en su directorio adelantaba el diagnóstico; el contrato real lo impone el pipeline, donde los mismos sensores se ejecutan de nuevo con autoridad y detrás vienen los controles caros que solo tienen sentido tras la integración.

La política de fallo determina qué ocurre cuando algo no pasa. Un fallo de tipo o de linter vuelve al programador con el log entero y sin intervención humana, porque es el tipo de error que el agente corrige con la señal que ya tiene. Un fallo repetido tras dos intentos, o uno en un control que exige juicio, escala a una persona, porque insistir a ciegas gasta presupuesto sin mejorar el resultado. Capa responsable: el harness fija los controles; el grafo escribe el retorno, el techo y la escalada.

7.4 Un verificador contra la spec

El agente que implementó una tarea tiende a aprobar su propio resultado: los mismos huecos que dejó en el código los deja en la revisión. Por eso la fábrica añade una capa que un flujo de trabajo humano normal no necesita: un segundo agente que revisa

el resultado contra la spec original, con un radio de permisos más estrecho, porque le basta con leer el diff, pasar los sensores baratos y comparar.

Varía algo entre programador y verificador, aunque sea solo el prompt de sistema o la versión del modelo. Si los dos comparten modelo y prompt, comparten también los fallos sistemáticos. Basta con que el verificador reciba la tarea desde la spec, sin el razonamiento intermedio del programador en su contexto. Cuando rechaza un resultado, la tarea vuelve a la bandeja con el motivo anotado. Capa responsable: el loop separa a quien hace de quien juzga; la spec es el criterio.

7.5 Un registro con el porqué

Si nadie anota por qué se tomó cada decisión, la tarea siguiente repite el diagnóstico de la anterior. Antes de que la bandeja seleccione la tarea siguiente, el resultado se escribe en un sitio compartido: qué se hizo, qué alternativa se descartó y por qué, qué obstáculo detuvo el avance si lo hubo. Esa misma anotación alimenta la telemetría: tarea, modelo, turnos, tokens, coste, resultado del pipeline, veredicto del verificador y destino final del código. Con ese registro acumulado, la fábrica empieza a mostrar tendencias, como el coste por pull request fusionado o qué tipo de tarea genera más rechazos. Capa responsable: el loop guarda el estado en disco; el harness mide.

Ese es el ciclo completo. Reparto, implementación con arnés, verificación por pipeline, verificación por spec, registro con el porqué, y vuelta al reparto con lo aprendido ya anotado.

7.6 Tres reglas para cualquier fábrica

La fábrica produce más de lo que una persona puede revisar. Ese es a la vez su propósito y su peligro, y de ahí salen tres reglas.

- **Lo que no puedes verificar, no lo automatices.** Antes de añadir una etapa, pregúntate quién confirmará que su «hecho» es cierto, y cómo. Un loop cuya salida nadie sabe juzgar es deuda que llega más rápido.
- **La fábrica anota por qué, además de qué.** Un pull request sin el razonamiento detrás es opaco. Esa nota es lo que hace revisable lo que la fábrica produce.
- **La persona se reserva el juicio y cede la mecánica.** Lo que sueltas a la fábrica es el trabajo repetitivo y verificable. Retienes la decisión de qué construir, qué compromiso aceptar y qué riesgo no correr.

La misma fábrica, dos resultados. Dos ingenieros pueden montar la misma fábrica y obtener resultados opuestos. Uno la usa para avanzar más deprisa sobre una materia que domina: lee cada pull request, entiende cada cambio, y la fábrica le ahorra el tecleo mientras él conserva el criterio. El otro la usa para saltarse esa materia, y acepta lo que sale con que haya pasado la verificación. La fábrica produce igual de bien para los dos. A medio plazo, uno acumula sistema y comprensión, y el otro acumula código que no entiende.

El equipo de OpenAI que, según el post «Harness engineering» citado en la sección 2, construyó en unos cinco meses un producto de en torno a un millón de líneas sin escribirlas a mano no tenía un modelo distinto del tuyo. Tenía un arnés mejor, construido a fuerza de convertir cada fallo temprano en una pieza que impidiera su repetición, y una forma de trabajar en la que el ingeniero diseñaba el entorno, especificaba la intención y construía los bucles de realimentación. Esa es la fábrica.

SECCIÓN 8

Diagnóstico y orden de lectura

Ante un fallo se responden cuatro preguntas, siempre en el mismo orden, y el orden importa tanto como las preguntas. Se empieza por abajo, por el harness, porque un fallo de harness se disfraza de fallo de grafo con mucha facilidad: una herramienta que le falta a un paso se ve igual que una rama mal trazada. El camino contrario casi no ocurre. Si empiezas por arriba, acabas rediseñando la topología para tapar un fichero de configuración desactualizado.

Orden	Pregunta	Si la respuesta es no…
1	¿Tenía el agente delante todo lo necesario para hacerlo bien?	Es harness. Construye la pieza que falta y para aquí.
2	¿Había una prueba mecánica de que había terminado, juzgada por alguien distinto?	Es loop. Escribe la condición de parada y separa a quien juzga.
3	¿Estaba escrito qué paso podía venir después de cuál, y qué viaja entre ellos?	Es grafo. Declara la topología y compruébala.
4	¿Estaba escrito qué había que construir y qué cuenta como hecho?	Es spec. Concreta la tarea antes de volver a meterla en la bandeja.

Una capa cada vez. Cuando hay dos capas implicadas, y suele haberlas, se arregla la de abajo y se vuelve a medir antes de tocar la otra. Si cambias las dos a la vez, el sistema mejora y te quedas sin saber cuál de los dos cambios lo consiguió. Para comprobar que has acertado, provoca el fallo otra vez y mira si el arreglo lo impide: estos sistemas tienen variabilidad de sobra para que cualquier cambio parezca funcionar durante unas horas.



Figura 8.1: Las cuatro preguntas se recorren de abajo arriba, y cada una lleva a una acción distinta.

El diagnóstico por defecto, «el modelo se ha despistado», es el que más tiempo cuesta, porque manda a alguien a ajustar prompts y a probar modelos durante semanas para arreglar una arista o una línea de configuración. Antes de aceptarlo, pasa las cuatro preguntas en orden.

8.1 Doce comprobaciones antes de dejar el sistema solo

Cada línea se responde con una comprobación ejecutada en el momento, y lo recomendable es pasarlas con alguien que no haya montado el sistema.

Harness.

- Las instrucciones del repositorio describen el proyecto de hoy, y lo has verificado abriéndolas.
- Cada paso tiene sus herramientas y permisos acotados a su mandato.
- Los pasos paralelos trabajan aislados, con directorio y rama propios.
- Las comprobaciones que autorizan transiciones devuelven un resultado que una máquina puede leer.

Loop.

- Cada paso tiene una condición de parada que sale de ejecutar algo.
- Quien juzga no es quien produjo el trabajo, ni en agente ni en modelo.
- Todo ciclo lleva techo, contador y destino al agotarse.

- Lo que hay que conservar entre pasos está en disco, y no en una conversación.

Grafo y spec.

- Todo paso tiene camino hacia algún final que el equipo reconoce.
- Los campos de estado tienen propietario y lectores declarados.
- El paso humano lleva plazo y política de vencimiento.
- Cada tarea de la bandeja trae qué se pide, qué puede tocar y qué cuenta como hecho.

8.2 Por dónde empezar

Si el sistema aún no existe, el orden de construcción es el mismo que el de diagnóstico. Primero el harness, porque sin él ningún loop tiene con qué trabajar. Después el loop, con su condición de parada y su verificador separado. La spec entra en cuanto haya una bandeja, porque es lo que decide qué merece entrar en ella. Y el grafo, el último, solo cuando aparezcan la ramificación real, las aprobaciones o el trabajo en paralelo que lo justifican.

8.3 Qué cubre cada libro de la serie

Los cuatro volúmenes de la serie AI Engineering desarrollan cada capa con la práctica que aquí no cabe, y se leen por separado, en el orden que marque tu sistema.

Loop Engineering monta el ciclo que funciona sin ti. Es el punto de entrada si todavía sostienes la herramienta turno a turno.

Harness Engineering construye el entorno que hace fiable lo que ese ciclo produce. Es el volumen para quien ya tiene loops y no se fía de lo que sale.

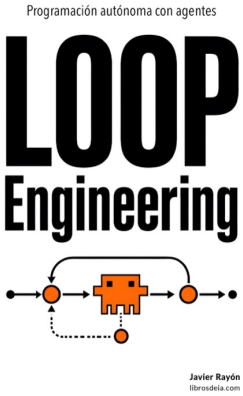
Graph Engineering reparte el trabajo entre nodos cuando una sola tarea deja de bastar. Va dirigido a quien coordina varios agentes y ya nadie puede explicar el sistema entero.

AI Spec-driven se ocupa del nivel de arriba, el que decide qué hay que construir. Lo lee con provecho quien ya produce deprisa y quiere dejar de producir lo que no era.

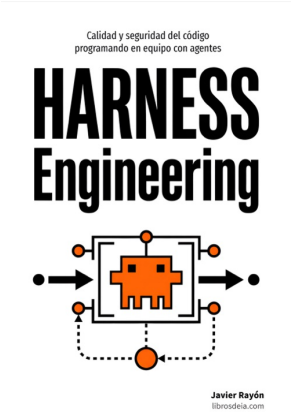
Los cuatro están disponibles en librosdeia.com, junto con la colección Claude Code, que enseña la herramienta sobre la que se apoyan los ejemplos de la serie. Ninguno es necesario para aplicar el diagnóstico de esta sección.

La serie AI Engineering

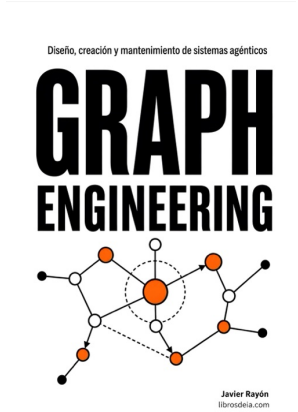
Los cuatro volúmenes de la serie, disponibles en librosdeia.com.



Loop Engineering



Harness Engineering



Graph Engineering



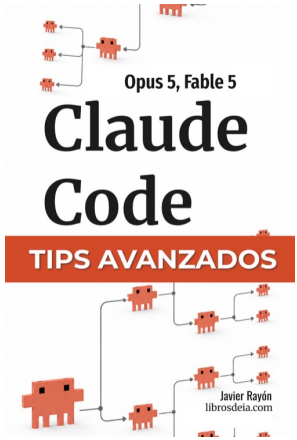
AI Spec-driven

La colección Claude Code

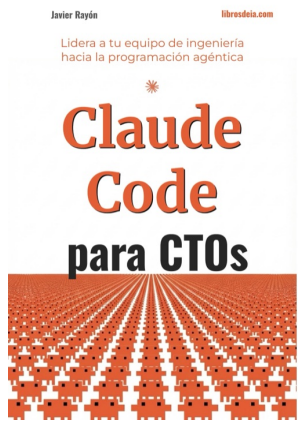
Los títulos de la colección, disponibles en librosdeia.com.



Primeros Pasos con Claude Code



Claude Code. Tips Avanzados para Power Users



Claude Code para CTOs