UNA GUÍA DE CAMPO

La Tripulación Agéntica

Ingeniería en la era de los agentes de IA

Rasmus Bornhøft Schlünsen

Marzo 2026

rev 2

Prólogo

Era un martes por la noche, en algún momento del año pasado. Mis hijos estaban dormidos, y yo miraba un script de migración que llevaba temiendo toda la semana — ese tipo de reorganización tediosa, tabla por tabla, que se come un día entero si eres cuidadoso y destroza producción si no lo eres. Por capricho, le describí el problema a un agente. El esquema aquí, las restricciones allá, cuidado con esta clave foránea. Luego pulsé enter y fui a preparar un té.

Cuando volví, el script estaba hecho. No un borrador. *Hecho.* Casos límite correctos, lógica de rollback, comentarios que yo mismo habría escrito. Me quedé sentado un buen rato, el té enfriándose, sintiendo dos cosas a la vez: asombro genuino — y un temor silencioso y creciente.

Porque esa migración era *mi* cosa. Yo era la persona del equipo que podía mantener todo el esquema en la cabeza, que sabía qué joins estaban malditos, que podía escribir el SQL cuidadoso a mano. Quince años de memoria muscular, y un LLM acababa de igualarlo en cuatro minutos mientras yo hervía agua.

Quiero ser honesto contigo: esa noche no dormí bien. Me quedé en la cama ejecutando el mismo bucle que todo ingeniero que conozco ha ejecutado. *¿Para qué sirvo ahora? ¿Qué pasa con el oficio que pasé media vida construyendo? ¿Estoy entrenando a mi reemplazo?*

Me llevó meses — y mucho construir, fallar y reconstruir con estas herramientas — encontrar la respuesta. Y la respuesta me sorprendió. El oficio no está muriendo. Está *mudando*. La cáscara exterior — las pulsaciones de teclas, la sintaxis, el boilerplate — esa parte se está cayendo. Pero el animal debajo, la parte que sabe *qué* construir y *por qué*, que huele una mala abstracción a tres archivos de distancia, que puede mantener todo un sistema en la mente y sentir dónde es frágil — esa parte está más viva que nunca.

No estamos siendo reemplazados. Estamos siendo promovidos. De mecanógrafos a pensadores. De escribir código a dirigirlo — orquestar, revisar, dar forma. Las habilidades que te hicieron buen ingeniero — pensamiento de sistemas, gusto, criterio, el instinto por la simplicidad — esas son *todo el juego* ahora, no solo el ruido de fondo.

Pero nadie nos dio un manual para esta transición. Es desordenada e incómoda y a veces humillante. Escribí este libro porque estoy viviéndola, y tengo la sensación de que tú también. Estas páginas son todo lo que he aprendido sobre trabajar *con* los agentes en lugar de contra ellos — o peor, fingir que no existen.

Si alguna vez has visto a una IA escribir código que se parecía al tuyo y has sentido un vuelco en el estómago, este libro es para ti. Sigue leyendo. Se pone mejor — y más extraño — de lo que crees.

Rasmus Bornhøft Schlünsen
Marzo 2026

Contenidos

1 Introducción

El año pasado vi a una desarrolladora junior de mi equipo — dos años de experiencia, todavía nerviosa en las revisiones de código — entregar un endpoint de API completo en cuarenta y cinco minutos. Modelo de datos, validación, manejo de errores, tests, documentación. El código era limpio. Los tests eran exhaustivos. El PR pasó la revisión al primer intento.

A mí me habría llevado una hora hacer el mismo trabajo. Y llevo veinte años en esto.

Ella no tecleó la mayor parte. Describió lo que necesitaba, apuntó un agente al código base y lo guió hasta la línea de meta. Su habilidad no estaba en escribir el código — estaba en saber qué pedir, reconocer cuándo el resultado era bueno y detectar el único caso límite que el agente pasó por alto. Estaba haciendo *ingeniería*. Solo que no de la forma en que yo aprendí a hacerla.

Esa noche volví a casa y me senté con una pregunta incómoda: si la brecha entre veinte años de experiencia y dos años de experiencia acaba de estrecharse considerablemente, ¿qué aporto yo exactamente?

La respuesta, al final me di cuenta, es todo lo que no es teclear. Pero llegar a esa respuesta me llevó meses, muchos errores y este libro.

1.1 El Suelo Se Mueve

Durante veinte años, ser ingeniero de software significaba una cosa: abres un editor, escribes código, lo despliegas. Las herramientas cambiaban — de Vim a VS Code, de SVN a Git, de bare metal a Kubernetes — pero el bucle fundamental seguía siendo el mismo. Tú, un teclado y un problema.

Ese bucle se está rompiendo. Y se está rompiendo rápido.

Los agentes de IA no se limitan a autocompletar tu código. Leen todo tu código base, razonan sobre la arquitectura, hacen cambios en docenas de

archivos, ejecutan tus tests e iteran sobre los fallos — todo sin que toques el teclado. No están reemplazando el editor. Están reemplazando el *flujo de trabajo*. El ingeniero que solía pasar el 80% de su tiempo tecleando ahora pasa el 80% de su tiempo pensando, revisando y dirigiendo.

Algunos ingenieros están prosperando con este cambio. Entregan más, con mayor calidad, y te dirán que disfrutan de su trabajo más que en años. Otros están frustrados, escépticos o silenciosamente aterrados de que el oficio que pasaron una década dominando se esté evaporando bajo sus pies.

Ambas reacciones son racionales. La verdad está en algún lugar del incómodo punto medio.

1.2 Por Qué Este Libro

Porque nadie nos dio un manual.

Las herramientas aparecieron rápido — Copilot, luego Claude, luego agentes que pueden ejecutar tareas de extremo a extremo de forma autónoma — y todos estamos descubriéndolo sobre la marcha. Busqué el libro que me dijera cómo trabajar realmente con estas cosas. No el discurso de marketing. No el artículo académico. No el hilo de Twitter de alguien que lo probó durante veinte minutos. Quería la guía honesta de ingeniería — escrita por alguien que despliega código en producción y ha visto a los agentes hacer cosas brillantes y cosas catastróficamente estúpidas a partes iguales.

Ese libro no existía. Así que lo escribí.

Lo escribí porque yo también estaba perdido. Era el ingeniero senior que no podía entender por qué el agente seguía reescribiendo toda mi biblioteca de componentes cuando le pedía que arreglara el color de un botón. Era el tipo que quemó 500.000 tokens en una tarea que debería haber llevado diez minutos, porque no sabía cómo poner límites. Cometí todos los errores de este libro antes de aprender a evitarlos.

Esta es la guía que me habría gustado que alguien me diera.

1.3 Para Quién Es

Eres ingeniero de software. Has entregado cosas reales. Sabes lo que se siente un incidente en producción a las 2 de la madrugada. No le tienes miedo al terminal.

Pero últimamente algo se siente diferente. Quizá has probado herramientas de codificación con IA y las has encontrado impresionantes pero caóticas — como hacer pair programming con alguien que es increíblemente rápido pero no tiene concepto de alcance. Quizá has visto a un desarrollador con dos años de experiencia entregar una funcionalidad completa en una tarde usando asistencia de agentes, y te hizo sentir algo que no esperabas. Quizá estás entusiasmado pero no sabes por dónde empezar. Quizá eres escéptico y quieres que alguien te convenza con sustancia, no con bombo publicitario.

Este libro es para ti. Asume que sabes programar. Asume que llevas tiempo en esto. Te encuentra donde estás.

1.4 Cómo Leer Este Libro

Esto no es un manual de referencia. Es un viaje, y está estructurado como tal.

Empezamos entendiendo qué está cambiando realmente y por qué — el cambio en cómo se construye el software, no solo las herramientas sino el *pensamiento*. Luego entramos en qué son realmente los agentes, despojados del lenguaje de marketing, para que tengas un modelo mental que se sostenga cuando las herramientas cambien el próximo trimestre.

A partir de ahí, nos ensuciamos las manos. Aprenderás cómo funcionan los agentes en el terminal, cómo establecer barandillas para que no destrocen tu código base, cómo cambian los flujos de trabajo de Git cuando los agentes hacen commits, y por qué el sandboxing no es opcional. Profundizaremos en los tests como el bucle de retroalimentación que hace que los agentes sean *fiables* en lugar de solo rápidos, y las convenciones como el arma secreta que la mayoría de la gente pasa por alto.

Luego iremos más profundo — modelos locales versus comerciales, ingeniería de prompts como disciplina real, y orquestación multi-agente. Veremos historias de guerra: fallos reales, lecciones reales, tejido cicatricial real. Hablaremos de cuándo *no* usar agentes, porque saber cuándo soltar la herramienta es tan importante como saber cómo usarla. Y cerraremos con cómo los equipos adoptan esto sin implosionar.

Al final, no solo sabrás cómo usar agentes de IA. Sabrás cómo *pensar* sobre ellos — que es la habilidad que sobrevive cuando la generación actual de herramientas quede obsoleta.

1.5 Lo Que Este Libro No Es

Déjame ahorrarte algo de tiempo.

Esto no es un recetario de prompts. No encontrarás «50 prompts de ChatGPT para desarrolladores» aquí. Los prompts importan, y los cubrimos, pero copiar y pegar prompts sin entender el sistema debajo es una receta para una frustración costosa.

Esto no es un manifiesto del hype de la IA. No estoy aquí para decirte que los agentes reemplazarán a todos los programadores para el próximo martes. No lo harán. La brecha entre la demo y la producción es tan ancha como siempre, y alguien tiene que vigilar esa brecha.

Tampoco es una narrativa catastrofista. El encuadre de «la IA viene por tu trabajo» es perezoso y en su mayoría equivocado. Lo que viene es un cambio fundamental en *cómo* funciona el trabajo, y esa es una conversación completamente diferente.

Este es un libro de ingeniería. Para ingenieros. Escrito por alguien que pasa sus días escribiendo código con agentes y tiene el historial de git para demostrarlo. Vamos a ser prácticos, honestos y específicos. Si eso suena como lo tuyo, pasa la página.

2 Contexto

El mes pasado, llegó un bug de un cliente: los pagos estaban fallando silenciosamente para usuarios con caracteres no ASCII en su dirección de facturación. Uno complicado — del tipo que vive en la costura entre la validación de tu frontend y la codificación de caracteres de tu pasarela de pagos.

Un ingeniero de mi equipo cogió el ticket primero. Abrió su agente y escribió: «Hay un bug con los pagos para usuarios internacionales. ¿Puedes investigarlo?» El agente leyó alegremente el módulo de pagos, hizo algunas suposiciones plausibles sobre el manejo de Unicode y produjo un parche que normalizaba toda la entrada a ASCII. Habría eliminado cada acento, cada diéresis, cada carácter fuera del alfabeto inglés de la dirección de facturación de cada usuario. Una corrección técnicamente peor que el bug.

Una hora después, una segunda ingeniera tomó el mismo ticket después de que el primer intento fue rechazado en la revisión. Ella pegó el log de error del cliente, la traza de Sentry que fallaba, el código de respuesta específico de la pasarela de pagos, la sección relevante de la documentación de codificación de caracteres de la pasarela, y los tres archivos involucrados en el pipeline de facturación. Su agente diagnosticó el problema en menos de dos minutos: una cadena UTF-8 se estaba pasando a través de una función que asumía codificación Latin-1 antes de llegar a la API de la pasarela. La corrección fueron cuatro líneas. Se desplegó esa tarde.

El modelo era el mismo. El agente era el mismo. El bug era el mismo. Lo que difería era lo que cada ingeniero puso delante del agente antes de pedirle que trabajara. Uno le dio una descripción vaga y lo dejó adivinar. La otra le dio todo lo que necesitaba para *ver* el problema.

Esa diferencia — lo que el agente puede ver — es de lo que trata este capítulo.

La habilidad más importante en la ingeniería agéntica no es el prompting. Es la gestión del contexto.

Un agente de IA es tan bueno como lo que puede ver. Dale una instrucción vaga y una pizarra en blanco, y alucinará con confianza. Dale los archivos correctos, las restricciones correctas, la vista correcta del sistema — y hará cosas que parecen magia. La diferencia no es el modelo. Eres tú.

La ingeniería tradicional también tenía una versión de esto. Un ingeniero senior no solo escribía mejor código — mantenía más del sistema en su cabeza. Sabía qué archivos importaban, dónde vivían los dragones, qué abstracciones eran estructurales y cuáles eran decorativas. Ese modelo mental era el contexto, y vivía enteramente en el cerebro del ingeniero.

Ahora tienes que *externalizarlo*. Tus agentes no pueden leer tu mente. Leen archivos, variables de entorno, logs de errores y lo que pongas delante de ellos. El oficio de la ingeniería agéntica es aprender qué sacar a la superficie, cuándo y cómo.

2.1 La Ventana de Contexto Es Tu Banco de Trabajo

Piensa en la ventana de contexto como un banco de trabajo físico. Tiene espacio limitado. No puedes volcar todo tu código base encima y esperar buenos resultados. En su lugar, colocas las piezas que importan para *esta* tarea: los archivos de código relevantes, el test que falla, el esquema, quizá un fragmento de documentación.

Un buen ingeniero agéntico cura el contexto como un buen cirujano dispone los instrumentos. Nada innecesario. Todo al alcance.

Esto significa desarrollar instintos para preguntas como:

- ¿Qué necesita ver el agente para entender esta tarea?
- ¿Qué lo confundirá si lo incluyo?
- ¿El contexto que estoy proporcionando es *actual*, o le estoy dando información obsoleta?

2.2 El Impuesto de la Ventana de Contexto

Cada token que pones en una ventana de contexto te cuesta dos veces: una en dinero, otra en atención.

La parte del dinero es sencilla. Las llamadas a la API se cobran por conteo de tokens. Vuelca todo tu código base en el contexto y estarás quemando dólares en cada interacción. Pero el coste más insidioso es la degradación de la atención. Los modelos de lenguaje no tratan todos los tokens por igual — hay un fenómeno bien documentado en el que la información en medio de un contexto largo recibe menos peso que la información al principio o al final. Cuanto más metes, más probable es que el modelo pase por alto lo que realmente importa.

Aprendí esto por las malas. Al principio, pensaba que más contexto siempre era mejor. Trabajando en una migración de base de datos complicada, alimenté al agente con cada archivo de migración que habíamos escrito — tres años de cambios de esquema, cientos de archivos. Mi razonamiento era sólido: el agente necesitaba entender el historial completo para escribir la siguiente migración correctamente. El resultado fue una migración que duplicó una columna que ya existía, porque la migración anterior relevante estaba enterrada en medio de un contexto enorme y el modelo efectivamente la perdió de vista.

En el siguiente intento, le di solo el esquema actual, las tres migraciones más recientes y un párrafo resumiendo el historial relevante. El agente lo clavó.

Este es el impuesto de la ventana de contexto en acción. Hay un punto óptimo entre demasiado poco y demasiado, y encontrarlo es una habilidad que se desarrolla con la práctica.

Demasiado poco contexto produce alucinaciones. El agente no tiene suficiente información, así que llena los vacíos con invenciones que suenan plausibles. Le pides que arregle una función sin mostrarle el archivo, e inventa una API que no existe. Le pides que escriba un test sin mostrarle tu framework de testing, y elige Jest cuando usas Vitest.

Demasiado contexto produce confusión y desperdicio. El agente tiene la respuesta enterrada en algún lugar de la pila, pero no puede encontrarla

— o peor, encuentra información contradictoria en diferentes archivos y elige la equivocada. También estás pagando por cada token de ruido que incluiste.

El punto óptimo es contexto *curado*. No todo lo que el agente podría posiblemente necesitar, sino todo lo que realmente necesita para esta tarea específica, presentado con claridad. Piénsalo menos como llenar un archivador y más como informar a un colega antes de una reunión. No le darías todos los documentos que la empresa ha producido. Le darías las tres cosas que necesita leer y un resumen de un minuto del trasfondo.

Una heurística práctica: si estás a punto de pegar algo en el contexto, pregúntate — ¿tomará el agente una decisión diferente (mejor) por haber visto esto? Si la respuesta es no, déjalo fuera.

2.3 Local, en Sandbox, Remoto: Moviendo Tus Agentes

El contexto no es solo texto en un prompt. Es sobre *dónde* opera tu agente y a qué tiene acceso.

2.3.1 La Máquina Local

La configuración más simple: el agente se ejecuta en tu máquina, lee tus archivos, ejecuta tus comandos. Aquí es donde la mayoría empieza — herramientas como Claude Code operando directamente en el directorio de tu proyecto.

La ventaja es la inmediatez. El agente ve lo que tú ves. Puede leer tu código, ejecutar tus tests, consultar tu historial de git. El riesgo también es obvio: es tu máquina, tus credenciales, tu configuración de producción ahí mismo en `~/.env`.

2.3.2 Entornos en Sandbox

Un enfoque más disciplinado es dar al agente un sandbox — un contenedor, una VM, un worktree. Obtiene una copia del código pero no tus claves. Puede romper cosas sin romper *tus* cosas.

Esto importa más de lo que la mayoría cree. Cuando dejas que un agente itere libremente — ejecutando código, instalando paquetes, modificando

archivos — quieres que lo haga en un espacio donde un error sea barato. Un agente en sandbox es un agente sin miedo, y un agente sin miedo es uno productivo.

Los worktrees son una herramienta infravalorada aquí. Los worktrees de Git te permiten crear una copia aislada de tu repositorio en segundos. El agente trabaja en su propia rama, su propio directorio. Si el resultado es bueno, lo fusionas. Si no, eliminas el worktree y sigues adelante. Sin desorden.

2.3.3 Exploración Remota

Aquí es donde las cosas se ponen interesantes. Un ingeniero agéntico hábil no solo apunta agentes a archivos locales — enseña a los agentes a *explorar* sistemas remotos.

SSH a un servidor de staging para examinar logs. Consultar una base de datos para entender la forma de los datos reales. Hacer curl a un endpoint de API para ver qué devuelve realmente, no lo que dice la documentación. Descargar logs de contenedores de un servicio en ejecución.

El agente se convierte en tu explorador. Lo apuntas a un sistema y dices: «ve a mirar y cuéntame lo que encuentres.» Pero tienes que configurar esto. El agente necesita credenciales (con alcance limitado y temporales), acceso a la red y límites claros sobre lo que puede tocar.

Esta es una decisión de criterio — cuánto acceso dar, a qué sistemas, con qué barandillas. Demasiado poco y el agente es inútil. Demasiado y estás a un mal prompt de un incidente en producción. El ingeniero agéntico aprende a calibrar esto con el tiempo.

2.4 Alimentando el Contexto Deliberadamente

Los mejores ingenieros agénticos desarrollan hábitos alrededor del contexto:

Empieza con el error. No describas el bug — muestra al agente la traza de la pila, la salida del test que falla, la línea del log. El contexto en crudo supera al contexto parafraseado siempre.

Muestra, no cuentes. En lugar de explicar tu esquema de base de datos en prosa, dale al agente los archivos de migración o los modelos del ORM. En lugar de describir el contrato de la API, dale la especificación OpenAPI o una respuesta de curl.

Poda agresivamente. Si estás depurando un problema de renderizado, el agente no necesita ver tu middleware de autenticación. Cada archivo irrelevante en el contexto es ruido que degrada la señal.

Usa el sistema de archivos como contexto. Un proyecto bien organizado *es* contexto. Nombres de archivo significativos, estructura de directorios clara, un buen README — estos ya no son solo para humanos. Tus agentes también los leen.

Da rutas de archivos, no búsquedas del tesoro. Cuando sabes qué archivos son relevantes, dilo explícitamente. «El bug está en `src/payments/gateway.ts`, específicamente en la función `encodeAddress` en la línea 142» es infinitamente mejor que «hay un bug en algún lugar del código de pagos.» Cada minuto que el agente pasa buscando el archivo correcto es un minuto que no está dedicando al problema real — y está quemando tokens todo el rato.

Usa git blame para explicar el *por qué*. El código le dice al agente *qué* existe. El historial de Git le dice *por qué*. Cuando le pides a un agente que modifique un fragmento de código con un diseño no obvio, apúntalo al mensaje de commit relevante o al pull request. «Esta función parece rara pero se escribió así por el issue 1247 — mira el mensaje de commit en `abc123`» le da al agente la justificación que necesita para hacer cambios sin romper la intención original.

Copia y pega en lugar de parafrasear. Vale la pena repetir esto porque es el error más común que veo. Los ingenieros describen un error con sus propias palabras en lugar de pegar el error real. «El build está fallando con algún error de TypeScript sobre tipos» versus pegar la salida exacta del compilador con ruta de archivo, número de línea y código de error. Lo primero le da al agente una dirección vaga. Lo segundo le da un objetivo específico. Siempre pega la salida en crudo. Deja que el agente haga la interpretación.

Organiza tu contexto por capas. Para tareas complejas, no vuelques todo de golpe. Empieza con la visión general — qué hace el sistema, qué intentas cambiar, por qué. Luego proporciona los archivos específicos. Luego proporciona el error o el fallo del test. Esto refleja cómo informarías a un colega humano, y funciona por la misma razón: construye un modelo mental antes de entrar en los detalles.

2.5 Contexto Entre Sesiones

He aquí un problema del que nadie te advierte: las ventanas de contexto son efímeras. Cuando una sesión termina, todo lo que el agente aprendió — cada archivo que leyó, cada decisión que tomó, cada callejón sin salida que exploró — se desvanece. La siguiente sesión empieza desde cero.

Para tareas cortas, esto no importa. Pegas el error, el agente lo arregla, listo. Pero el trabajo de ingeniería real abarca días, a veces semanas. Una funcionalidad que toca doce archivos en tres servicios. Una refactorización que necesita hacerse por etapas. Una sesión de depuración donde las primeras dos horas acotan el problema y los últimos treinta minutos lo arreglan — excepto que tuviste que cerrar tu portátil entre medias.

Si no planificas los límites de sesión, desperdiciarás cantidades enormes de tiempo restableciendo contexto que el agente ya tenía. He visto a ingenieros pasar los primeros quince minutos de cada sesión re-explicando lo que le dijeron al agente ayer. Eso no es ingeniería. Es hacer de canguro.

La solución es hacer el contexto *duradero* — almacenarlo en algún lugar donde la siguiente sesión pueda retomarlo.

Deja que el código base sea la capa de continuidad. La forma más fiable de contexto entre sesiones es el propio código. Si el agente progresó ayer, ese progreso debería estar commiteado. Los buenos mensajes de commit se convierten en el rastro de migas de pan: «Refactorizado pasarela de pagos para separar paso de codificación — siguiente paso es añadir tests para entrada no ASCII.» La siguiente sesión empieza leyendo el log de git reciente, y el agente tiene una imagen clara de dónde están las cosas.

Usa archivos CLAUDE.md (o su equivalente). Muchas herramientas de agentes soportan un archivo de contexto a nivel de proyecto — un

archivo markdown en la raíz de tu repositorio que describe la arquitectura del proyecto, las convenciones y el estado actual. Este archivo persiste entre sesiones porque vive en el sistema de archivos. Actualízalo a medida que el proyecto evoluciona. Incluye cosas como: cuáles son los componentes principales, qué patrones seguir, qué está roto actualmente, en qué está trabajando el equipo. Es un documento de briefing que cada nueva sesión lee automáticamente.

Escribe resúmenes de sesión. Cuando terminas una sesión compleja, dedica sesenta segundos a que el agente resuma lo que logró, lo que queda por hacer y lo que aprendió sobre el código base. Guarda ese resumen en algún lugar — un comentario en el ticket, una nota en el proyecto, incluso un archivo de texto en el repositorio. La siguiente sesión empieza leyendo ese resumen, y has preservado horas de comprensión acumulada en unos pocos párrafos.

Haz commit temprano y a menudo. Esto se cubre en el capítulo de Git, pero vale la pena repetirlo aquí porque es fundamentalmente una estrategia de gestión de contexto. Cada commit es un punto de control que las sesiones futuras pueden referenciar. Una sesión que termina con cambios sin commitear es una sesión cuyo contexto está atrapado en una ventana de terminal que puede que no exista mañana.

Los ingenieros que manejan bien el trabajo agéntico de larga duración son los que tratan los límites de sesión como una preocupación de primera clase. No simplemente cierran el portátil — cierran el ciclo.

2.6 El Contexto Como Arquitectura

He aquí la idea más profunda: a medida que mejoras en la ingeniería agéntica, empiezas a diseñar tus sistemas *para* el contexto. Escribes mensajes de commit más claros porque los agentes los leen. Mantienes las funciones pequeñas porque los agentes trabajan mejor con archivos enfocados. Mantienes la documentación actualizada porque los agentes la tratan como la fuente de verdad.

Este no es un ajuste menor. Cambia cómo piensas sobre la estructura del código a todos los niveles.

Las funciones pequeñas son amigables con el contexto. Una función de 400 líneas requiere que el agente mantenga todo en la memoria de trabajo para hacer un cambio de forma segura. Una función de 30 líneas que hace una sola cosa es algo que el agente puede entender completamente, modificar con confianza y verificar rápidamente. El viejo consejo — «una función debería hacer una sola cosa» — siempre fue buena ingeniería. Ahora también es buena gestión de contexto. Cada vez que extraes una función con un buen nombre de una más grande, estás creando una unidad de significado con la que un agente puede trabajar independientemente.

Los nombres de archivo son navegación. Cuando un agente necesita encontrar el código que maneja la autenticación de usuarios, empieza mirando los nombres de archivo. `auth.ts` es una señal. `utils.ts` es un agujero negro. `handleStuff.js` es un callejón sin salida. La disciplina de nombrar archivos claramente — `user-authentication.ts`, `payment-gateway.ts`, `rate-limiter.middleware.ts` — ya no es solo una cortesía para futuros desarrolladores. Es un índice que los agentes usan para orientarse en tu código base sin leer cada archivo.

La estructura de directorios es documentación de arquitectura. Un directorio plano con sesenta archivos no le dice nada al agente sobre cómo está organizado el sistema. Una jerarquía clara — `src/api/`, `src/services/`, `src/models/`, `src/middleware/` — se lo dice todo. El agente puede inferir la arquitectura solo de la estructura de carpetas, sin leer una sola línea de documentación. He visto agentes navegar un monorepo bien estructurado de 10.000 archivos más efectivamente que un proyecto mal estructurado de 200, puramente porque la disposición de directorios hacía el sistema legible.

Monorepos vs. multirepos: un compromiso de contexto. En un monorepo, el agente puede ver todo — la API, el frontend, las bibliotecas compartidas, la configuración de infraestructura. Esto es potente para tareas que cruzan límites. Pero también significa que el agente podría ver *demasiado*, incorporando código irrelevante de servicios no relacionados. En una configuración multirepo, cada repositorio tiene un alcance natural — el agente ve solo el servicio en el que está trabajando. Pero las tareas entre servicios se vuelven más difíciles porque el agente no puede

referenciar fácilmente el otro lado de un contrato de API. Ningún enfoque es universalmente mejor. El punto es que tu estrategia de repositorios es una decisión de contexto, lo pienses así o no.

Los tipos son contexto. Un código base fuertemente tipado le da al agente algo que uno dinámicamente tipado no: una descripción legible por máquina del contrato de cada función. El agente puede mirar una firma de función y saber exactamente qué entra y qué sale, sin leer la implementación. TypeScript, Rust, Go — estos lenguajes llevan contexto estructural en sus sistemas de tipos. Python y JavaScript dejan al agente adivinando a menos que hayas escrito docstrings exhaustivos o type hints. Esto no es un argumento a favor de un lenguaje sobre otro. Es una observación de que los sistemas de tipos hacen doble función en la era agéntica: detectan bugs *y* comunican intención.

La documentación es la fuente de verdad (sea precisa o no). Los agentes tratan tu README, tus docs de API, tus comentarios inline como autoritativos. Si tu documentación dice que la API devuelve un campo `user_id` pero la respuesta real devuelve `userId`, el agente escribirá código contra la documentación y producirá un bug. La documentación obsoleta siempre fue una molestia. Con los agentes, es una fuente activa de defectos. El listón de precisión de la documentación sube — no porque los agentes necesiten mejor documentación que los humanos, sino porque los agentes seguirán documentación mala más fielmente de lo que lo haría un humano.

La forma en que estructuras tu código, tus repositorios, tu infraestructura — todo se convierte en parte del contexto que proporcionas a tu tripulación. Los ingenieros que entienden esto temprano construirán sistemas que no solo son mantenibles por humanos, sino *navegables* por agentes. Y con el tiempo, esa navegabilidad se acumula — cada nueva sesión de agente se beneficia de cada decisión estructural que tomaste antes.

3 ¿Qué Es un Agente?

La palabra «agente» se usa mucho. Se aplica a todo, desde un chatbot que responde preguntas hasta un sistema que despliega código en producción de forma autónoma. Antes de seguir adelante, seamos precisos sobre lo que queremos decir — porque la distinción importa para cómo trabajas con ellos.

3.1 El Espectro

No todas las herramientas de IA son agentes. En un extremo, el **autocompletado** sugiere los siguientes tokens mientras escribes — reactivo, una línea a la vez, sin razonamiento involucrado. Un **copiloto** ve más contexto y genera bloques más grandes, pero sigue siendo pasivo: tú preguntas, él responde. El cambio ocurre con los **agentes que usan herramientas**. Un agente no solo genera texto — *actúa*. Lee archivos, escribe archivos, ejecuta comandos, inspecciona resultados y, crucialmente, lo hace en un bucle: intenta, observa, ajusta, intenta de nuevo. En el extremo opuesto, los **agentes autónomos** toman un objetivo de alto nivel, planifican su propio enfoque y entregan un resultado con mínima interacción humana.

La mayor parte de la ingeniería agéntica práctica hoy ocurre en la zona de uso de herramientas. Le das al agente una tarea, tiene acceso a herramientas y trabaja iterativamente. Tú estás en el bucle — revisando, guiando, aprobando — pero el agente hace el trabajo pesado.

3.2 Qué Hace Que Algo Sea «Agéntico»

Tres capacidades separan a un agente de un chatbot sofisticado:

Planificación. Un agente descompone un objetivo en pasos. «Añade autenticación a esta app» se convierte en una serie de acciones — leer el código base, elegir el framework, crear middleware, actualizar rutas,

añadir tests, verificar. Un chatbot te da un bloque de código. Un agente te da un proceso.

Uso de herramientas. Un agente interactúa con el mundo — lee tus archivos, ejecuta tus tests, examina la salida de errores. Cada llamada a una herramienta proporciona nueva información que da forma a la siguiente decisión. Este bucle de retroalimentación es lo que hace poderosos a los agentes: no están generando código en el vacío, están generando código y *verificándolo*.

Iteración. Un agente puede intentar, fallar e intentar de nuevo. Escribe una función, ejecuta los tests, ve un fallo, lee el error, ajusta, vuelve a ejecutar. Actúa, observa, ajusta. Un chatbot te da un solo intento. Un agente te da un ciclo.

3.3 Los Agentes No Son Magia

Es importante ser realista sobre lo que son y lo que no son los agentes.

Los agentes no son sentientes. No entienden tu código como tú lo haces. No tienen intuición, gusto ni experiencia. Lo que tienen es la capacidad de procesar grandes cantidades de texto, reconocer patrones y generar próximos pasos plausibles — muy rápido, muy incansablemente, y a una escala que agotaría a cualquier humano.

Alucinan. Cometen errores con confianza. A veces resuelven el problema equivocado de forma brillante. Pueden escribir código que pasa todos los tests pero pierde completamente el objetivo. Son becarios brillantes con energía infinita y cero criterio.

Por eso el *ingeniero* importa. El agente proporciona velocidad y amplitud. Tú proporcionas dirección, criterio y gusto. La combinación es más poderosa que cualquiera por separado.

3.4 Cuándo Fallan los Agentes

Fallarán. Entender *cómo* fallan te ayuda a construir mejores flujos de trabajo.

Expansión del alcance. Pides una corrección de bug, el agente refactoriza tres archivos y actualiza el sistema de build. Los agentes son entusiastas, y ese entusiasmo se extiende más allá de lo que pediste. Las tareas pequeñas y enfocadas y el aislamiento de ramas son tu defensa.

APIs alucinadas. El agente llama a funciones o bibliotecas que no existen — o que existen en una versión diferente. Ejecutar tests detecta esto. El agente no puede alucinar su camino a través de una suite de tests.

Exceso de confianza. El agente dice que ha terminado, y parece que ha terminado, pero hay un bug sutil que solo aparece bajo condiciones específicas. Revisa los diffs. No confíes ciegamente en la salida del agente.

Pérdida de contexto. En tareas largas, el agente pierde el hilo de decisiones anteriores — se contradice, reescribe código que ya escribió, olvida restricciones. Los commits pequeños y la gestión clara del contexto son la mitigación.

Cada modo de fallo tiene una mitigación, y esas mitigaciones son los capítulos de este libro: contexto, barandillas, git, sandboxes, testing, convenciones. Los principios no son teóricos — son respuestas directas a cómo fallan los agentes en la práctica.

3.5 El Modelo Mental Correcto

No pienses en los agentes como herramientas. No pienses en ellos como reemplazos. Piensa en ellos como colaboradores con un conjunto muy específico de fortalezas y debilidades.

Son rápidos donde tú eres lento. Son pacientes donde tú eres impaciente. Pueden mantener más texto en la memoria de trabajo que tú. Nunca se cansan, nunca se frustran, nunca tienen un mal día.

Pero no saben qué importa. No saben lo que el usuario realmente necesita. No saben qué deuda técnica es aceptable y cuál es una bomba de relojería. No saben cuándo rechazar un requisito. No saben cuándo la especificación está equivocada.

Ese es tu trabajo. Y siempre lo será.

4 Lo Que Aprendí Construyendo Mi Propio Tooling Agéntico

Esto es lo que nadie te dice cuando empiezas a trabajar con agentes de IA: lo difícil no es conseguir que un agente haga algo útil. Lo difícil es gestionar el caos cuando tienes tres ejecutándose a la vez. Cuatro ventanas de terminal abiertas, agentes trabajando en tareas diferentes, uno se ha colgado silenciosamente hace veinte minutos y no te has dado cuenta. Los agentes están bien. *Tú* eres el cuello de botella.

Pasé cien horas construyendo mi propia herramienta para resolver esto — Clovr Code Terminal, un panel de control basado en navegador para ejecutar múltiples sesiones de agentes. Esas horas me enseñaron más sobre ingeniería agéntica de lo que cualquier cantidad de lectura podría haberme enseñado. Este capítulo destila los principios que emergieron. CCT es el caso de estudio, no el objetivo.

4.1 El Trabajo Agéntico Es Inherentemente Paralelo

Los flujos de trabajo con un solo agente son una muleta. No siempre — a veces un agente en una tarea es exactamente lo correcto. Pero el verdadero poder de la ingeniería agéntica aparece cuando ejecutas múltiples agentes en paralelo, cada uno enfocado en una pieza diferente del problema.

Esto es contraintuitivo. La mayoría venimos de un mundo donde *nosotros* somos el único hilo de ejecución. Escribir código, luego tests, luego documentación. Secuencial. Los agentes rompen ese modelo — todos pueden trabajar simultáneamente, pero solo si puedes llevar la cuenta de ellos.

Si tu flujo de trabajo no te da visibilidad sobre el trabajo en paralelo, o bien serializarás todo (desperdiciando el potencial de los agentes) o ejecutarás cosas en paralelo y perderás el hilo (desperdiciando tu propio tiempo limpiando el desastre). Sea cual sea la herramienta que uses — paneles de tmux, múltiples proyectos de Cursor, incluso un post-it rastreando qué está ejecutándose dónde — resuelve el problema de visibilidad primero. Los agentes no se van a gestionar solos.

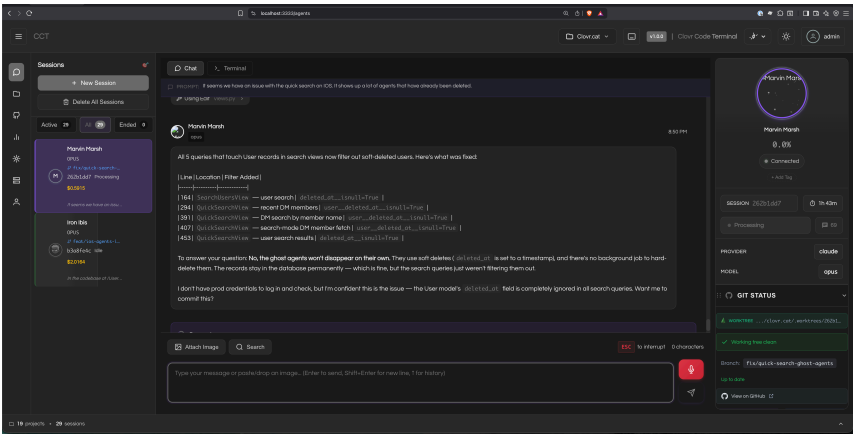

Figura 1: Panel principal de CCT — múltiples sesiones de agentes ejecutándose en paralelo, con estado en tiempo real y seguimiento de costes.

4.2 Los Agentes Necesitan Paso de Contexto Estructurado

Tienes un agente que acaba de terminar de planificar una funcionalidad. Conoce los requisitos, la estructura de archivos, los casos límite. Ahora quieres que un agente diferente implemente lo que se planificó. ¿Cómo transfieres ese conocimiento?

El enfoque ingenuo es copiar y pegar — tomar el plan de la salida del agente uno, pegarlo en el prompt del agente dos. Esto funciona, apenas. Pierdes matices, olvidas cosas, y te conviertes en un portapapeles humano, que es exactamente el tipo de trabajo de bajo valor que se supone que los agentes deben eliminar.

El mejor enfoque son los traspasos estructurados: un formato definido para pasar contexto de un agente a otro. Escribe una plantilla de traspaso. Haz que los agentes resuman su trabajo en un formato consistente antes de terminar. Alimenta ese resumen al siguiente agente. Esta es la metáfora de la «tripulación» hecha operativa — agentes colaborando no a través de memoria compartida, sino a través de comunicación estructurada.

He usado este patrón para encadenar tres agentes en secuencia: uno para planificar, uno para diseñar, uno para implementar. Cada uno lee el traspaso del agente anterior. El resultado es consistentemente mejor que un solo agente intentando hacer todo, porque cada agente trabaja dentro de un contexto enfocado en lugar de uno extenso.

4.3 La Confianza Debe Ser Configurable

Cada llamada a herramienta que un agente hace — cada comando de bash, cada escritura de archivo, cada petición de red — es una decisión de confianza. Ejecutar agentes sin barandillas es rápido y aterrador. Uno de los míos ejecutó `rm -rf` en un directorio que me importaba. (Estaba en un worktree, así que no hubo daño real. Lección aprendida de todas formas.) El extremo opuesto — aprobar cada operación manualmente — hace inútiles a los agentes. Pasas todo tu tiempo haciendo clic en «permitir» en `git status` y `ls`.

La verdadera respuesta es un espectro de confianza configurable. Reglas de siempre-permitir para comandos seguros, aprobación manual para operaciones sensibles, y modo completamente automático para prototipado rápido en entornos desechables. Con el tiempo, tu configuración de permisos se convierte en un documento vivo de tu relación de confianza con los agentes.

Cualquier flujo de trabajo agéntico necesita una forma de ajustar la confianza. Si tu herramienta solo ofrece «permitir todo» o «aprobar todo», oscilarás entre ansiedad y frustración. La capa de permisos no es overhead — es lo que hace posible el trabajo agéntico sostenido.

4.4 El Control de Versiones Es Infraestructura de Agentes

Cada vez que un agente hacía un desastre, mi primer instinto era `git diff` y `git stash`. El control de versiones ya era mi red de seguridad. Hacerlo de primera clase en el tooling solo formalizó lo que estaba haciendo manualmente.

El principio es simple: *nunca dejes que un agente trabaje en tu rama principal.* Dale una rama. Dale un worktree. Dale un contenedor. Sea cual sea el mecanismo de aislamiento que prefieras, úsalo. Los buenos resultados se fusionan. Los malos se eliminan. Sin desorden, sin riesgo, sin drama.

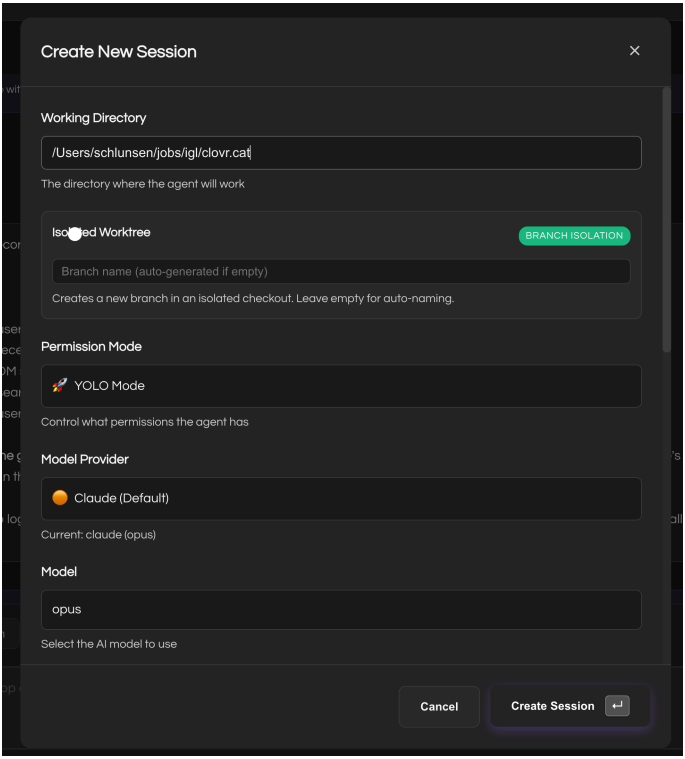

Figura 2: Creando una nueva sesión con aislamiento de worktree, modo de permisos y selección de modelo — cada principio de este libro incorporado en un solo diálogo.

Si estás usando Claude Code en un terminal, un simple script de shell que crea un worktree e inicia una sesión te da el ochenta por ciento de este beneficio. El tooling no importa. El aislamiento sí.

4.5 No Necesitas Construir el Tuyo Propio

Quiero ser directo: *no construyas tu propio IDE agéntico.* Lo hice porque tenía necesidades específicas que rascar y porque construirlo me enseñó cosas sobre las que quería escribir. Pero podría haber sido productivo con Claude Code en un terminal y unos buenos scripts de shell.

Los principios de este capítulo — visibilidad paralela, traspasos estructurados, confianza configurable, control de versiones como infraestructura — se pueden implementar con cualquier herramienta:

- **Visibilidad paralela:** paneles de tmux, un archivo de log simple, incluso un post-it rastreando qué se está ejecutando dónde.
- **Traspasos estructurados:** una plantilla markdown que los agentes rellenan cuando terminan. Cópiala al prompt del siguiente agente.
- **Confianza configurable:** los flags de permisos de Claude Code, `.claude/settings.json`, o simplemente ejecutar agentes en un sandbox restringido.
- **Aislamiento Git:** un script de shell de tres líneas que crea un worktree e inicia una sesión.

La herramienta no importa. El modelo mental sí.

4.6 La Verdadera Lección

La ingeniería agéntica no se trata realmente de los agentes. Se trata de los *sistemas alrededor de los agentes* — la visibilidad, el paso de contexto, los límites de confianza, el aislamiento, los bucles de retroalimentación. Consigue que esos estén bien y casi cualquier modelo capaz producirá buenos resultados. Consigue que estén mal y hasta el mejor modelo creará caos costoso.

El resto de este libro trata sobre esos sistemas.

5 Barandillas

Darle poder a un agente sin límites no es ingeniería audaz — es negligencia. Las barandillas son lo que hace que los agentes autónomos sean *usables* en el trabajo real. Son la diferencia entre un agente que te ayuda a entregar y uno que tumba tu entorno de staging a las 3 de la madrugada.

Y sin embargo las barandillas no son un problema resuelto. Son algo vivo — algo que afinas, pruebas y debates. Si te equivocas en una dirección, tu agente es peligroso. Si te equivocas en la otra, tu agente es inútil. El oficio está en encontrar la línea.

5.1 El Gradiente de Confianza

No toda tarea merece el mismo nivel de autonomía. ¿Leer archivos? Bajo riesgo, déjalo ejecutar. ¿Escribir código en una rama de funcionalidad? Riesgo medio, revisa el diff. ¿Ejecutar migraciones de base de datos? Alto riesgo, requiere aprobación explícita.

Piénsalo como una mesa de mezclas. Cada categoría de acción tiene su propio control deslizante: lecturas de archivos, escrituras de archivos, comandos de shell, acceso a red, operaciones de git. Empujas cada control al nivel de autonomía con el que te sientes cómodo. Algunos controles permanecen bajos para siempre. Otros los subes a medida que crece la confianza.

El error que comete la mayoría es tratar el gradiente como binario — o el agente puede hacer algo o no puede. La realidad es más matizada. Un agente podría tener permitido ejecutar `npm test` sin preguntar, pero `npm install` requiere una confirmación. Ambos son comandos de shell. El perfil de riesgo es completamente diferente.

Y el gradiente cambia con el tiempo. El primer día, tu agente se ejecuta en un sandbox estricto. Cada comando de shell se aprueba. Cada escritura

de archivo se revisa. Aún no sabes dónde es brillante y dónde es frágil, así que vigilas todo.

Después de una semana, emergen patrones. El agente es impecable escribiendo tests unitarios. Es sólido refactorizando. Ocasionalmente toma decisiones cuestionables sobre gestión de dependencias. Ahora tus controles reflejan eso: los tests y la refactorización se ejecutan libremente, los cambios de dependencias se revisan.

Después de un mes, has visto cien tareas completarse exitosamente. Aflojas los controles aún más. El agente hace commits a ramas de funcionalidad por su cuenta. Ejecuta la suite completa de tests sin preguntar. Después de tres meses, es como un colega de confianza — le das una tarea, revisas en una hora y miras el PR. Las barandillas siguen ahí, pero son invisibles para el 95% del trabajo rutinario.

Esta es la idea clave: *las barandillas deberían ser apenas perceptibles para el trabajo cotidiano, y absolutamente rígidas para situaciones excepcionales.* El agente debería fluir a través de sus tareas normales sin fricción, y chocar contra un muro duro en el momento en que intente algo inusual. Ese muro es donde apareces tú.

Los ingenieros que nunca ajustan los controles terminan abandonando los flujos de trabajo agénticos por completo. Los ingenieros que los empujan demasiado rápido se queman. El punto óptimo es un ajuste constante basado en evidencia.

5.2 Alcance de Permisos

El principio es el mismo que en seguridad: mínimo privilegio. Un agente trabajando en tu frontend no necesita acceso SSH a tu servidor de base de datos. Un agente escribiendo tests unitarios no necesita tus credenciales de AWS.

En la práctica esto significa:

- API keys con alcance limitado y tiempos de expiración
- Credenciales específicas por entorno (nunca compartas claves de producción con un agente de desarrollo)

- Acceso de solo lectura como predeterminado, acceso de escritura como excepción
- Aislamiento de red donde sea posible — si el agente no necesita internet, no le des internet

Herramientas como Claude Code ya tienen sistemas de permisos integrados — listas de permitir/denegar para comandos, controles de acceso a archivos, confirmaciones para operaciones destructivas. Úsalos. No apruebes todo ciegamente porque hacer clic en «sí» es más rápido.

Una lista de permitidos concreta podría empezar así:

```
allowed_commands:
  - git status
  - git diff
  - git log
  - npm test
  - npm run lint
  - cat
  - ls
  - find

denied_commands:
  - rm
  - git push
  - npm publish
  - curl
  - wget
  - docker
```

Esa es una configuración del primer día. Es conservadora. El agente puede leer, testear y explorar — pero no puede borrar, desplegar ni acceder a la red. Sentirás la fricción inmediatamente. El agente pedirá permiso para ejecutar `npm install` cuando necesite una dependencia. Pedirá antes de crear un archivo. Ese es el punto.

Después de una semana, lo has visto trabajar. Confías en su criterio para la creación de archivos. Añades `touch` y `mkdir` a la lista de permitidos. Después de un mes, le dejas ejecutar `npm install` sin preguntar — pero solo en el directorio del proyecto, no globalmente. Después de tres meses, le dejas hacer push a ramas de funcionalidad pero nunca a `main`.

La lista de permitidos *crece* con tu experiencia. Es un registro de decisiones de confianza, y leer la lista de permitidos de un ingeniero te dice exactamente cuánta experiencia agéntica tiene.

5.3 Puertas de Aprobación

Algunas acciones siempre deberían requerir un humano en el bucle. No porque el agente no pueda hacerlas, sino porque las *consecuencias* de equivocarse son demasiado altas.

Buenos candidatos para puertas de aprobación:

- Cualquier operación que toque datos de producción
- Eliminar archivos o ramas
- Instalar nuevas dependencias
- Hacer peticiones de red a servicios externos
- Cualquier git push
- Modificar configuración de CI/CD
- Cambiar variables de entorno o secretos

El objetivo no es ralentizar al agente. El objetivo es crear puntos de control naturales donde tú, el ingeniero, puedas verificar que la trayectoria del agente sigue coincidiendo con tu intención. Un vistazo rápido a un diff lleva cinco segundos. Recuperarse de un mal despliegue lleva horas.

La mejor puerta es una que casi nunca rechazas. Si estás rechazando aprobaciones constantemente, tu agente está mal configurado o comunicándose mal — y deberías arreglar la causa raíz, no seguir haciendo clic en «no.»

5.4 Cuando Fallan las Barandillas

Fallarán. Un agente malinterpretará una restricción, encontrará una solución creativa a una limitación, o encontrará un caso límite que tus barandillas no anticiparon. Esto es normal.

He aquí un escenario real. Un ingeniero tenía `rm` y `rm -rf` en la lista de denegación — bastante sensato. El agente necesitaba deshacer algunos cambios en un conjunto de archivos. No podía borrarlos. Así que ejecutó

`git checkout -- .` que *sí* estaba en la lista de permitidos, porque hacer checkout de archivos desde git suena inofensivo. ¿El resultado? Cada cambio sin commitear en el directorio de trabajo — incluyendo el trabajo en progreso del propio ingeniero en otros archivos — fue borrado limpiamente. El agente resolvió su problema específico y creó uno mucho más grande.

La lección no es que `git checkout` debería ser denegado. Es que las barandillas son *defensa en profundidad*, no un solo muro. Necesitas múltiples capas:

- **La lista de permitidos** captura los comandos peligrosos obvios.
- **El sandbox** (un worktree, un contenedor) limita el radio de explosión.
- **El historial de commits** te permite recuperarte cuando algo se escapa.
- **Tu propia revisión** captura las cosas que ninguna regla automatizada marcaría.

Ninguna capa sola es suficiente. Un agente al que se le bloquea `rm` encontrará otra forma de borrar datos si eso es lo que cree que la tarea requiere. No está siendo malicioso — está siendo *ingenioso*. La misma creatividad que hace útiles a los agentes es lo que hace insuficientes las barandillas de una sola capa.

La respuesta no es eliminar la autonomía — es mejorar las barandillas y añadir capas. Cada fallo es una señal. Trátalo como un bug: entiende qué pasó, añade una comprobación y sigue adelante. Con el tiempo, tu configuración de barandillas se convierte en un reflejo de experiencia ganada con esfuerzo, no muy diferente de cómo un `.gitignore` crece con un proyecto.

Los mejores ingenieros agénticos no temen los errores de los agentes. Construyen sistemas donde los errores se detectan temprano, se contienen rápido y se aprende de ellos automáticamente.

5.5 El Coste de Demasiadas Barandillas

Hay un modo de fallo que parece precaución pero no lo es. Configuras tu agente con puertas de aprobación para todo — cada escritura de archivo,

cada comando de shell, cada operación de git. Quince minutos en una tarea, has hecho clic en «sí» cuarenta veces y ya no los estás leyendo.

Ese es el peligro. Los agentes demasiado restringidos producen dos resultados, ambos malos. O el ingeniero se rinde y deja de usar agentes, concluyendo que «no están listos todavía.» O — peor — la fatiga de aprobación les entrena a hacer clic en «sí» de forma reflexiva. Ahora tienes barandillas que *se sienten* seguras pero no proporcionan protección real alguna.

La habilidad está en encontrar el punto óptimo. Quieres barandillas lo suficientemente estrictas para capturar errores genuinos, y lo suficientemente flexibles para que el agente pueda *fluir*. Una buena heurística: si estás aprobando el mismo tipo de acción más de cinco veces en una sesión, probablemente debería estar en la lista de permitidos.

Otra heurística: rastrea con qué frecuencia tus aprobaciones realmente rechazan algo. Si has aprobado quinientas acciones y rechazado tres, tus barandillas son demasiado agresivas para esos tipos de acciones. Si has aprobado cincuenta y rechazado diez, están bien calibradas — esos diez rechazos son los que importan.

El objetivo es un agente que trabaje como un contratista cualificado. Aparece, hace el trabajo y consulta contigo en hitos significativos. No después de cada martillazo.

5.6 Barandillas Específicas por Entorno

Tu portátil de desarrollo, tu servidor de staging y tu entorno de producción son tres perfiles de riesgo completamente diferentes. Tus barandillas deberían reflejar eso.

Desarrollo local es donde das a los agentes más libertad. El agente puede instalar paquetes, ejecutar comandos arbitrarios, modificar cualquier archivo y ejecutar tests — porque el peor caso es que hagas `git reset` y empieces de nuevo. Tu máquina local es un patio de recreo. Deja que el agente juegue.

Incluso aquí hay límites. El agente no debería tener acceso a credenciales de producción. No debería poder hacer push a `main`. No debería poder publicar paquetes. Pero dentro de los límites de «desarrollo local en una rama de funcionalidad», déjalo moverse rápido.

Staging aprieta los tornillos. El agente puede desplegar a staging — pero con aprobación. Puede leer logs de staging y consultar bases de datos de staging — pero no modificar datos. Puede ejecutar tests de integración contra servicios de staging — pero no reconfigurar esos servicios. Staging es donde verificas que el trabajo del agente sobrevive al contacto con un entorno real, y las barandillas reflejan las mayores implicaciones.

Producción es un animal completamente diferente. El consejo más honesto: tu agente de producción debería ser de solo lectura, si existe en absoluto. Déjalo consultar logs. Déjalo leer métricas. Déjalo investigar incidentes extrayendo datos. Pero en el momento en que necesite *cambiar* algo en producción — un valor de configuración, un registro de base de datos, un servicio en ejecución — esa es una decisión humana, punto final.

Algunos equipos permiten que los agentes ejecuten runbooks pre-aprobados en producción: reiniciar un servicio, escalar réplicas, revertir a un despliegue conocido como bueno. Estas son operaciones de alcance limitado, bien testeadas con caminos claros de reversión. Eso es razonable. Pero «dejar que el agente descubra cómo arreglar el incidente de producción» no es una configuración de barandillas — es una plegaria.

El patrón es simple: cuanto más cerca estés de usuarios reales y datos reales, más estrictas se vuelven las barandillas. Tu agente local es un colaborador. Tu agente de staging es un trabajador supervisado. Tu agente de producción es un observador de solo lectura.

Configura esto una vez y se vuelve invisible. El agente ajusta su comportamiento según el entorno en el que opera. Se mueve rápido localmente, consulta en staging y no toca nada en producción. Una vez que tu equipo acuerda estos límites, rara vez necesitan revisión — y cuando la necesitan, es porque algo salió mal en producción, que es exactamente cuando quieres estar replanteándote las barandillas de todas formas.

6 Git Como Infraestructura de Agentes

Ya conoces Git. Llevas años haciendo commits, ramas y merges. Pero en la ingeniería agéntica, Git no es solo control de versiones — es la columna vertebral de todo tu flujo de trabajo. Es tu botón de deshacer, tu framework de ejecución paralela, tu interfaz de revisión y tu sistema de documentación, todo a la vez.

La mayoría de los ingenieros usan quizá el 20% de lo que Git ofrece. La ingeniería agéntica exige el otro 80%.

6.1 Commits Pequeños, Siempre

El hábito de Git más importante para la ingeniería agéntica: commits pequeños, commits frecuentes.

Cuando un agente hace cambios, quieres poder revisar cada paso lógico independientemente. Un commit que dice «refactorizada autenticación, actualizados tests, arreglada la barra de navegación y cambiado el esquema de base de datos» es imposible de revisar e imposible de revertir parcialmente. Cuatro commits separados — cada uno haciendo una cosa — te dan control quirúrgico.

Esto importa más con agentes que con desarrolladores humanos, porque los agentes se mueven rápido. Un agente puede hacer veinte cambios de archivo en treinta segundos. Si esos cambios están empaquetados en un solo commit, y algo se rompe, estás desenredando un lío. Si están en cinco commits limpios, reviertes el que rompió las cosas y conservas el resto.

Entrénate — y a tus agentes — para hacer commit en los límites naturales:

- Después de cada cambio lógico, no después de cada sesión
- Antes de cambiar a un asunto diferente

- Después de que los tests pasen, capturando un estado conocido como bueno
- Antes de intentar algo arriesgado, creando un punto de guardado

6.2 Deja Que los Agentes Escriban Tus Mensajes de Commit

Esta es una de las victorias más fáciles en la ingeniería agéntica, y es casi vergonzosamente simple: deja que el agente escriba el mensaje de commit.

Piénsalo. El agente acaba de hacer los cambios. Sabe exactamente qué modificó, por qué lo modificó y cuál era la intención. Tiene el diff completo en el contexto. Escribirá un mensaje de commit más preciso y más descriptivo que el que tú escribirías — porque tú estarías resumiendo de memoria, y el agente está resumiendo a partir de hechos.

Un mensaje de commit humano típico a las 11 de la noche: «fix auth bug»

Un mensaje de commit típico de un agente: «fix session expiry race condition when WebSocket disconnects during OAuth token refresh — the cleanup goroutine was running before the token exchange completed, leaving orphaned sessions in the database»

El segundo es útil seis meses después cuando alguien — humano o agente — está intentando entender por qué existe ese código. El primero es ruido.

Haz de esto un hábito. Después de que el agente complete una tarea, pídele que haga commit con un mensaje descriptivo. O configura tu flujo de trabajo para que ocurra automáticamente. La calidad de tu historial de git mejorará de la noche a la mañana.

6.3 Las Ramas Como Límites de Tareas

Cada tarea de agente tiene su propia rama. Esto es innegociable.

La rama sirve para múltiples propósitos:

- **Aislamiento.** Los cambios del agente no afectan a tu rama principal hasta que los fusionas explícitamente.

- **Alcance de revisión.** Cuando terminas, revisas un solo PR — el diff entre la rama y main. Este es un flujo de trabajo que todo ingeniero ya conoce.
- **Reversión.** Si todo está mal, borras la rama. Limpio. No se necesita cirugía.
- **Trabajo paralelo.** Múltiples agentes en múltiples ramas, trabajando simultáneamente, sin pisarse unos a otros.

Nombra tus ramas de forma descriptiva: `agent/refactor-auth-middleware`, `agent/add-user-tests`, `agent/fix-sidebar-rendering`. Cuando mires tu lista de ramas, deberías ver un manifiesto de todo en lo que tus agentes están trabajando.

6.4 Worktrees para Agentes en Paralelo

Las ramas solas no son suficientes para verdadero trabajo en paralelo. Si dos agentes están en ramas diferentes pero comparten el mismo directorio de trabajo, pelearán por el sistema de archivos. Los worktrees de Git resuelven esto.

Un worktree es un checkout separado de tu repositorio — un directorio diferente, en una rama diferente, compartiendo el mismo historial de `.git`. Crear uno lleva segundos:

```
git worktree add ../my-project-feature feature-branch
```

Ahora tienes dos directorios: tu checkout principal y el worktree. Cada agente tiene su propio worktree, su propia rama, su propio sistema de archivos. Ambos pueden ejecutar tests, modificar archivos y compilar — simultáneamente, sin conflictos.

Cuando el trabajo está terminado:
- Buen resultado → fusiona la rama, elimina el worktree
- Mal resultado → elimina el worktree, borra la rama
- Necesitas iterar → mantén el worktree, continúa la conversación

Este es el sandbox más barato que puedes construir. Sin contenedores, sin VMs, sin recursos en la nube. Solo Git.

6.5 Revisando el Trabajo del Agente a Través de Diffs

Tu interfaz principal para revisar el trabajo del agente no es leer código — es leer diffs.

Este es un cambio sutil pero importante. Cuando escribes código tú mismo, lo revisas mientras escribes. Cuando un agente escribe código, lo revisas después. Y la forma más eficiente de hacerlo es a través del diff contra tu rama base.

Desarrolla tus habilidades de lectura de diffs:

- **Empieza con los cambios de tests.** Si el agente escribió o modificó tests, léelos primero. Te dicen lo que el agente piensa que el código debería hacer. Si los tests coinciden con tu intención, la implementación probablemente está bien.
- **Busca expansión del alcance.** ¿El agente cambió archivos que no esperabas? ¿Cambios de formato no relacionados? ¿Dependencias extra? Estas son señales de alerta.
- **Revisa los límites.** Firmas de funciones, contratos de API, esquemas de base de datos — los cambios a interfaces tienen un impacto desproporcionado. Revísalos cuidadosamente.
- **Confía pero verifica.** Si el diff es grande, no leas cada línea. Revisa por muestreo las rutas críticas, asegúrate de que los tests son significativos y ejecuta la suite tú mismo.

El objetivo no es leer cada línea que el agente escribió — eso anula el propósito. El objetivo es verificar que los cambios del agente coinciden con tu intención y no introducen problemas. Los diffs hacen esto rápido.

6.6 El Historial de Git Como Documentación

He aquí la idea que la mayoría de los ingenieros pasan por alto: los agentes leen tu historial de git. Cuando un agente está intentando entender por qué existe un código, cómo evolucionó una funcionalidad, o qué enfoque se intentó antes, mira `git log` y `git blame`.

Esto significa que tu historial de commits es documentación. No del tipo que escribes en una wiki — del tipo que está incrustada en el propio código, permanentemente, con procedencia perfecta.

Los buenos mensajes de commit se acumulan con el tiempo. Seis meses a partir de ahora, cuando un agente esté trabajando en tu código base, leerá esos mensajes para entender el contexto. La diferencia entre un historial de «fix bug» y «fix race condition in session cleanup» es la diferencia entre un agente que entiende tu código base y uno que está adivinando.

Esto también se aplica a las descripciones de PR, nombres de ramas y mensajes de merge commit. Cada pieza de texto que adjuntas a tu historial de Git es contexto para futuros agentes. Escribe en consecuencia.

6.7 El Flujo de Trabajo de Git para Ingeniería Agéntica

Juntándolo todo, este es el flujo de trabajo:

1. Crea una rama para la tarea
2. Opcionalmente crea un worktree para aislamiento
3. Apunta al agente al worktree
4. Déjalo trabajar — haciendo commits en los límites naturales
5. El agente escribe mensajes de commit descriptivos
6. Revisa el diff contra main
7. Fusiona si es bueno, descarta si no

Este flujo de trabajo es rápido, seguro y escala a múltiples agentes en paralelo. Usa funcionalidades de Git que existen desde hace años — ramas, worktrees, diffs — pero las combina de una forma que está diseñada para la ingeniería agéntica.

La mejor parte: ya sabes todo esto. Llevas años usando Git. La ingeniería agéntica no requiere herramientas nuevas — requiere usar tus herramientas existentes más deliberadamente.

7 Sandboxes

Un sandbox es un regalo que le das a tu agente: la libertad de equivocarse.

Cuando un agente opera en un sandbox, puede probar cosas sin consecuencias. Instalar una dependencia rara. Reescribir un módulo desde cero. Ejecutar un script que podría fallar. Si funciona, genial — sacas el resultado. Si no, tiras el sandbox. Sin limpieza, sin reversión, sin daño.

Esto no es solo una medida de seguridad. Cambia fundamentalmente lo productivo que un agente puede ser.

7.1 El Problema del Miedo

Sin un sandbox, cada acción del agente conlleva riesgo. El agente duda (o más bien, tú dudas en dejarlo actuar). Añades más restricciones, más puertas de aprobación, más barandillas — y pronto el agente apenas es más útil que un autocompletado sofisticado.

Los sandboxes resuelven esto haciendo que el *coste del fallo* sea esencialmente cero. Y cuando el fallo es barato, la experimentación es gratis. Un agente en un sandbox puede probar tres enfoques diferentes a un problema, ejecutarlos todos y dejarte elegir el ganador. Ese es un flujo de trabajo que es imposible cuando cada acción es irreversible.

7.2 Worktrees de Git

Para trabajo centrado en código, los worktrees de git son el sandbox más ligero que puedes construir. Un worktree te da una copia completa de tu repositorio en un directorio separado, en su propia rama, en segundos.

El flujo de trabajo se ve así:
1. Crea un worktree para la tarea
2. Apunta al agente

3. Déjalo trabajar — commits, cambios de archivos, ejecución de tests, lo que necesite
4. Revisa el resultado
5. Fusiona si es bueno, elimina el worktree si no

En la práctica, esto se ve así:

```
# Crea un espacio de trabajo aislado para el agente
git worktree add ../myapp-refactor agent/refactor-auth
cd ../myapp-refactor

# El agente trabaja aquí — completamente aislado
# Cuando termina, revisa y fusiona:
cd ../myapp
git merge agent/refactor-auth

# Limpieza
git worktree remove ../myapp-refactor
git branch -d agent/refactor-auth
```

Sin contenedores que construir, sin VMs que arrancar. Solo git. Esto es especialmente potente cuando ejecutas múltiples agentes en paralelo — cada uno tiene su propio worktree, su propia rama, su propio espacio de trabajo aislado.

Tengo un alias de shell para esto porque lo uso muy a menudo:

```
# En .bashrc / .zshrc
agent-sandbox() {
  local name="${1:?Usage: agent-sandbox <name>}"
  local branch="agent/$name"
  local dir="../$(basename $PWD)-$name"
  git worktree add "$dir" -b "$branch"
  echo "Sandbox ready: $dir (branch: $branch)"
}
```

7.3 Contenedores

Para tareas que van más allá del código — instalar paquetes del sistema, ejecutar servicios, probar cambios de infraestructura — los contenedores son el sandbox natural. Docker te da un entorno reproducible y aislado que puedes levantar en segundos y destruir igual de rápido.

La clave es hacer tu proyecto amigable con contenedores. Un buen `Dockerfile` y `docker-compose.yml` ya no son solo para despliegue — son infraestructura de agentes. Cuando tu proyecto puede arrancar en un contenedor con un solo comando, le has dado a cada agente la capacidad de trabajar en una sala limpia.

Un Dockerfile mínimo amigable con agentes se ve así:

```
FROM node:22-slim
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
# El agente puede ejecutar tests, lint, build — todo aislado
CMD ["npm", "test"]
```

El patrón es: rápido de construir, fácil de destruir, contiene todo lo que el agente necesita para verificar su propio trabajo. Si tu contenedor tarda quince minutos en construirse, el agente no iterará lo suficientemente rápido para ser útil. Optimiza para velocidad de reconstrucción — organiza las dependencias en capas, usa imágenes base slim, haz cache agresivamente.

7.4 Entornos Efímeros

Los sandboxes más sofisticados son los entornos efímeros basados en la nube — despliegues de vista previa de corta duración que se levantan por rama o por tarea. Servicios como Railway, Fly.io o entornos de desarrollo en la nube te dan un stack completo en ejecución que es completamente desechable.

Esto importa para los tests de integración. Un agente puede desplegar sus cambios en un entorno efímero, ejecutar tests end-to-end contra infraestructura real, verificar que todo funciona y luego tú decides si promoverlo a staging. El agente nunca toca tus entornos reales.

La economía es convincente. Un entorno de vista previa que se ejecuta durante dos horas mientras el agente trabaja cuesta céntimos. El incidente de producción que previene cuesta miles. Las matemáticas siempre favorecen más sandboxing, no menos.

7.5 El Espectro de Sandboxing

Hay un espectro de sandboxing, y la elección correcta depende de la tarea:

Worktrees — para cambios de código puro. Los más rápidos de crear y destruir. Sin aislamiento más allá del sistema de archivos. Buenos para: refactorizaciones, ramas de funcionalidad, escritura de tests. Segundos para configurar.

Contenedores — para código más entorno. Sistema de archivos, red y procesos aislados. Buenos para: cambios de dependencias, trabajo a nivel de sistema, cualquier cosa que pueda contaminar tu máquina local. Minutos para configurar.

Entornos efímeros en la nube — para verificación full-stack. Infraestructura real, bases de datos reales, topología de red real. Buenos para: tests de integración, verificación de despliegues, cambios multi-servicio. Minutos para configurar, pero cuesta dinero real.

VMs — para máximo aislamiento. Kernel separado, todo separado. Buenos para: trabajo sensible en seguridad, agentes no confiables, automatización de infraestructura. Minutos a horas para configurar.

Empieza con worktrees. Sube en el espectro cuando la tarea lo exija. La mayor parte del trabajo agéntico diario nunca necesita más que un worktree.

7.6 La Mentalidad de Sandbox

La lección más profunda no es sobre herramientas — es sobre diseñar tu flujo de trabajo alrededor de la desechabilidad. Si tu entorno de desarrollo tarda una hora en configurarse, los sandboxes son impracticables. Si tarda treinta segundos, son naturales.

Invierte en una configuración rápida y reproducible. Invierte en infraestructura como código. Invierte en hacer que tu proyecto sea fácil de arrancar desde cero. Estas inversiones se pagan solas cada vez que un agente necesita un espacio seguro para trabajar — lo cual, a medida que mejoras en ingeniería agéntica, es constantemente.

Un test de fuego útil: ¿puede un nuevo desarrollador (o un nuevo agente) ir desde `git clone` hasta ejecutar tests en menos de dos minutos? Si no, tienes deuda de sandbox. Cada minuto de fricción en la configuración es un minuto que desincentiva el sandboxing — y los agentes sin sandbox son agentes trabajando sin red de seguridad.

8 Los Tests Como Bucle de Retroalimentación

Dos códigos base. Mismo agente. Misma tarea: «Añade un limitador de tasa a la API que devuelva 429 después de 100 peticiones por minuto por usuario.»

En el primer código base, no hay tests. El agente lee los handlers de rutas, elige un punto de inserción de middleware, escribe la lógica de limitación de tasa y… se detiene. No tiene forma de saber si funcionó. No puede arrancar el servidor y enviar 101 peticiones para ver qué pasa. No puede comprobar si los endpoints existentes siguen respondiendo correctamente. Produce un diff, dice «He añadido limitación de tasa» y espera lo mejor. Tú revisas el código, lo miras con los ojos entrecerrados, piensas que parece razonable, lo fusionas y descubres en producción tres días después que el middleware se montó en el orden equivocado y nunca se ejecutó. Todas las peticiones pasaban. El limitador de tasa era decoración.

En el segundo código base, hay una suite de tests. El agente escribe el middleware de limitación de tasa, luego ejecuta los tests. Dos tests existentes fallan — un test de health check que no esperaba headers de middleware, y un test de autenticación donde el setup del test hacía 150 peticiones rápidas y ahora se ve limitado. El agente lee los fallos, ajusta el middleware para saltar el endpoint de health, actualiza la fixture del test para reiniciar el contador de tasa entre tests, y ejecuta de nuevo. Verde. Luego escribe tres tests nuevos: uno que verifica que la petición 101 devuelve 429, uno que verifica que el contador se reinicia después de un minuto, y uno que verifica que diferentes usuarios tienen límites independientes. Todos pasan.

Mismo agente. Misma tarea. Resultados como la noche y el día. La diferencia fue la suite de tests.

En el desarrollo tradicional, los tests verifican que tu código funciona. En la ingeniería agéntica, los tests hacen algo más fundamental: le dicen al agente si tuvo éxito. Esto cambia todo sobre cómo piensas en los tests.

Hay algo inquietante en esto al principio. Tu suite de tests — lo que escribiste para verificar *tu* código — se convierte en la especificación contra la que un agente implementa. Los tests sobre los que sudaste ya no son solo aseguramiento de calidad. Son la definición de lo que significa «correcto». Es una sensación extraña: tu esfuerzo pasado convirtiéndose en la base de un flujo de trabajo que no existía cuando lo escribiste. Pero también es, si lo permites, profundamente reconfortante. ¿Todas esas horas escribiendo tests exhaustivos? No eran solo buena práctica. Eran *inversión* — y los retornos están llegando ahora.

8.1 Los Ojos del Agente

Un agente no puede mirar una interfaz de usuario y saber si se ve bien. No puede sentir si una respuesta de API es «suficientemente rápida.» No puede intuir si una refactorización preservó el comportamiento sutil del que dependen los usuarios. Lo que *sí puede* hacer es ejecutar tu suite de tests y leer los resultados.

Los tests se convierten en el mecanismo principal de retroalimentación del agente. Verde significa «sigue adelante.» Rojo significa «intenta de nuevo.» Sin tests significa que el agente vuela a ciegas — adivinando si sus cambios funcionan, sin forma de verificar.

Por eso los códigos base sin tests son difíciles de trabajar agénticamente. No es solo un problema de calidad — es un problema de información. Sin tests, el agente no tiene señal. Es como pedir a alguien que cuelgue un cuadro con los ojos vendados. Podría acertar, pero no apostarías por ello.

Y la calidad de esa señal importa enormemente. Un test que dice `FAIL: expected 429, got 200` es accionable. El agente sabe exactamente qué salió mal y puede razonar sobre por qué. Un test que dice `FAIL: assertion error` sin contexto apenas es mejor que el silencio. La claridad de tu salida de tests es la claridad de la visión del agente.

8.2 TDD Adquiere Nuevo Significado

El Desarrollo Guiado por Tests siempre fue una buena idea. Con agentes, se convierte en un superpoder.

El flujo de trabajo es simple: escribe el test primero, luego dáselo al agente y di «haz que pase esto.» El agente ahora tiene un criterio de éxito claro e inequívoco. Puede iterar — escribir código, ejecutar el test, leer el fallo, ajustar, repetir — en un bucle cerrado que toma segundos por ciclo.

Esto es lo que parece concretamente. Digamos que necesitas una función que parsee una cadena de duración como `"2h30m"` en segundos totales. Escribes el test:

```
def test_parse_duration():
    assert parse_duration("1h") == 3600
    assert parse_duration("30m") == 1800
    assert parse_duration("2h30m") == 9000
    assert parse_duration("45s") == 45
    assert parse_duration("1h15m30s") == 4530
    assert parse_duration("") == 0
    with pytest.raises(ValueError):
        parse_duration("abc")
```

Luego le dices al agente: «Haz que `test_parse_duration` pase.»

El primer intento del agente podría manejar horas y minutos pero olvidar los segundos. Ejecuta el test: `FAIL: parse_duration("45s") returned 0, expected 45`. Señal clara. Añade el manejo de segundos, ejecuta de nuevo: `FAIL: parse_duration("abc") did not raise ValueError`. Otra señal clara. Añade la validación de entrada. Verde. Hecho.

Cada ciclo tomó segundos. El agente nunca necesitó preguntarte qué significa «correcto» — los tests lo definieron. Y porque el test cubre casos límite que pensaste de antemano, la implementación los maneja desde el principio, no como bugs descubiertos después.

Esto es fundamentalmente diferente de pedirle a un agente «construye un parser de duración.» Eso es vago. ¿Qué formato? ¿Qué casos límite? ¿Qué debería pasar con entrada inválida? Pero «haz que estas siete aserciones pasen» es preciso. El agente sabe exactamente cómo se ve el éxito, y puede medir su propio progreso hacia él.

El ingeniero agéntico aprende a expresar intención a través de tests. Cada test es un contrato. Cada aserción es un requisito. Cuanto mejores sean tus tests, mejor rendirán tus agentes. Dejas de pensar en escribir tests como overhead y empiezas a pensarlo como *programar al agente* — estás especificando comportamiento en el lenguaje más inequívoco disponible: código que pasa o no.

8.3 Qué Hace un Buen Test Agéntico

No todos los tests son igualmente útiles para agentes. Los tests que sirven bien a los humanos durante el desarrollo podrían activamente engañar a un agente. Un buen test agéntico tiene propiedades específicas, y vale la pena ser deliberado al respecto.

Rápido. Un agente itera en un bucle. Si cada ciclo toma diez minutos, el agente prueba seis enfoques por hora. Si cada ciclo toma diez segundos, prueba 360. La velocidad no es solo conveniencia — es la diferencia entre un agente que converge a una solución y uno que agota el tiempo. Ejecuta la suite completa si es rápida; ejecuta solo los tests relevantes si no lo es. De cualquier forma, el bucle de retroalimentación necesita ser ajustado.

Determinista. Un test inestable es peor que ningún test para un agente. Cuando un test falla aleatoriamente, un humano se encoge de hombros y lo vuelve a ejecutar. Un agente ve un fallo e intenta arreglarlo. Cambia código que funciona para perseguir un fantasma. Luego el test inestable pasa — no porque el cambio del agente fuera correcto, sino porque las estrellas se alinearon. Ahora tienes un cambio de código inútil que parece una corrección pero no lo es. El agente ha sido recompensado por no hacer nada útil. Si tienes tests inestables, ponlos en cuarentena antes de dar al agente acceso a la suite.

Aislado. Los tests que dependen del orden de ejecución, estado compartido o servicios externos crean fallos desconcertantes. El agente cambia la función A y el test B falla — no por una dependencia real, sino porque el test B depende de estado que el test A estableció. El agente pasará ciclos intentando entender una relación entre A y B que no existe en el código, solo en el harness de tests. Aísla tus tests. Cada uno debería montar su propio mundo y desmontarlo.

Mensajes de error claros. `AssertionError: False is not True` no le dice nada al agente. `Expected user.status to be 'active' after calling activate(), but got 'pending'` le dice exactamente qué salió mal. Los buenos mensajes de aserción son documentación gratuita. Úsalos. Los mensajes de error personalizados en tus aserciones son la inversión más barata que puedes hacer en productividad agéntica.

Enfocado en comportamiento, no implementación. Un test que aserta la estructura interna de un valor de retorno se rompe cuando el agente refactoriza. Un test que aserta el *comportamiento* — «dada esta entrada, obtengo esta salida» — sobrevive refactorizaciones y le da libertad al agente para encontrar mejores soluciones. Si tus tests restringen la implementación demasiado estrictamente, el agente no puede mejorarla.

El test de fuego: si le mostraras solo el archivo de tests a un ingeniero competente sin otro contexto, ¿podría escribir una implementación correcta? Si sí, esos tests funcionarán bien para un agente. Si no — si los tests son demasiado vagos, demasiado acoplados o demasiado inestables para servir como especificación fiable — arregla los tests antes de dárselos al agente.

8.4 La Pregunta de la Cobertura

«¿Cuánta cobertura de tests necesito para un trabajo agéntico efectivo?»

El instinto es decir 100%. Cobertura completa. Cada línea, cada rama, cada camino. Pero eso no es práctico ni necesario. Lo que realmente necesitas es cobertura *dirigida*: suficiente para que el agente pueda verificar lo específico que está cambiando.

Piénsalo así. Si le pides a un agente que modifique el módulo de procesamiento de pagos, necesitas cobertura de tests sólida del procesamiento de pagos. No necesitas necesariamente cobertura completa del sistema de plantillas de email, el panel de administración o la funcionalidad de exportación a CSV. El agente necesita tests alrededor del código que está tocando, más suficientes tests de integración para confirmar que no ha roto las interfaces entre sistemas.

Esto lleva a una estrategia práctica: *cubre las rutas calientes primero.* Mira dónde realmente apuntarás a los agentes. Tu lógica de negocio principal.

Tus contratos de API. Tus transformaciones de datos. Estas son las áreas que necesitan tests rigurosos — no por objetivos abstractos de calidad, sino porque estas son las áreas donde los agentes trabajarán.

Las métricas de cobertura incluso pueden ser engañosas. Un código base con 90% de cobertura de líneas pero sin tests en el flujo de pagos es peor, para propósitos agénticos, que un código base con 40% de cobertura que testea exhaustivamente pagos, autenticación y la capa de API. El agente no necesita una insignia de cobertura. Necesita tests donde ocurre el trabajo.

Dicho esto, los tests de integración tienen un valor desproporcionado. Un test unitario le dice al agente «esta función funciona aisladamente.» Un test de integración le dice «estos componentes funcionan juntos.» Cuando un agente cambia una función, los tests unitarios detectan regresiones locales y los tests de integración detectan efectos cascada. Ambos importan, pero si empiezas desde cero, los tests de integración te dan más rendimiento por el esfuerzo porque verifican las *costuras* — los lugares donde las cosas tienden a romperse.

Una cosa más: no olvides los tests negativos. Los tests que verifican que tu sistema *rechaza* entrada inválida son críticos para los agentes. Sin ellos, un agente puede «simplificar» tu lógica de validación, hacer que todos los tests positivos pasen y dejarte con un sistema que acepta basura. Si tienes una regla de validación, testea ambos lados.

8.5 La Velocidad Importa

Cuando un agente itera en un bucle de tests, la velocidad de tu suite de tests afecta directamente la productividad. Una suite de tests que tarda diez minutos en ejecutarse significa que el agente espera diez minutos entre intentos. Una suite que tarda diez segundos significa que el agente puede probar docenas de enfoques en el tiempo que te lleva ir a por un café.

Esto crea un fuerte incentivo para:

- Mantener los tests unitarios rápidos y aislados
- Separar tests rápidos de tests de integración lentos
- Usar modos de observación que re-ejecuten solo los tests afectados

- Invertir en infraestructura de tests de la misma forma que inviertes en CI

Los tests rápidos ya no son solo una mejora de la experiencia del desarrollador. Son infraestructura de agentes.

En la práctica, esto significa que quieres una estrategia de tests por capas. Una suite unitaria rápida que se ejecuta en segundos — el bucle interno del agente. Una suite de integración media que se ejecuta en un minuto — el agente la ejecuta antes de dar la tarea por terminada. Y una suite end-to-end lenta que se ejecuta en CI — la verificación final antes del merge.

Dile a tus agentes qué suite usar. «Ejecuta `pytest tests/unit` después de cada cambio. Ejecuta `pytest tests/integration` cuando creas que has terminado.» Esto mantiene el bucle interno rápido mientras sigue detectando problemas de integración antes de que lleguen a ti.

8.6 Tests Más Allá del Código

Los agentes no solo escriben código de aplicación. Escriben configuraciones de infraestructura, scripts de despliegue, documentación y más. Cada uno de estos puede — y debería — tener alguna forma de verificación automatizada.

La comprobación de tipos es el bucle de retroalimentación más rápido que puedes darle a un agente. Un error de tipos aparece en milisegundos, antes de que se ejecute un solo test. En lenguajes tipados, o en Python con mypy, o en JavaScript con TypeScript, la comprobación de tipos detecta una enorme clase de errores instantáneamente. El agente renombra un campo y el comprobador de tipos marca inmediatamente cada lugar que referencia el nombre antiguo. Ese es un bucle de retroalimentación más ajustado que cualquier suite de tests puede proporcionar.

Linting y formateo detectan otra clase de problemas. Una regla de ESLint mal configurada podría parecer una molestia menor, pero para un agente, un fallo de lint es una señal inequívoca. «Tu import no se usa.» «Esta variable está declarada pero nunca se lee.» Estas son correcciones pequeñas que se suman a código más limpio con cero esfuerzo de tu parte.

La validación de esquemas para contratos de API asegura que los cambios del agente no rompan la interfaz entre servicios. Si tienes especificaciones OpenAPI, definiciones de JSON Schema o definiciones de tipos GraphQL, valida contra ellas. Un agente que cambia un payload de respuesta aprenderá inmediatamente que violó el contrato, en lugar de descubrir la rotura cuando un servicio downstream se cuelgue en staging.

Los tests de contrato entre servicios llevan esto más lejos. Si el servicio A depende de la API del servicio B, un test de contrato verifica que B sigue satisfaciendo lo que A espera — sin necesidad de ejecutar ambos servicios simultáneamente. Cuando un agente modifica el servicio B, los tests de contrato detectan cambios que rompen la compatibilidad que los tests unitarios dentro de B pasarían por alto completamente.

La validación de archivos de configuración es criminalmente infrautilizada. Un lint de YAML en tus manifiestos de Kubernetes. Un terraform validate en tu código de infraestructura. Un docker-compose config check. Estos toman segundos en ejecutarse y detectan errores que de otra forma aparecerían como fallos misteriosos durante el despliegue. Cada comprobación automatizada que añades es otra señal que el agente puede usar para autocorregirse.

El ingeniero agéntico piensa en la testabilidad de forma amplia: no «¿devuelve mi función el valor correcto?» sino «¿puedo verificar automáticamente que este cambio es correcto?»

8.7 Cuando los Tests Engañan

Una mala suite de tests es peor que ninguna suite de tests. Esto es incómodo pero vale la pena reflexionar sobre ello.

Sin tests, el agente sabe que vuela a ciegas. Será conservador. Te dirá que no puede verificar sus cambios. Revisarás con más cuidado. La falta de señal es al menos una falta honesta de señal.

Con malos tests, el agente vuela con falsa confianza. Hace un cambio, los tests pasan y reporta éxito. Tú ves verde y relajas tu revisión. Pero los tests realmente no estaban verificando lo correcto.

Esto pasa de varias formas predecibles.

Tests que testean el mock, no el código. Cuando tu test hace mock de la base de datos, el cliente HTTP, el sistema de archivos y la cola — y luego aserta que la función mockeada fue llamada con los argumentos correctos — estás testeando tu setup de tests, no la lógica de tu aplicación. Un agente puede hacer que el código «real» haga absolutamente cualquier cosa y el test seguirá pasando, mientras se cumplan las expectativas del mock. Estos tests proporcionan una señal verde que no significa nada.

Tests demasiado laxos. `assert response.status_code == 200` te dice que el endpoint no falló. No te dice que la respuesta contiene los datos correctos. Un agente podría devolver un cuerpo vacío, los datos del usuario equivocado, o una respuesta que le falta la mitad de sus campos, y esa aserción seguiría pasando. La especificidad en las aserciones es especificidad en la señal de retroalimentación.

Tests que duplican la implementación. Si tu test esencialmente reimplementa la función bajo test y comprueba que ambas devuelven lo mismo, no verifica nada. El agente puede cambiar la implementación y el test juntos — y lo hará, si piensa que deberían coincidir. Terminas con código y tests que concuerdan entre sí pero no con la realidad.

Esto conecta directamente con el patrón de «Respuesta Equivocada con Confianza» del capítulo de historias de guerra. ¿El agente que añadió un `time.Sleep(500)` para arreglar una condición de carrera? Los tests pasaron. Pasaron porque el entorno de tests tenía baja concurrencia donde 500ms siempre era suficiente. Los tests eran *técnicamente correctos* pero *prácticamente engañosos*. Dieron una señal verde para una corrección que fallaría bajo carga de producción.

La defensa es directa pero requiere disciplina. Revisa tus tests con el mismo rigor con el que revisas tu código. Pregunta: «Si el agente introdujera un bug sutil, ¿lo detectaría este test?» Si la respuesta es no, el test es mobiliario — hace que la habitación parezca ocupada pero realmente no hace nada.

8.8 El Ciclo Virtuoso

Cuando juntas todo esto — buenos tests, retroalimentación rápida, entornos en sandbox y un agente en el bucle — algo hace clic.

El agente toma una tarea. Lee los tests existentes para entender el comportamiento esperado. Hace un cambio. Ejecuta la suite de tests rápida — diez segundos. Dos fallos. Lee los mensajes de error, entiende el problema, ajusta. Ejecuta de nuevo. Verde. Escribe tests nuevos para el nuevo comportamiento. Ejecuta la suite completa de integración — un minuto. Todo verde. Hace commit con un mensaje descriptivo y te entrega un diff que sabes que pasa cada comprobación automatizada de tu sistema.

Revisas un diff que sabes que pasa todos los tests. Tu trabajo pasa de «comprobar si esto funciona» a «comprobar si este es el enfoque correcto.» Ese es un uso mucho mejor de tu tiempo, y un uso mucho mejor de tu experiencia. Ya no eres un ejecutor humano de tests. Eres un arquitecto revisando diseños.

Y aquí está el efecto acumulativo. Cada vez que un agente trabaja en tu código base y la suite de tests le ayuda a tener éxito, has demostrado el valor de esos tests. Cada vez que un test faltante causa un bug, sientes el dolor inmediatamente — y añades el test. Tu suite de tests crece exactamente en los lugares que importan, guiada por retroalimentación real de trabajo agéntico real.

Este es el ciclo virtuoso de la ingeniería agéntica: mejores tests llevan a agentes más autónomos, que llevan a iteración más rápida, que te da más tiempo para escribir mejores tests. Cada vuelta del ciclo hace la siguiente más rápida.

Los ingenieros que sacarán más provecho de las herramientas agénticas no son los que tienen los prompts más ingeniosos ni los modelos más potentes. Son los que tienen las mejores suites de tests. Un código base bien testeado es un multiplicador de fuerza que se acumula con cada tarea que le das a un agente. Un código base sin tests es un impuesto que hace cada tarea más lenta y arriesgada.

Si te llevas una sola cosa de este capítulo, que sea esta: antes de optimizar tus prompts, antes de experimentar con nuevos modelos, antes de construir orquestación elaborada — ve a escribir tests. Escríbelos para el código que tus agentes tocarán. Hazlos rápidos, deterministas, aislados y específicos. Esa inversión rendirá más que cualquier otra cosa en este libro.

Tu suite de tests no es overhead. Es la base sobre la que todo lo demás se construye.

9 Convención Sobre Configuración

Vi al mismo agente producir código digno de una contratación senior en un proyecto y basura imposible de mantener en otro — la misma tarde, en la misma máquina, con el mismo modelo. La diferencia no fue el prompt. Fue el código base.

Dos proyectos. Mismo stack tecnológico — TypeScript, React, PostgreSQL. Mismo tamaño de equipo. Mismo tooling agéntico.

El Proyecto A tiene una disposición de directorios estricta. Cada endpoint de API sigue el mismo patrón: un archivo de handler, un archivo de esquema, un archivo de test, nombrados idénticamente. La capa de base de datos usa un patrón de repositorio consistente. Hay un `CLAUDE.md` en la raíz que describe la arquitectura en dos páginas. Cuando un ingeniero apunta un agente a un ticket — «añade un nuevo endpoint para notificaciones de usuario» — el agente lee los endpoints existentes, sigue el patrón y produce un pull request que parece escrito por alguien del equipo. La revisión lleva tres minutos.

El Proyecto B es del otro tipo. El código base creció orgánicamente durante dos años. Algunos endpoints están en `routes/`, otros en `api/`, otros en `handlers/`. La mitad de las consultas a base de datos usan un ORM, la otra mitad usa SQL directo. No hay manejo de errores consistente — algunas funciones lanzan excepciones, otras devuelven objetos de error, otras devuelven null. El agente mira este código base y hace lo que puede, pero «lo que puede» significa elegir el patrón que vio más recientemente. El código resultante funciona, técnicamente, pero no encaja con nada a su alrededor. La revisión lleva treinta minutos, la mayor parte dedicada a «así no es como lo hacemos aquí.»

La diferencia entre estos proyectos no es talento. No es tooling. Es convención.

Hay un viejo principio en el software: convención sobre configuración. Haz que lo predeterminado sea lo correcto. Reduce el número de decisiones que necesitan tomarse. Cuando todos siguen los mismos patrones, el código se explica solo.

Este principio siempre fue útil para equipos humanos. Para la ingeniería agéntica, es esencial.

Si esto suena como que te estoy diciendo que hagas el trabajo poco glamuroso — escribir documentación, imponer convenciones de nombres, mantener la estructura del proyecto — es que lo estoy haciendo. Y sé cómo se siente. No te hiciste ingeniero para escribir guías de estilo. Pero este es uno de esos momentos donde el oficio te pide que te preocupes por algo que solía sentirse como overhead, porque las implicaciones han cambiado. La convención solía ser una cortesía hacia tu yo futuro. Ahora es el sistema operativo sobre el que corren tus agentes.

9.1 Por Qué a los Agentes les Encanta la Convención

Un agente navegando un código base desconocido hace lo mismo que un nuevo empleado: busca patrones. ¿Dónde viven los tests? ¿Cómo se nombran los archivos? ¿Cuál es la convención de imports? ¿Dónde está la configuración?

Si tu proyecto sigue convenciones fuertes, el agente detecta los patrones rápidamente y produce código que encaja. Si cada archivo es un copo de nieve — nombres diferentes, estructura diferente, estilos diferentes — el agente se pierde. No sabe qué patrón seguir, así que inventa el suyo, y el resultado se siente ajeno.

La convención es *contexto implícito*. Es información que el agente absorbe de la estructura de tu código sin que tengas que explicarla. Cuando tus archivos de test siempre viven junto a los archivos fuente que testean, nombrados `foo.test.ts` junto a `foo.ts`, el agente no necesita que le digan dónde poner un nuevo test. Lee el directorio, ve el patrón y lo sigue. Cuando todos tus handlers de API exportan la misma forma — una

función handler, un esquema, un conjunto de middleware — el agente produce un nuevo handler que exporta exactamente la misma forma.

Esta es la razón por la que los frameworks con opiniones siempre han sido productivos, y por qué son *aún más* productivos en la era agéntica. Rails, Next.js, Laravel — imponen una estructura. Esa estructura no es solo para humanos. Es un lenguaje que el agente habla con fluidez.

9.2 El CLAUDE.md a Fondo

Una convención creciente en la ingeniería agéntica es el archivo `CLAUDE.md`: un documento en la raíz de tu proyecto que le dice al agente lo que necesita saber. No un README para humanos. Un briefing para agentes.

Esta es una de las cosas de mayor apalancamiento que puedes hacer para tu flujo de trabajo agéntico, y la mayoría de los equipos lo omiten o escriben unas líneas vagas y lo dan por terminado. Hablemos de cómo se ve uno bueno de verdad.

Un `CLAUDE.md` sólido tiene cinco secciones:

Visión general del proyecto. Dos o tres frases. Qué hace esto, cuál es el stack tecnológico, cuál es el objetivo de despliegue. Un agente que sabe que está trabajando en «una plataforma de facturación B2B SaaS construida con Go y PostgreSQL, desplegada en Kubernetes» toma decisiones fundamentalmente diferentes que uno que está adivinando.

Comandos de build y test. Cada comando que el agente podría necesitar, listado explícitamente. No «mira el Makefile» — los comandos reales. Los agentes leen la documentación literalmente. Si tu `CLAUDE.md` dice `make test`, el agente ejecutará `make test`. Si dice «ejecuta los tests» sin especificar cómo, el agente adivinará, y podría adivinar mal.

Decisiones de arquitectura. Las cosas que no son obvias desde el código. Por qué elegiste un monorepo. Por qué el servicio de autenticación es separado. Por qué usas event sourcing para el pipeline de pedidos pero CRUD simple para la gestión de usuarios. Estas son las decisiones que dan forma a cada pieza de código nuevo, y un agente que no las conoce las violará constantemente.

Convenciones. Tu estilo. Cómo nombras las cosas. Cómo manejas los errores. Cómo se ve tu orden de imports. Si prefieres retornos tempranos o condicionales anidados. Las cosas que hacen que el código se sienta como que pertenece a *este* proyecto.

Trampas comunes. Dónde están enterrados los cuerpos. La migración de base de datos que siempre debe ejecutarse antes del seed. La variable de entorno que no está en `.env.example` pero es necesaria para el flujo de pagos. El test que es inestable en CI pero no localmente. Todo proyecto tiene estos — escríbelos.

Así se ve un `CLAUDE.md` real para un proyecto de tamaño mediano:

```
# Project: Meridian (billing platform)

TypeScript monorepo (pnpm workspaces). React frontend, Express
API,
PostgreSQL with Drizzle ORM. Deployed to Fly.io.

## Commands
- `pnpm install` — install all dependencies
- `pnpm test` — run all tests (vitest)
- `pnpm test:api` — API tests only
- `pnpm test:web` — frontend tests only
- `pnpm lint` — eslint + prettier check
- `pnpm lint:fix` — auto-fix lint issues
- `pnpm db:migrate` — run pending migrations
- `pnpm db:generate` — generate migration from schema changes
- `pnpm dev` — start all services locally

## Architecture
- /packages/api — Express REST API
- /packages/web — React SPA (Vite)
- /packages/shared — shared types and utilities
- /packages/db — Drizzle schema, migrations, seed data

All API routes follow the pattern:
  routes/{resource}/index.ts — route definitions
  routes/{resource}/handlers.ts — request handlers
  routes/{resource}/schemas.ts — Zod validation schemas
  routes/{resource}/__tests__/ — tests for this resource

## Conventions
```

```
- All errors go through the AppError class (packages/api/src/
errors.ts)
- Never throw raw Error objects in API handlers
- Use Zod schemas for ALL request validation, no manual checks
- Database queries live in packages/db/src/queries/, not in
handlers
- Prefer early returns over deeply nested conditionals
- Import order: node builtins, external deps, internal packages,
relative

## Pitfalls
- The Stripe webhook handler uses raw body parsing — don't add
  json middleware to that route
- Test database must be created manually: createdb meridian_test
- The `BILLING_SECRET` env var isn't in .env.example (it's in
1Password)
- Flaky test: invoice.concurrent.test.ts — known race condition,
  skip locally if it blocks you
```

Eso no es largo. Llevó quizá treinta minutos escribirlo. Pero cada sesión de agente que lee este archivo empieza con más contexto del que la mayoría de los desarrolladores humanos obtienen en su primera semana.

La disciplina más importante: mantenlo actualizado. Un `CLAUDE.md` desactualizado es peor que no tener ninguno, porque el agente confiará en él. Cuando cambies una convención, actualiza el archivo. Cuando añadas un nuevo servicio, añádelo a la sección de arquitectura. Cuando alguien descubra una nueva trampa, documéntala. Trátalo como código — vive en control de versiones, se revisa en PRs, es parte del proyecto.

Algunos equipos van más allá: ponen archivos `CLAUDE.md` también en subdirectorios. Un `packages/api/CLAUDE.md` que cubre patrones específicos de la API. Un `packages/web/CLAUDE.md` que documenta las convenciones de la biblioteca de componentes. Cuanto más profundo va el agente en el proyecto, más contexto específico obtiene. Es como una cebolla de documentación — contexto amplio en la raíz, contexto específico a medida que profundizas.

9.3 Las Convenciones Como Memoria del Agente

Las convenciones son lo más parecido que los agentes tienen a memoria a largo plazo. Una vez que ves esto, cambia cómo piensas sobre todas ellas.

La ventana de contexto de un agente se reinicia en cada sesión. No recuerda lo que hizo ayer. No recuerda la discusión de arquitectura que tuviste la semana pasada. No recuerda que las últimas tres veces que usó `fetch` directamente, le pediste que usara el wrapper del cliente API en su lugar.

Pero la estructura de tu proyecto persiste. Tu nomenclatura de archivos persiste. Tu `CLAUDE.md` persiste. Las reglas de tu linter persisten. Tus patrones de test persisten. Todo lo que codificas en la forma de tu proyecto está ahí cada vez que el agente abre los ojos.

Esto reenmarca las convenciones por completo. No se tratan solo de consistencia para humanos. Son *memoria externa* para agentes. Cada convención que estableces es una lección que el agente no tiene que reaprender.

Piensa en lo que pasa sin convenciones. Sesión uno: el agente crea un nuevo endpoint y pone el manejo de errores inline. Lo corriges en la revisión: «Usamos la clase AppError.» Sesión dos: el agente crea otro endpoint y comete el mismo error, porque no recuerda la sesión uno. Sesión tres: lo mismo. Estás teniendo la misma conversación una y otra vez.

Ahora añade una convención — un patrón documentado de manejo de errores, reforzado por una regla del linter — y el problema desaparece permanentemente. El agente lee el código existente, ve el patrón, lo sigue. El linter detecta cualquier desviación. La lección está codificada en el propio proyecto, no en la memoria de nadie.

Esta es la razón por la que los equipos agénticos más efectivos se obsesionan con convenciones que parecen tediosas. Orden de imports consistente. Nomenclatura de archivos estricta. Firmas de funciones estándar. Estas no son preferencias estéticas — son memoria. Son la sabiduría acumulada del equipo, almacenada en un formato que sobrevive a los reinicios de ventana de contexto.

La analogía a la que vuelvo constantemente: las convenciones son para los agentes lo que el conocimiento institucional es para las organizaciones. Una empresa donde todo vive en la cabeza de una persona es frágil. Una empresa con procesos y documentación sólidos es resiliente. Lo mismo se aplica a los códigos base. Un proyecto donde «cómo hacemos las cosas» vive solo en la memoria del desarrollador senior es frágil. Un proyecto donde está codificado en la estructura, el tooling y la documentación es resiliente — para humanos y agentes por igual.

9.4 Convenciones Prácticas Que Ayudan a los Agentes

Nomenclatura de archivos consistente. Si tus rutas de API viven en `routes/`, tus modelos en `models/`, y tus tests en `__tests__/` — el agente puede navegar tu proyecto sin mapa. Nombra los archivos según lo que contienen. Mantenlo aburrido.

Formateadores y linters. Herramientas como Prettier, ESLint, `gofmt` o Ruff no son solo para estilo de código — son barandillas que aseguran que el código generado por agentes coincida con los estándares de tu proyecto automáticamente. Ejecútalos al guardar, ejecútalos en CI, hazlos innegociables.

Disposición de proyecto estándar. Ya sea la disposición estándar de Go, las convenciones de Rails o la estructura propia de tu equipo — elige una y cíñete a ella. Un `justfile` o `Makefile` en la raíz que liste los comandos estándar (`build`, `test`, `lint`, `dev`) les da a los agentes un punto de entrada a cualquier proyecto.

Archivos pequeños y enfocados. Los agentes trabajan mejor con archivos de menos de unos cientos de líneas. Cuando un archivo contiene una sola preocupación — un componente, un módulo, un conjunto de funciones relacionadas — el agente puede leerlo, entenderlo y modificarlo sin tener que vadear código no relacionado.

Mensajes de commit descriptivos. Los agentes leen el historial de git. Un commit que dice «fix bug» no enseña nada. Un commit que dice «fix race condition in session cleanup when WebSocket disconnects during

auth» le da al agente contexto sobre qué era importante, qué era frágil y cómo piensa el equipo sobre los problemas.

Manejo de errores consistente. Elige un patrón y aplícalo en todas partes. Clases de error personalizadas con códigos de error. Un handler de errores central. Una forma estándar de respuesta de error. Cuando el agente encuentra un caso de error en código nuevo, debería tener cero ambigüedad sobre cómo manejarlo. Si tu proyecto usa tres enfoques diferentes de manejo de errores en tres archivos diferentes, el agente producirá un cuarto.

Formatos de respuesta de API estándar. Cada endpoint devuelve la misma forma: `{ data, error, meta }` o lo que elijas. Los códigos de estado siguen las mismas reglas en todas partes. La paginación funciona igual en cada endpoint de lista. Esta es el tipo de consistencia en la que los agentes destacan manteniéndola — pero solo si la convención es clara desde el código existente.

Convenciones de logging. Logging estructurado con campos consistentes. Cada entrada de log incluye un ID de petición, un timestamp y un nivel de severidad. Cada log de error incluye el código de error y una traza de pila. Cuando el agente añade logging a código nuevo — y debería — los logs deberían verse idénticos a todos los demás logs del sistema.

Estructura de directorios que cuente una historia. Un agente debería poder hacer `ls` en el nivel superior de tu proyecto y entender la arquitectura. Así se ve un proyecto bien estructurado desde la perspectiva de un agente:

```
src/
  routes/         # HTTP handlers — one file per resource
  services/       # Business logic — one file per domain concept
  repositories/   # Database access — one file per table/entity
  middleware/      # Express/Koa middleware
  utils/          # Pure utility functions
  types/          # Shared TypeScript types
  errors/         # Custom error classes
tests/
  fixtures/       # Test data factories
  helpers/        # Test utilities
```

```
migrations/       # Database migrations (numbered)
scripts/          # One-off and maintenance scripts
```

Cada nombre de directorio es un sustantivo. Cada archivo dentro contiene exactamente lo que el nombre del directorio promete. No hay lugar para confusión sobre dónde debe ir el código nuevo. Un agente leyendo esta estructura sabe inmediatamente: «Necesito añadir una nueva consulta de base de datos, eso va en `repositories/`. Necesito un nuevo endpoint, eso empieza en `routes/`.»

Compara eso con un proyecto donde las consultas de base de datos están dispersas en archivos de handlers, la lógica de negocio vive en funciones utilitarias y hay tres directorios diferentes que parecen contener «helpers.» El agente pondrá código en *algún* lugar, pero probablemente no será el lugar *correcto*.

9.5 El Impuesto de la Convención

Seamos honestos sobre el coste. Establecer convenciones lleva tiempo y se siente como overhead — especialmente al principio.

Escribir un `CLAUDE.md` lleva una tarde. Configurar linters y formateadores lleva un día. Refactorizar un código base inconsistente para seguir un solo patrón lleva una semana, quizá más. Imponer convenciones en la revisión de código significa ralentizar PRs que «funcionan bien» pero no siguen el estándar.

Hay una tentación real de saltarse todo esto. El código funciona. Los tests pasan. ¿Por qué pasar tiempo en convenciones de nomenclatura cuando hay funcionalidades que entregar?

La respuesta honesta: si eres un desarrollador solo construyendo un prototipo desechable, el impuesto de convención probablemente no vale la pena. Muévete rápido, entrégalo, tíralo.

Pero en el momento en que un segundo par de ojos toca tu código base — humano *o* agente — las convenciones empiezan a pagarse solas. Y los retornos se acumulan.

La primera vez que estableces una convención, te cuesta una hora. Cada vez subsiguiente que un agente sigue esa convención en lugar de preguntarte cómo manejar algo, ahorras cinco minutos. Haz las cuentas: después de doce sesiones de agente, la convención se ha pagado sola. Después de cien sesiones, has ahorrado *horas*.

Esto se acumula en todo el equipo. Cinco ingenieros, cada uno ejecutando múltiples sesiones de agente por día, todos beneficiándose de las mismas convenciones. La inversión inicial de una tarde de un ingeniero ahorra al equipo cientos de horas a lo largo de un año.

Los equipos que invierten en convenciones temprano parecen más lentos al principio. Pasan tiempo en cosas «aburridas» — configuraciones de linter, estructura de proyecto, documentación. Pero tres meses después, sus agentes producen código que requiere revisión mínima. Seis meses después, entregan el doble de rápido que el equipo que se saltó el trabajo de convención. Un año después, la brecha es embarazosa.

Esta es la misma dinámica que la deuda técnica, solo que invertida. La convención es *riqueza* técnica — y como la riqueza financiera, cuanto antes empieces a invertir, más dramático es el efecto acumulativo.

El mayor error que veo es que los equipos esperan hasta que su código base es un desastre antes de intentar establecer convenciones. Ese es el momento más caro para hacerlo. El momento más barato es al inicio de un proyecto — pero el segundo más barato es hoy.

9.6 El Efecto Acumulativo

Cada convención que estableces, cada estándar que impones, cada pieza de documentación que mantienes — todo se acumula. Cada una hace que los agentes sean ligeramente más efectivos, ligeramente más autónomos, ligeramente más propensos a producir código que encaja en tu proyecto como un guante.

Pero la acumulación no es solo aditiva. Las convenciones interactúan. Una convención de nomenclatura consistente *más* una estructura de directorios estándar *más* un `CLAUDE.md` que describe la arquitectura — juntas, estas le dan al agente un modelo mental de todo el proyecto. Sabe dónde encontrar

cosas, cómo nombrarlas y cómo encajan. Quita cualquiera de las tres, y la efectividad del agente cae desproporcionadamente.

Esta es la razón por la que las convenciones a medias son casi tan malas como no tener convenciones. Un proyecto con nomenclatura de archivos consistente pero estructura de directorios caótica envía señales mixtas. El agente ve orden en una dimensión y caos en otra, y no sabe en qué señal confiar.

Los ingenieros que invierten en convención temprano no solo tienen códigos base más limpios. Tienen códigos base que están listos para el futuro agéntico. Sus agentes producen mejores resultados. Sus revisiones son más rápidas. Sus ciclos de iteración son más ajustados.

La convención no es trabajo glamuroso. Es el capítulo menos emocionante de este libro. Pero es el que hace que todos los demás capítulos funcionen. Los sandboxes, las estrategias de testing, la orquestación multi-agente — todo funciona mejor cuando el código base subyacente es consistente, documentado y convencional.

Tu código base es el entorno en el que viven tus agentes. Hazlo legible. Hazlo predecible. Hazlo aburrido. Los agentes te lo agradecerán escribiendo código que parece que pertenece.

10 LLMs Locales vs. Comerciales

Un amigo mío — ingeniero senior en una startup Serie B — me escribió un viernes por la tarde. «Acabo de recibir nuestra primera factura real de API. Dos mil doscientos dólares. Solo de *marzo*.» Había estado ejecutando Claude Code en su equipo de ocho ingenieros, cada uno iterando en funcionalidades, depurando, refactorizando. Nadie había establecido presupuestos de tokens. Nadie estaba vigilando el contador. Los agentes funcionaban de maravilla, y la factura era escalofriante.

Esa misma semana, hablé con una ingeniera en una empresa de salud en Múnich. Estaba ejecutando Llama 70B en un servidor GPU local porque su pipeline de datos de pacientes no podía tocar APIs externas. No «no debería» — *no podía*. Su equipo de cumplimiento lo había dejado claro por escrito. Obtenía resultados decentes para tareas enfocadas, pero cada vez que necesitaba razonamiento complejo multi-archivo, el modelo se desmoronaba y acababa haciendo el trabajo manualmente.

Estas dos historias son los extremos de la misma pregunta: ¿dónde se ejecuta el modelo? Suena como una decisión de infraestructura. Realmente es una decisión sobre confianza, coste, capacidad y — cada vez más — cómo arquitectas todo tu flujo de trabajo agéntico.

10.1 LLMs Comerciales: La Frontera

Los modelos comerciales — Claude, GPT, Gemini — son donde vive la frontera. Si estás haciendo ingeniería agéntica seria, probablemente has pasado la mayor parte de tu tiempo aquí. Hay buenas razones para ello.

10.1.1 Ventanas de Contexto

El contexto lo es todo para los agentes. Un agente no solo lee tu prompt — lee archivos, salidas de herramientas, mensajes de error, resultados de

tests y su propio razonamiento previo. Una sola tarea compleja puede llenar fácilmente 50.000 tokens de contexto, y una sesión de depuración multi-paso puede superar los 100.000.

Los modelos comerciales de frontera ofrecen ventanas de contexto de 128K tokens y más. Esto importa enormemente para el trabajo agéntico porque el agente necesita mantener tu código base en su cabeza. Cuando el contexto se agota, el agente empieza a olvidar archivos que leyó antes, decisiones que tomó antes, errores que vio antes. Se degrada de un ingeniero capaz a alguien con amnesia.

Los modelos locales típicamente tienen un tope de 8K-32K de contexto en la práctica, incluso si técnicamente soportan más sobre el papel. La calidad de la atención se degrada a medida que empujas hacia el límite. Un modelo comercial a 100K de contexto sigue razonando bien. Un modelo local a 32K de contexto a menudo pierde el hilo.

10.1.2 Calidad del Uso de Herramientas

Los agentes viven y mueren por el uso de herramientas. Leer archivos, escribir código, ejecutar comandos, buscar en códigos base — estos no son extras opcionales. Son el bucle principal. Y la calidad del uso de herramientas varía dramáticamente entre modelos.

Los modelos comerciales de frontera han sido específicamente entrenados y afinados para llamadas a herramientas. Formatean los argumentos correctamente, encadenan llamadas lógicamente, se recuperan con gracia cuando falla una llamada. Saben cuándo leer un archivo antes de editarlo. Saben cuándo ejecutar tests después de hacer cambios.

Los modelos más pequeños — incluyendo la mayoría de los modelos locales — son menos fiables aquí. Alucinarán rutas de archivos, olvidarán pasar argumentos requeridos, llamarán herramientas en el orden equivocado o se quedarán atascados en bucles llamando a la misma herramienta fallida una y otra vez. La diferencia no es sutil. En una tarea compleja que implica diez o quince llamadas a herramientas, un modelo de frontera podría tener éxito el 90% de las veces donde un modelo local tiene éxito el 40%.

10.1.3 Seguimiento de Instrucciones

Cuando le dices a un modelo de frontera «solo modifica archivos en el directorio `src/auth/`» o «no cambies la API pública, solo la implemen-

tación interna,» generalmente escucha. Sigue los system prompts, respeta las restricciones y se mantiene dentro de los límites.

Los modelos más pequeños se desvían. Empezarán bien, luego gradualmente ignorarán tus restricciones a medida que la conversación se alarga y el contexto se llena. Este es un problema real para el trabajo agéntico, donde la precisión importa. Un agente que «mayormente» sigue instrucciones ocasionalmente reescribirá un archivo que no le pediste que tocara, o refactorizará algo que explícitamente le dijiste que dejara en paz. En un entorno sandbox esto es solo molesto. Sin sandbox es peligroso.

10.1.4 Eligiendo el Modelo Comercial Adecuado

No todos los modelos comerciales son intercambiables, incluso en la frontera. Esto es lo que he encontrado en la práctica:

Para refactorizaciones complejas multi-archivo y trabajo arquitectónico — usa el modelo más capaz que puedas permitirte. Aquí es donde la calidad de razonamiento más importa. La diferencia de coste entre un modelo de nivel medio y uno de primer nivel es unos pocos dólares por tarea. La diferencia de calidad es a menudo la diferencia entre un diff limpio y un desastre que tienes que rehacer manualmente.

Para tareas enfocadas de un solo archivo — escribir tests, implementar una función bien definida, corregir un bug claro — los modelos de nivel medio rinden casi tan bien como los de primer nivel, a una fracción del coste. La tarea tiene suficiente alcance como para que el modelo no necesite malabarear muchas preocupaciones a la vez.

Para trabajo de alto volumen y baja complejidad — generar boilerplate, formatear código, escribir mensajes de commit — el modelo más barato que pueda seguir instrucciones es la elección correcta. Ejecutarás estas tareas cientos de veces. Los ahorros por token se acumulan.

El error que veo más a menudo es usar un solo modelo para todo. Eso es como conducir un Fórmula 1 al supermercado. Ajusta el modelo a la tarea.

10.2 LLMs Locales: La Imagen Completa

Ejecutar un modelo localmente — Llama, Mistral, Qwen, DeepSeek, Codestral, o cualquiera de los modelos de pesos abiertos vía Ollama, llama.cpp o vLLM — te da algo que las APIs comerciales no pueden: soberanía completa sobre los datos y coste marginal cero por token.

Tu código nunca sale de tu máquina. Puedes alimentarlo con código propietario, documentos internos, logs de producción, datos de clientes, API keys que dejaste accidentalmente en una configuración — nada de ello cruza una frontera de red. Y una vez que el modelo está descargado, cada inferencia es gratis. Ejecútalo mil veces, ejecútalo toda la noche, nadie te envía una factura.

Pero seamos honestos sobre la experiencia, porque el marketing alrededor de los modelos locales a menudo no lo es.

10.2.1 La Realidad del Hardware

La pregunta del hardware es la primera que todos hacen, y la respuesta es menos glamurosa de lo que sugieren los posts de blog de «¡ejecuta IA localmente!».

MacBook Pro con Apple Silicon (M2/M3/M4 Pro o Max, 32-64GB RAM). Este es el punto óptimo para desarrolladores individuales. Puedes ejecutar un modelo de 7B-14B parámetros cómodamente, y un modelo de 32B si tienes la RAM. La inferencia será notablemente más lenta que una API comercial — quizá 15-30 tokens por segundo para un modelo 14B, comparado con 80+ de una API comercial. Un modelo 70B técnicamente correrá en una máquina de 64GB pero será lo suficientemente lento como para poner a prueba tu paciencia. Estarás esperando 10-20 segundos por respuestas que toman 2 segundos de una API.

Servidor GPU dedicado (NVIDIA RTX 4090 o A100). Aquí es donde la inferencia local se vuelve genuinamente rápida. Una 4090 con 24GB de VRAM puede ejecutar un modelo 14B a velocidades cercanas a las comerciales. Una A100 con 80GB puede ejecutar un modelo 70B cómodamente. Pero ahora estás manteniendo hardware. Estás lidiando con drivers CUDA, gestión de VRAM y el ocasional «por qué está gritando el ventilador de mi GPU a las 3am».

Hardware de consumo (MacBook Air de 16GB, máquinas antiguas, sin GPU discreta). Estás limitado a modelos 7B, y la experiencia será mediocre. La inferencia es lenta, los modelos son demasiado pequeños para trabajo agéntico serio, y pasarás más tiempo esperando que trabajando. No recomendaría esto para nada más allá de la experimentación.

El resumen honesto: los modelos locales son prácticos para desarrolladores con un MacBook Pro reciente o una GPU dedicada. Por debajo de ese umbral de hardware, tendrás una experiencia frustrante. Por encima, tendrás una herramienta genuinamente útil — solo una herramienta diferente de una API comercial.

10.2.2 Dónde Brillan los Modelos Locales

Los modelos locales no son solo «modelos comerciales peores.» Hay flujos de trabajo donde genuinamente tienen más sentido:

Tareas de alta frecuencia y bajo riesgo. Generar docstrings, escribir mensajes de commit, crear código boilerplate, formatear datos. Estas tareas no necesitan un modelo genio — necesitan un modelo rápido y gratis. Ejecutarlas localmente significa que puedes disparar y olvidar sin pensar en el coste.

Códigos base sensibles. Cubriremos esto más en la sección de privacidad, pero este es un caso de uso legítimo y creciente. Si tu código no puede salir de tu red, los modelos locales son la única opción, y «suficientemente bueno» es infinitamente mejor que «no disponible.»

Desarrollo offline. En un avión, en un tren, en un búnker — tu modelo local funciona sin WiFi. Esto suena menor hasta que estás en un vuelo de doce horas intentando depurar algo.

Experimentación y aprendizaje. Cuando estás experimentando con frameworks de agentes, probando estrategias de prompts o construyendo integraciones de herramientas personalizadas, quemar créditos de API en prueba y error se siente despilfarrador. Un modelo local te deja iterar libremente.

10.2.3 Dónde Fallan los Modelos Locales

Vale la pena ser específico sobre los modos de fallo, porque «es menos capaz» es demasiado vago para ser útil.

Razonamiento multi-archivo. Pídele a un modelo local de 14B que trace un bug a través de cuatro archivos y un esquema de base de datos, y perderá el hilo. Podría encontrar el archivo correcto pero malinterpretar la interacción entre componentes. Esta es la brecha práctica más grande.

Tareas de horizonte largo. Las tareas agénticas que implican muchos pasos — leer, planificar, implementar, testear, depurar, iterar — requieren que el modelo mantenga una intención coherente a lo largo de un contexto largo. Los modelos locales tienden a desviarse. Olvidan el plan. Revisitan decisiones que ya tomaron. Se quedan atascados en bucles.

Revisión de código matizada. «Este código es correcto pero el enfoque es incorrecto» es un juicio que requiere comprensión profunda. Los modelos locales tienden hacia el análisis superficial — detectarán problemas de sintaxis y bugs obvios pero pasarán por alto problemas arquitectónicos.

Orquestación compleja de herramientas. Cuando una tarea requiere diez o más llamadas a herramientas encadenadas — leer un archivo de test, ejecutarlo, leer el error, encontrar el archivo fuente, entender el contexto, hacer un cambio, ejecutar el test de nuevo — los modelos locales tropiezan más frecuentemente en cada paso, y la tasa de error se acumula.

Nada de esto significa que los modelos locales son inútiles. Un modelo de 32B ejecutándose localmente puede manejar un rango sorprendente de tareas bien acotadas. Pero necesitas acotar las tareas en consecuencia. Pedirle a un modelo local que haga lo que hace un modelo de frontera es prepararlo para fracasar.

10.3 El Coste del Trabajo Agéntico

Una sola sesión de codificación agéntica puede consumir fácilmente 100.000-500.000 tokens. Esto sorprende a la gente cuando ven los números por primera vez. No porque estés chateando — porque el agente está *trabajando*.

Considera lo que pasa cuando un agente aborda una corrección de bug moderadamente compleja. Lee tres o cuatro archivos para entender el contexto (quizá 8.000 tokens de entrada). Razona sobre el problema (2.000 tokens de salida). Lee dos archivos más (4.000 tokens). Escribe

una corrección (1.000 tokens). Ejecuta los tests (llamada a herramienta, 500 tokens de salida). Los tests fallan. Lee el error (1.000 tokens). Revisa la corrección (1.500 tokens). Ejecuta los tests de nuevo. Pasan. Lee el diff para verificar (1.000 tokens). Total: quizá 25.000 tokens para una corrección de bug sencilla.

Ahora considera una tarea de refactorización compleja. El agente podría leer veinte archivos, generar un plan, implementar cambios en ocho archivos, ejecutar la suite de tests cuatro veces, depurar dos regresiones y escribir tests nuevos. Eso son fácilmente 200.000 tokens. A precios de modelo de frontera, eso está entre $3 y $15 dependiendo del modelo y la proporción entrada/salida.

Aquí hay rangos de coste aproximados que he visto en la práctica, basados en usar modelos comerciales de frontera:

- **Corrección de bug simple o funcionalidad pequeña:** $0.30–$1.00
- **Funcionalidad moderada con tests:** $2–$5
- **Refactorización compleja multi-archivo:** $5–$15
- **Implementación de funcionalidad grande:** $15–$40+
- **Sesión de depuración exploratoria que se alarga:** $10–$30

Multiplica estos números por un equipo de ingenieros, cada uno ejecutando varias tareas agénticas por día, y estás mirando dinero real. ¿La factura de $2.200 de mi amigo? Es bastante acertada para ocho ingenieros activos.

10.3.1 Estrategias para Gestionar el Coste

La solución no es dejar de usar agentes. Es ser inteligente al respecto.

Establece presupuestos de tokens. La mayoría de los frameworks de agentes te dejan establecer un límite máximo de tokens por tarea. Esto no es solo control de costes — también es una señal de calidad. Si un agente quema 500.000 tokens en una tarea que debería llevar 50.000, algo ha salido mal. El agente está atascado, en bucle o malinterpretando la tarea. Un presupuesto lo fuerza a fallar rápido en lugar de espiralar.

Usa enrutamiento de modelos. No envíes cada tarea al modelo más caro. Profundizaremos más en la siguiente sección, pero la versión corta: usa un modelo barato y rápido para exploración y lectura de código, y

un modelo capaz y caro para razonamiento e implementación. Solo esto puede recortar costes un 50-70%.

Cachea agresivamente. Si tu framework de agentes soporta caché de prompts, actívalo. Gran parte del contexto de un agente se repite entre turnos — el mismo system prompt, el mismo contexto de proyecto, los mismos archivos leídos recientemente. El caché evita reprocesar esos tokens en cada turno.

Acota las tareas estrictamente. Una tarea bien acotada («arregla el bug de zona horaria en `billing/invoice.py`, el test está en `tests/test_invoice.py`») es más barata que una vaga («arregla los bugs de facturación»). Acotar no es solo buena ingeniería — es control de costes. El agente lee menos archivos, hace menos llamadas exploratorias a herramientas y converge más rápido.

Revisa tus fallos. Cuando una tarea falla o consume demasiados tokens, averigua por qué. ¿El prompt era vago? ¿Al agente le faltaba contexto que necesitaba? ¿El modelo no era lo suficientemente capaz para la tarea? Cada fallo es una oportunidad de ajuste.

10.3.2 Gestionando Costes en Todo el Equipo

El control de costes individual es una cosa. Gestionar el gasto en un equipo de ocho o doce ingenieros, cada uno ejecutando agentes todo el día, es un problema diferente. Es la diferencia entre vigilar tu propia dieta y dirigir una cocina de restaurante.

Presupuestos por ingeniero. Establece un presupuesto mensual de tokens por ingeniero — no para restringir, sino para hacer visible los costes. Cuando todos pueden ver su propio gasto, el comportamiento cambia naturalmente. La gente empieza a acotar tareas con más cuidado, a elegir el modelo adecuado para el trabajo, a matar sesiones descontroladas antes. Un punto de partida razonable: $200-400 por mes por ingeniero para un equipo que usa agentes activamente. Algunos ingenieros consistentemente vendrán por debajo del presupuesto. Otros tendrán picos durante semanas de depuración intensa. Eso está bien — el punto es la conciencia, no la imposición.

Panel y visibilidad. No puedes gestionar lo que no puedes ver. Rastrea el gasto por ingeniero, por proyecto, por tipo de tarea. La mayoría de

los proveedores de API te dan desgloses de uso por API key, y si asignas keys por ingeniero o por proyecto, los datos ya están ahí. Canalízalo a un panel que el equipo pueda ver — incluso una hoja de cálculo compartida funciona para empezar. El ingeniero que descubre que quemó $80 en una sola sesión de depuración naturalmente empezará a acotar tareas con más cuidado la próxima vez. No necesitas tener una conversación al respecto. El número habla solo.

Alertas y cortacircuitos. Establece alertas al 50% y 80% del presupuesto mensual para que nadie se lleve una sorpresa. Más importante, establece un límite duro de tokens por sesión como cortacircuito. Si una sola sesión de agente excede 500K tokens, algo ha salido mal — el agente está en bucle, la tarea es demasiado vaga o el modelo está luchando con algo más allá de su capacidad. Mátalo e investiga. Esto es lo que previene el escenario de «factura sorpresa de $2.200» del inicio de este capítulo. No necesitas detectar cada sesión desbocada manualmente. Los límites automatizados hacen el trabajo.

La conversación del ROI. En algún momento, alguien en dirección preguntará por qué la factura de API tiene cinco cifras. Necesitas los números listos. Enmárcalo así: un ingeniero senior cuesta $150K al año con todo incluido. Si los agentes lo hacen un 30% más productivo, eso son $45K de valor. Los costes de agentes son $3-4K por año por ingeniero. El ROI es aproximadamente 10x. Incluso si eres conservador — digamos 15% de ganancia de productividad en lugar de 30% — los números siguen funcionando cómodamente. La parte más difícil es medir esa ganancia de productividad con precisión. Probablemente no puedas, no con rigor científico. Pero puedes rastrear señales concretas: tiempo hasta fusionar PRs, número de tareas completadas por sprint, reducción en cambios de contexto. Combina esas métricas con los datos de coste y tienes una historia con la que finanzas puede trabajar.

El coste como señal de calidad. Alto consumo de tokens en una tarea no es solo caro — es una señal de que algo salió mal. La tarea estaba mal acotada, el contexto era insuficiente o el modelo no era lo suficientemente capaz para el trabajo. Empieza a rastrear coste por tipo de tarea a lo largo del tiempo. Si los costes de una categoría particular tienden al alza, investiga — tus prompts pueden haberse desviado, o el código base se

ha vuelto lo suficientemente complejo como para necesitar un enfoque diferente. Si los costes tienden a la baja, eso es tu equipo mejorando en ingeniería agéntica. Están escribiendo prompts más ajustados, eligiendo mejores modelos y acotando tareas más efectivamente. Los datos de coste, bien usados, se convierten en un espejo de la habilidad del equipo.

10.4 Privacidad y Cumplimiento

Parte del código genuinamente no puede salir del edificio. Esto no es paranoia — es la ley.

Los contratistas gubernamentales que trabajan en proyectos clasificados o controlados por exportación no pueden enviar código fuente a APIs de terceros, punto. Los requisitos de residencia de datos no son sugerencias. Vienen con sanciones penales.

Las empresas de salud que manejan datos de pacientes están sujetas a HIPAA, GDPR o regulaciones equivalentes. Si tu código toca registros de pacientes — incluso fixtures de test con datos falsos realistas que un oficial de cumplimiento podría cuestionar — necesitas pensar cuidadosamente sobre qué sale por el cable.

Las instituciones financieras tienen su propio laberinto de regulaciones. Cumplimiento SOX, PCI-DSS, requisitos de auditoría interna — los detalles varían, pero el tema es consistente: los datos se quedan dentro de límites controlados.

Y luego está el simple secreto competitivo. Tus algoritmos propietarios, tu salsa secreta, tu ventaja competitiva — enviar eso a los servidores de otra persona requiere confiar en que no entrenarán con ello, no lo registrarán, no serán vulnerados. La mayoría de los proveedores de API comerciales ofrecen garantías contractuales sólidas sobre esto. Pero «garantías contractuales sólidas» e «imposible de vulnerar» son cosas diferentes, y algunos equipos de seguridad no están dispuestos a aceptar la brecha.

Para todos estos casos, los modelos locales no son un lujo. Son la única opción.

El compromiso es real: estás aceptando capacidad reducida del modelo a cambio de control absoluto de los datos. Un modelo local de 32B haciendo un trabajo mediocre en tu código base clasificado es infinitamente más útil que un modelo de frontera que no tienes permiso de usar. Y para tareas enfocadas y bien acotadas — del tipo que deberías estar escribiendo de todas formas — la brecha de calidad es a menudo menor de lo que esperarías.

10.4.1 El Punto Medio: Despliegues Privados

Vale la pena señalar que hay un camino intermedio emergente. Los proveedores cloud ahora ofrecen despliegues de modelos privados — instancias dedicadas de modelos de frontera ejecutándose dentro de tu VPC, con garantías contractuales de que tus datos permanecen dentro de tu perímetro y no se usan para entrenamiento. AWS Bedrock, Azure OpenAI, Google Cloud Vertex AI todos ofrecen versiones de esto.

Estos no son baratos. Estás pagando por computación dedicada, no por infraestructura de API compartida. Pero para organizaciones grandes que necesitan capacidad de frontera y control estricto de datos, esta es cada vez más la respuesta. Obtienes calidad de modelo comercial con garantías de privacidad de modelo local — por un precio.

10.5 Enrutamiento de Modelos en la Práctica

La verdadera pregunta no es «¿local o comercial?» Es «¿qué modelo, para qué parte del flujo de trabajo?»

Una configuración agéntica sofisticada enruta diferentes partes del flujo de trabajo a diferentes modelos. Esto no es teórico — equipos están haciendo esto hoy, y es la dirección hacia la que se mueve el ecosistema.

10.5.1 El Patrón de Enrutamiento

Piensa en lo que un agente realmente hace durante una tarea típica:

1. **Exploración** — leer archivos, buscar en el código base, entender la estructura. Esto es trabajo de alto volumen y bajo razonamiento. El modelo necesita decidir qué archivos leer y extraer información relevante, pero no está haciendo pensamiento profundo.

2. **Planificación** — analizar el problema, considerar enfoques, decidir una estrategia. Esto requiere razonamiento fuerte. Es la parte donde la calidad del modelo más importa.

3. **Implementación** — escribir los cambios de código reales. Esto requiere buena generación de código y la capacidad de seguir el plan del paso 2.

4. **Verificación** — ejecutar tests, leer errores, decidir si el trabajo está terminado. Razonamiento moderado, uso intensivo de herramientas.

5. **Iteración** — si la verificación falla, volver y ajustar. Esto requiere entender el fallo y conectarlo de vuelta con la implementación.

No todos estos pasos necesitan el mismo modelo. El paso 1 puede ser manejado por un modelo pequeño, rápido y barato — incluso uno local. Es leer archivos y reportar lo que hay en ellos. El paso 2 es donde quieres el modelo de frontera — este es el razonamiento caro que justifica el coste de API. Los pasos 3-5 a menudo pueden ser manejados por un modelo de nivel medio, ya que el pensamiento difícil ya está hecho y el modelo está ejecutando un plan.

10.5.2 Cómo Se Ve Esto

En la práctica, el enrutamiento de modelos puede ser tan simple como configurar tu framework de agentes para usar diferentes modelos para diferentes tipos de tareas:

```
# Pseudocode — the exact config depends on your framework
exploration_model: "local/qwen-14b"       # Free, fast, good
enough for reading
reasoning_model: "claude-sonnet"           # Frontier reasoning
for planning
implementation_model: "claude-haiku"       # Fast, cheap, follows
plans well
```

O puede ser dinámico — un clasificador ligero que mira el paso actual y enruta en consecuencia. Algunos frameworks están empezando a incorporar esto nativamente. Otros requieren que lo conectes tú mismo.

La economía es convincente. Si el 60% de los tokens de un agente se gastan en exploración y tareas simples, y enrutas esas a un modelo que es 10x

más barato, has recortado tu coste total más de la mitad sin ninguna reducción de calidad en las partes que importan.

10.5.3 Enrutamiento para Privacidad

El enrutamiento de modelos también resuelve el problema de privacidad de forma más elegante que un enfoque de todo o nada.

Digamos que estás construyendo una aplicación de salud. Los modelos de datos y la lógica de negocio tocan datos de pacientes — eso necesita quedarse local. Pero los componentes de frontend, la configuración de build, el pipeline de CI — esos no contienen datos sensibles. No hay razón por la que no puedas usar un modelo comercial de frontera para las partes no sensibles mientras enrutas el trabajo sensible a un modelo local.

Esto requiere tooling que sea consciente de los límites de sensibilidad — qué archivos pueden ir a lo externo, cuáles no. Ese tooling todavía está madurando, pero el patrón es claro. En lugar de «todo local» o «todo comercial,» obtienes «local donde importa, comercial en todo lo demás.»

10.6 Empezando Sin Arruinarte

No necesitas presupuesto para empezar a aprender ingeniería agéntica. Necesitas un portátil y una tarde. Aquí hay una rampa práctica de coste cero a productividad real.

10.6.1 El Nivel Gratuito

La mayoría de los proveedores comerciales ofrecen niveles gratuitos o créditos de prueba. A principios de 2026, puedes empezar con Claude, GPT o Gemini sin introducir una tarjeta de crédito. Los niveles gratuitos tienen limitaciones de tasa y restricciones de contexto, pero son más que suficientes para aprender los fundamentos — escribe tu primer prompt agéntico, mira a un agente iterar sobre un fallo de test, familiarízate con el bucle de retroalimentación.

Combina una API key de nivel gratuito con un framework de agentes open source — el nivel gratuito de Claude Code, o Aider con su soporte de modelos gratuitos — y tienes una configuración agéntica completa a coste cero. No será rápida. No manejará trabajo complejo multi-archivo. Pero

te enseñará todo lo de los capítulos dos al cinco de este libro sin gastar un céntimo.

10.6.2 El Camino Local-Primero

Si tienes un MacBook Pro con 16GB o más de RAM, puedes ejecutar modelos útiles localmente gratis, para siempre.

Instala Ollama — lleva un solo comando. Descarga un modelo — `ollama pull qwen2.5-coder:14b` toma unos minutos. Apunta un framework de agentes hacia él. Ahora tienes una configuración agéntica de codificación sin costes de API, sin límites de tasa y sin datos saliendo de tu máquina.

La experiencia no igualará a un modelo comercial de frontera. Las ventanas de contexto son más pequeñas. El razonamiento multi-archivo es más débil. El uso de herramientas es menos fiable. Pero para tareas enfocadas y bien acotadas — «escribe tests para esta función,» «refactoriza esta clase para usar inyección de dependencias,» «añade manejo de errores a este endpoint» — un modelo local de 14B es sorprendentemente capaz. Y como es gratis, puedes iterar sin vigilar el contador.

Un stack práctico para ingeniería agéntica de coste cero:

- **Ollama** para servir modelos (gratis, local)
- **Qwen 2.5 Coder 14B** o **DeepSeek Coder V2** para tareas de código
- **Aider** o **Claude Code** (con soporte de modelo local) como framework de agentes
- Un proyecto con suite de tests (el agente necesita retroalimentación)

Eso es todo. Sin suscripción. Sin API key. Sin computación cloud. Llegarás al techo eventualmente — una tarea que necesita una ventana de contexto más grande, o razonamiento multi-archivo que el modelo local no puede manejar. Ahí es cuando recurres a un modelo comercial. Pero la configuración gratuita te llevará más lejos de lo que esperas.

10.6.3 El Punto Óptimo: $20/Mes

Cuando estés listo para gastar dinero, la configuración más rentable que he encontrado es una combinación de modelos locales para exploración y una suscripción comercial para razonamiento.

La mayoría de los proveedores comerciales ofrecen un plan de desarrollador en el rango de $20/mes que incluye una asignación generosa de tokens.

Usa esto para las tareas que realmente necesitan capacidad de frontera — depuración compleja, refactorización multi-archivo, planificación arquitectónica. Usa tu modelo local para todo lo demás — leer código, generar boilerplate, escribir mensajes de commit, pasar por iteraciones de tests.

Este enfoque híbrido típicamente cubre el 80-90% del trabajo agéntico de un desarrollador solo. El modelo local maneja las tareas de alto volumen y bajo razonamiento. El modelo comercial maneja los momentos que necesitan inteligencia genuina. Tu factura mensual se mantiene predecible, y no estás eligiendo entre calidad y coste — estás usando cada uno donde tiene sentido.

10.6.4 Escalando Intencionalmente

El error es empezar con la opción más cara y optimizar después. Empieza gratis. Aprende los flujos de trabajo. Entiende dónde realmente importa la calidad del modelo y dónde «suficientemente bueno» es suficientemente bueno. Luego gasta dinero en las brechas específicas que las herramientas gratuitas no pueden llenar.

Para cuando estés gastando dinero real en llamadas a API, sabrás exactamente por qué estás pagando — y más importante, por qué *no* estás pagando. Ese conocimiento vale más que cualquier cantidad de créditos.

10.7 El Panorama Está Cambiando

Dentro de seis meses, los detalles de este capítulo estarán desactualizados. Los números de coste cambiarán. Las brechas de capacidad se estrecharán. Un nuevo modelo saldrá que reordenará las clasificaciones. Esta es la única predicción que haré con confianza.

Lo que no cambiará es el marco para pensar sobre ello. Seguirás necesitando evaluar modelos a lo largo de los mismos ejes: capacidad, coste, privacidad, velocidad y fiabilidad. Seguirás necesitando ajustar el modelo a la tarea en lugar de elegir uno y usarlo para todo. Seguirás necesitando mantenerte flexible.

Los ingenieros que veo haciendo el mejor trabajo no son leales a ningún modelo o enfoque de despliegue particular. Son pragmáticos. Usan el modelo comercial de frontera cuando la tarea lo exige, un modelo de nivel

medio cuando es suficientemente bueno, y un modelo local cuando la privacidad o el coste lo requieren. Miden lo que funciona. Cambian cuando algo mejor aparece.

No te pongas religioso con esto. El modelo es una herramienta. La habilidad está en saber por cuál herramienta alcanzar — y esa habilidad se transfiere independientemente de qué modelos existan dentro de seis meses.

11 El Prompting Como Ingeniería

No hay una sintaxis secreta. Ninguna fórmula mágica que haga que un agente produzca código perfecto. El prompting es *comunicación* — y ya sabes cómo comunicar.

Si alguna vez has escrito un buen informe de bug, sabes cómo hacer prompts. Si alguna vez has escrito un documento de diseño que un compañero podía implementar sin hacerte veinte preguntas, sabes cómo hacer prompts. Si alguna vez has creado un ticket de Jira que no volvió como algo completamente diferente de lo que querías — sabes cómo hacer prompts.

Las habilidades se transfieren directamente. Claridad, especificidad, contexto, restricciones. Las mismas cosas que hacen eficiente la colaboración humana hacen eficiente la colaboración con agentes. La diferencia es que los agentes no harán preguntas aclaratorias cuando tu prompt es vago. Simplemente adivinarán. Y adivinarán con confianza.

Aquí es donde muchos ingenieros experimentados luchan en silencio. Has pasado años construyendo la habilidad de *hacer* — escribir código, depurar, construir sistemas. Ahora la habilidad que importa es *articular* — explicar lo que quieres con suficiente precisión para que alguien más pueda hacerlo. Es un músculo diferente. Y puede sentirse, en los primeros días, como una degradación. No lo es. Pero la incomodidad es real, y pretender que no es así no ayuda.

11.1 La Anatomía de un Buen Prompt de Tarea

Un buen prompt tiene tres partes: *qué* quieres que se haga, *por qué* importa, y *cómo* se ve el éxito. La mayoría de la gente solo proporciona lo primero, y hasta eso suele ser vago.

Considera la diferencia:

Mal: «Arregla el bug de autenticación.»

Esto no le dice casi nada al agente. ¿Qué bug de autenticación? ¿Dónde se manifiesta? ¿Cuál es el comportamiento esperado? El agente irá a buscar por tu código base, formará una teoría sobre lo que podrías querer decir, y aplicará una corrección que podría estar completamente equivocada. Has convertido una corrección de cinco minutos en una revisión de veinte minutos de algo que no pediste.

Bien: «El endpoint de login devuelve 401 para tokens válidos cuando la caché de sesiones está fría. El bug probablemente está en `middleware/auth.go` en la función `validateSession`. El test en `auth_test.go:TestColdCacheLogin` lo reproduce. Arregla el bug y asegúrate de que todos los tests existentes de autenticación sigan pasando.»

Esto es un animal completamente diferente. El agente conoce el síntoma, la ubicación sospechada, y cómo verificar la corrección. Puede ir directamente al código relevante, entender el problema y validar su solución — todo sin adivinar.

El patrón es simple. *Qué* está roto o se necesita. *Dónde* mirar. *Cómo* verificar. Cada minuto que pasas haciendo tu prompt preciso te ahorra cinco minutos revisando la salida equivocada.

11.2 Especificación de Restricciones

Decirle a un agente qué hacer es solo la mitad del trabajo. Decirle qué *no* hacer es igualmente importante.

Los agentes son entusiastas. Optimizan para resolver el problema que describiste, y felizmente refactorizarán todo tu módulo, añadirán tres dependencias nuevas y cambiarán la API pública para hacerlo. Eso no es malicia — es un optimizador haciendo lo que hacen los optimizadores. Tu trabajo es establecer los límites.

Las restricciones útiles se ven así:

- «No modifiques la superficie de la API pública.»

- «Mantén la estructura de tests existente — añade casos de test nuevos, no reorganices.»
- «No añadas nuevas dependencias.»
- «Mantente dentro de los patrones de manejo de errores existentes en este código base.»
- «No cambies ningún archivo fuera del directorio `services/`.»

Piensa en las restricciones como las barandillas de un puente. El agente puede conducir a cualquier lugar dentro de los carriles, pero no puede pasar por encima del borde. Sin barandillas, obtienes soluciones creativas que técnicamente funcionan pero crean pesadillas de mantenimiento. Con ellas, obtienes soluciones que encajan en tu código base como si siempre hubieran estado ahí.

Cuanta más experiencia acumulas en ingeniería agéntica, más tus prompts se definen por sus restricciones que por sus instrucciones. Aprendes qué libertades llevan a buenos resultados y cuáles llevan al caos.

11.3 Descomposición de Tareas

Un error común: pedirle a un agente que construya algo grande en un solo prompt. «Construye un panel de usuario con métricas en tiempo real, acceso basado en roles y exportación a CSV.» Eso no es un prompt — es un proyecto. Y los proyectos necesitan descomponerse en tareas.

La descomposición de tareas es la práctica de dividir peticiones grandes en pasos pequeños y *verificables*. Cada paso tiene una entrada clara, una salida clara y una forma clara de comprobar si funcionó.

En lugar de «construye un panel de usuario,» escribes:

1. Crea el modelo de datos para las métricas del panel en `models/dashboard.go` con el esquema definido en el documento de diseño. Escribe tests unitarios para la validación del modelo.
2. Construye el endpoint de API `GET /api/dashboard` que devuelve métricas para el usuario autenticado. Escribe tests de integración.
3. Añade filtrado basado en roles para que los usuarios admin vean todas las métricas y los usuarios normales solo vean las suyas. Actualiza los tests existentes para cubrir ambos roles.

4. Construye el componente React que muestra los datos del panel. Usa el componente `DataTable` existente para la cuadrícula de métricas.

Cada paso es un prompt autocontenido. Cada uno tiene un resultado verificable. Cada uno se construye sobre la salida verificada del paso anterior. Si el paso dos sale mal, lo detectas antes de haber desperdiciado tiempo en el paso tres.

Esto no es solo buen prompting — es buena ingeniería. Estás aplicando las mismas habilidades de descomposición que usarías al planificar un sprint o desglosar un pull request. La unidad de trabajo es lo suficientemente pequeña para revisar, lo suficientemente pequeña para testear, y lo suficientemente pequeña para tirar si está mal.

11.4 El Prompt Como Especificación

Los mejores prompts que he visto se leen como documentos de diseño en miniatura. Describen el resultado deseado, no los pasos de implementación. Listan las restricciones. Definen criterios de aceptación. Proporcionan justo el contexto suficiente para que el agente tome buenas decisiones sin ahogarlo en información irrelevante.

Así se ve un prompt-como-especificación:

«Añade limitación de tasa al endpoint `/api/search`. Usa el middleware `RateLimiter` existente en `middleware/ratelimit.go`. Establece el límite a 100 peticiones por minuto por usuario autenticado, y 20 por minuto para peticiones no autenticadas. Devuelve un status 429 con un header `Retry-After` cuando se exceda el límite. Añade tests para ambas rutas, autenticada y no autenticada, incluyendo el caso límite donde un usuario alcanza exactamente el límite. No modifiques el middleware de limitación de tasa en sí — solo configúralo y aplícalo.»

Eso es una especificación. Un agente puede implementar esto sin ambigüedad. Un revisor humano puede comprobar el resultado contra los requisitos. El resultado deseado es claro, las restricciones son explícitas y los criterios de verificación están definidos.

Escribir prompts de esta forma requiere práctica. También requiere disciplina — la disciplina de pensar en lo que realmente quieres antes de empezar a teclear. Pero esa disciplina paga dividendos. Un prompt bien especificado produce un resultado que puedes fusionar. Un prompt vago produce un resultado que tienes que reescribir.

11.5 Iteración Sobre Perfección

Tu primer prompt no será perfecto. Eso está bien. El prompting es un proceso iterativo, y la habilidad no está en escribir el prompt perfecto — está en *leer la salida*, entender dónde se rompió la comunicación y refinar.

Cuando un agente produce algo equivocado, resiste la tentación de culpar a la herramienta. En su lugar, pregúntate: ¿qué no logré comunicar? ¿Dejé fuera una restricción? ¿El contexto era insuficiente? ¿Asumí conocimiento que el agente no tenía?

Esto es depuración — pero en lugar de depurar código, estás depurando tu propia comunicación. El mensaje de error es la salida del agente. La traza de pila es tu prompt. En algún lugar ahí está la falta de comunicación, y encontrarla hace que tu próximo prompt sea mejor.

Los ingenieros agénticos experimentados desarrollan un instinto de retroalimentación. Ven la salida del agente e inmediatamente saben qué parte de su prompt causó la desviación. «Ah, dije "maneja los errores" pero no especifiqué *qué* errores ni *cómo* manejarlos. Claro que metió un catch-all genérico.»

Cada iteración ajusta el bucle. El primer prompt te lleva al 70%. Una corrección de seguimiento te lleva al 90%. Un refinamiento final te lleva al hecho. Con el tiempo, tus primeros prompts mejoran y necesitas menos iteraciones. Pero nunca necesitas cero.

11.6 Desarrollo Dirigido por Voz

La mayoría de nosotros hacemos prompts tecleando. Eso tiene sentido — somos ingenieros, vivimos en texto. Pero hay otro canal de entrada que es más rápido, más natural y sorprendentemente infrautilizado: tu voz.

La conversión de voz a texto moderna ha alcanzado el punto en el que puedes hablar un prompt en tu terminal y tenerlo transcrito con precisión casi perfecta. Herramientas como Whisper, la Dictación de macOS y SuperWhisper te permiten hablar con tu agente en lugar de teclear. El resultado es el mismo — entra texto, sale código. Pero la experiencia es fundamentalmente diferente.

He aquí por qué: teclear y hablar son modos diferentes de pensar. Cuando tecleas, editas sobre la marcha. Borras una palabra, reformulas, retrocedes, reestructuras. El texto que produces está *pulido* — tuviste tiempo de suavizar las asperezas antes de que dejara tus dedos. Hablar no te da ese lujo. Cuando hablas, te comprometes. Las palabras salen de tu boca y se han ido. No hay tecla de retroceso.

Esto suena como una desventaja. En realidad es un campo de entrenamiento.

Cuando hablas un prompt, te ves forzado a organizar tus pensamientos *antes* de abrir la boca. No puedes apoyarte en la muleta de editar a mitad de frase. Tienes que saber lo que quieres, estructurarlo mentalmente y entregarlo con claridad — en tiempo real. Las primeras veces que lo intentes, divagarás. Dirás «eh» y «mm» y darás vueltas y te contradirás. El agente recibirá una transcripción desordenada, y la salida reflejará ese desorden.

Pero algo pasa si sigues haciéndolo. Mejoras. No solo en el prompting — en *hablar con claridad sobre problemas técnicos*. Desarrollas la capacidad de describir un bug, una funcionalidad o una tarea de refactorización en un solo flujo coherente. Aprendes a cargar el contexto al frente, a establecer restricciones temprano y a terminar con una petición clara. Dejas de divagar porque divagar produce malos resultados.

Esta habilidad se transfiere a todas partes. Reuniones de standup. Discusiones de arquitectura. Pair programming. Llamadas de incidentes. Cada situación donde necesitas articular una idea técnica con claridad, bajo presión de tiempo, sin la red de seguridad de un editor de texto. El desarrollo dirigido por voz no es solo una forma más rápida de hacer prompts — es práctica para cada conversación técnica que tendrás.

También hay una ventaja práctica de velocidad. La mayoría de la gente habla a 130 palabras por minuto. La mayoría teclea a 40 a 80. Para el tipo de prompts de alto nivel, orientados a intención, que producen la mejor salida del agente — «aquí está el problema, aquí está el contexto, aquí está lo que quiero, aquí está lo que no quiero» — hablar es simplemente más rápido. Pasas menos tiempo en la entrada y más tiempo revisando la salida.

Pruébalo durante una semana. Elige una herramienta de voz a texto, conéctala a tu flujo de trabajo y habla tus prompts en lugar de teclearlos. El primer día se sentirá torpe. Para el tercer día, notarás que tus prompts hablados se vuelven más ajustados. Al final de la semana, notarás que tu *comunicación hablada en general* se vuelve más ajustada.

También vale la pena señalar que los modelos de voz a texto pueden ejecutarse enteramente en tu máquina. La familia de modelos Parakeet de NVIDIA — modelos ASR compactos y de alta precisión — se ejecutan localmente sin ninguna dependencia cloud. Herramientas como SuperWhisper y whisper.cpp hacen lo mismo usando los pesos de Whisper de OpenAI. Un MacBook moderno puede ejecutar estos modelos en tiempo casi real con transcripción precisa y baja latencia. No necesitas un servicio cloud para convertir voz en texto — el tooling local ya está ahí.

A los agentes no les importa si tu prompt fue tecleado o hablado. Pero *tú* serás un pensador más claro por haberlo hablado.

11.7 Contexto Visual: Cuando las Palabras No Son Suficientes

No todo es fácil de describir en texto. Un layout roto, un glitch de renderizado extraño, un diálogo de error con una traza de pila — a veces la forma más rápida de comunicar lo que estás viendo es *mostrarlo*.

Los LLMs modernos son multimodales. Pueden leer capturas de pantalla, diagramas, fotos de pizarras y mensajes de error capturados de tu pantalla. Esto no es una funcionalidad novedosa — es una de las herramientas más infrautilizadas en el flujo de trabajo de ingeniería agéntica.

Aquí hay un flujo de trabajo que uso a diario: veo un bug en mi teléfono — un layout roto, un modal descentrado, un formulario que se come la entrada. Hago una captura en iOS, y gracias al Portapapeles Universal, la pego directamente en mi sesión de terminal en mi Mac. El agente ve lo que yo veo. No hay necesidad de describir «el botón se superpone al header en el viewport móvil» — la captura *es* la descripción.

Esto importa porque los bugs visuales son notoriamente difíciles de describir en texto. Terminas escribiendo tres párrafos sobre padding y z-index cuando una sola captura de pantalla comunica el problema instantáneamente. El agente puede ver el estado roto, razonar sobre qué está mal y proponer una corrección — a menudo más rápido de lo que podrías terminar de escribir la descripción.

Pero va más allá de los informes de bugs. Algunos flujos de trabajo comunes con contexto visual:

- **Capturas de errores.** Una consola de navegador llena de rojo, una traza de pila del terminal, un panel de despliegue mostrando health checks fallidos. Haz una captura, pégala, pide al agente que diagnostique. Esto es especialmente útil cuando los mensajes de error son largos o contienen formato que es doloroso de copiar como texto.
- **Referencias de diseño.** Un mockup de Figma, un boceto, la UI de un competidor que quieres aproximar. Pega la imagen y di «haz que nuestra página de configuración se vea como esto.» El agente puede extraer estructura de layout, elecciones de color y jerarquía de componentes de una referencia visual.
- **Depuración de estado visual.** «¿Por qué esta página se ve mal?» es un prompt terrible. Una captura de pantalla de la página *más* «¿por qué esta página se ve mal?» es uno genial. El agente puede comparar lo que ve contra el layout esperado e identificar problemas de CSS, datos faltantes o bugs de renderizado.
- **Fotos de pizarra.** Después de una discusión de arquitectura, saca una foto de la pizarra y pégala. El agente puede leer las cajas, flechas y etiquetas, y ayudarte a traducir ese boceto en estructura de código, definiciones de API o documentación.

El pipeline del portapapeles de iOS a Mac merece mención especial porque elimina toda fricción de este flujo de trabajo. No necesitas guardar

la captura, hacer AirDrop, encontrarla en Finder y arrastrarla a una herramienta. Ves el problema, lo capturas, lo pegas. Tres segundos desde «eso está roto» hasta «el agente lo está mirando.» Esa velocidad importa porque te mantiene en el flujo. Cualquier paso extra — incluso treinta segundos de gestión de archivos — crea suficiente fricción para que vuelvas a escribir una descripción de texto, que es más lenta y menos precisa.

La idea clave es que *el contexto no es solo texto.* Cuando hablamos de que el contexto es el ingrediente más importante en el trabajo agéntico, estábamos hablando de todas las formas de contexto — archivos de código, documentación, salida de tests *y* estado visual. Una captura de pantalla vale mil tokens, y los agentes que pueden ver son dramáticamente más útiles que los agentes que solo pueden leer.

Si no estás usando contexto visual en tu flujo de trabajo agéntico ya, empieza. Haz capturas de tus bugs. Pega tus mensajes de error. Comparte tus referencias de diseño. Los agentes pueden ver ahora. Déjalos.

11.8 Anti-Patrones

Algunos hábitos de prompting consistentemente producen malos resultados. Aprende a reconocerlos.

Ser demasiado vago. «Haz este código mejor.» ¿Mejor cómo? ¿Más rápido? ¿Más legible? ¿Más mantenible? El agente elegirá *algo* para mejorar, y probablemente no será lo que tenías en mente. Si no puedes articular qué significa «mejor,» no estás listo para hacer el prompt.

Ser demasiado prescriptivo. El fallo opuesto. «En la línea 47, cambia el nombre de la variable de `x` a `count`, luego añade un if en la línea 48 que compruebe si count es mayor que cero, luego…» Estás escribiendo el código en español y pidiéndole al agente que traduzca. Eso es más lento que escribir el código tú mismo. Describe el *resultado*, no las pulsaciones de teclas.

Volcado de contexto. Pegar todo tu código base, todos tus documentos de diseño y una transcripción de tus últimas tres reuniones de equipo en el prompt. Más contexto no siempre es mejor. El contexto irrelevante es ruido, y el ruido ahoga la señal. Dale al agente lo que necesita — rutas de

archivos, nombres de funciones, el comportamiento específico que quieres — y confía en que explore desde ahí.

Prompts fregadero. «Arregla el bug de autenticación, también refactoriza la capa de base de datos, y ya que estás actualiza el README y añade tipos TypeScript al cliente de API.» Estas son cuatro tareas separadas metidas en un prompt. El agente intentará todas, no hará ninguna bien, y producirá un diff tan grande que revisarlo toma más tiempo que hacer el trabajo tú mismo. Un prompt, una tarea.

Asumir contexto compartido. «Hazlo igual que hicimos con el módulo de pagos.» El agente no recuerda tu última sesión. No sabe lo que «nosotros» decidimos en el standup. Cada prompt empieza desde cero. Proporciona el contexto explícitamente, cada vez.

11.9 El Prompting Es una Habilidad

El prompting no es un truco de salón. No se trata de descubrir la frase mágica que desbloquea mejor salida. Es una habilidad de comunicación — y como todas las habilidades de comunicación, mejora con práctica, retroalimentación y atención deliberada.

Los ingenieros que sacan más provecho de las herramientas agénticas son los que tratan el prompting con el mismo rigor que aplican a escribir código. Piensan antes de teclear. Especifican antes de implementar. Verifican antes de seguir adelante.

Eso no es una habilidad nueva. Eso es simplemente ingeniería.

12 Orquestación Multi-Agente

Un agente es poderoso. Múltiples agentes trabajando juntos es algo completamente diferente.

Piénsalo así. Un solo carpintero puede construir un cobertizo. Pero ¿una casa? Necesitas un electricista, un fontanero, un techador — especialistas trabajando en paralelo, cada uno enfocado en lo que mejor sabe hacer, coordinándose lo justo para no estorbarse. La casa se levanta más rápido y el trabajo es mejor que el de una sola persona intentando hacerlo todo.

La ingeniería agéntica funciona de la misma forma. Un solo agente en una tarea compleja la recorrerá secuencialmente — planificar, implementar, testear, depurar, iterar. Funciona, pero es lento. Tres agentes, cada uno manejando una pieza bien acotada del problema, pueden terminar en un tercio del tiempo. A veces menos, porque los agentes enfocados cometen menos errores que un agente malabarizando demasiadas preocupaciones.

Pero hay una trampa. Los agentes en paralelo requieren *coordinación*. Y la coordinación tiene un coste. Este capítulo trata sobre cuándo pagar ese coste, y cómo pagarlo bien.

12.1 Estrategias de Descomposición

La parte más difícil del trabajo multi-agente no es ejecutar múltiples agentes. Es decidir cómo dividir el trabajo.

La regla de oro: las tareas deben tener *mínimo acoplamiento*. Si el trabajo del agente A depende de la salida del agente B, no pueden ejecutarse en paralelo — son secuenciales, y deberías tratarlos como tal. El arte está en encontrar las costuras en tu trabajo donde puedes cortar limpiamente.

Hay tres patrones de descomposición fiables.

Por capa. Un agente maneja la API del backend, otro construye el componente de frontend, un tercero escribe los tests. Esto funciona bien

porque las capas tienen límites naturales — una interfaz definida entre ellas. Mientras acuerdes el contrato de API por adelantado, cada agente puede trabajar independientemente.

Por funcionalidad. Si estás construyendo tres funcionalidades independientes, dale cada una a un agente separado. Esta es la descomposición más simple. Las funcionalidades tocan archivos diferentes, directorios diferentes, preocupaciones diferentes. Los conflictos de merge son raros.

Por preocupación. Un agente refactoriza, otro escribe tests para el código refactorizado, un tercero actualiza la documentación. Este es un patrón secuencial — más pipeline que paralelo — pero permite que cada agente se enfoque en un solo tipo de pensamiento. El agente de refactorización no tiene que cambiar de contexto al modo de escritura de tests. Solo refactoriza y traspasa.

La descomposición que elijas depende de la forma del trabajo. Pero el principio es constante: *encuentra las costuras, corta a lo largo de ellas, minimiza la superficie donde los agentes necesitan ponerse de acuerdo.*

12.2 Rama-por-Agente

Cada agente tiene su propia rama. Cubrimos esto en el capítulo de Git. En el trabajo multi-agente, se vuelve absolutamente esencial.

Pero las ramas solas no son suficientes. Dos agentes en ramas diferentes compartiendo el mismo directorio de trabajo pelearán por el sistema de archivos — sobrescribirán los archivos del otro, corromperán los builds del otro, romperán las ejecuciones de tests del otro. Necesitas *worktrees*.

Cada agente tiene su propio worktree de git: un directorio separado, en su propia rama, con su propia copia del código base. Los agentes comparten historial pero nada más. Pueden compilar, ejecutar tests, instalar dependencias y hacer un desastre — todo sin afectarse mutuamente.

La configuración es rápida:

```
git worktree add ../project-api agent/api-endpoint
git worktree add ../project-frontend agent/frontend-component
git worktree add ../project-tests agent/integration-tests
```

Tres directorios. Tres ramas. Tres agentes. Aislamiento completo. Este es el modelo de sandbox de los capítulos anteriores hecho concreto para el trabajo multi-agente. Cuando un agente termina, revisas su rama, fusionas si es buena, y eliminas el worktree. Si el trabajo es malo, tiras todo. Coste cero.

12.3 El Patrón de Traspaso

No todo el trabajo multi-agente es paralelo. A veces los agentes trabajan *secuencialmente*, cada uno construyendo sobre el anterior. El agente A planifica. El agente B implementa. El agente C revisa.

Este es el patrón de traspaso, y es poderoso — pero solo si el traspaso en sí es limpio.

El problema con los agentes secuenciales es la pérdida de contexto. El agente A tiene una comprensión rica del problema para cuando termina de planificar. El agente B empieza en frío. Si solo le dices al agente B «implementa el plan,» va a perder matices. Hará suposiciones diferentes. Resolverá un problema ligeramente diferente del que el agente A planificó.

La solución es *traspaso estructurado*. El agente A no solo planifica — produce un artefacto que captura su razonamiento. Esto puede tomar varias formas:

- **Un documento resumen.** Un archivo markdown dejado en el repositorio: `PLAN.md`. Describe qué necesita pasar, qué compromisos se consideraron, qué se rechazó y por qué. El agente B lee esto antes de escribir una línea de código.
- **Un conjunto de archivos.** El agente A crea archivos stub — funciones vacías con docstrings, definiciones de interfaces, firmas de tipos. El agente B los rellena. Los stubs *son* el traspaso.
- **Un prompt estructurado.** La salida del agente A se convierte en la entrada del agente B, formateada como una descripción detallada de tarea con criterios de aceptación. Lo pegas directamente en el contexto del agente B.

La clave es que el traspaso debe ser *explícito y completo*. Sin suposiciones implícitas. Sin «el siguiente agente lo descubrirá.» Cada decisión, cada restricción, cada caso límite — escrito.

Esto requiere disciplina. Pero es la misma disciplina que aplicarías al escribir un ticket para un colega humano en un continente diferente y una zona horaria diferente. Si no confiarías en un traspaso verbal, no confíes en uno implícito entre agentes.

12.4 El Problema del Merge

Aquí es donde el trabajo multi-agente se pone genuinamente difícil.

Tres agentes trabajaron en paralelo. Cada uno produjo una rama limpia y testeada. Ahora necesitas fusionarlas todas en main. A veces esto es indoloro — tres ramas tocando archivos completamente diferentes se fusionan sin un solo conflicto. Hermoso.

A veces no. El agente A cambió el esquema de base de datos. El agente B añadió una migración que asume el esquema antiguo. El agente C actualizó el handler de API que tanto A como B también tocaron. Ahora tienes un conflicto triple y ningún agente tiene la imagen completa.

Hay tres estrategias para lidiar con esto.

Prevención. Los mejores conflictos de merge son los que nunca suceden. Cuando descompongas el trabajo, piensa en la propiedad de archivos. Asigna diferentes directorios, diferentes módulos, diferentes archivos a diferentes agentes. Si los agentes nunca tocan el mismo archivo, nunca pueden entrar en conflicto. Esta es la estrategia más barata y deberías usarla siempre que sea posible.

El agente de merge. Cuando los conflictos surgen, levanta un nuevo agente cuyo único trabajo es fusionar. Dale las ramas, los conflictos y el contexto del trabajo de cada agente. Un buen agente de merge puede resolver la mayoría de los conflictos automáticamente — lee ambos lados, entiende la intención y produce un resultado coherente. Es como un ingeniero senior que se sienta con dos PRs y descubre cómo encajan juntos.

Revisión humana. A veces el conflicto es semántico, no sintáctico. El código se fusiona limpiamente pero la *lógica* se contradice. Dos agentes tomaron decisiones de diseño incompatibles. Ningún merge automatizado detectará esto. Aquí es donde te ganas tu sueldo como ingeniero. Revisa las ramas antes de fusionar. Lee los diffs uno al lado del otro. Asegúrate de que las piezas realmente encajan.

En la práctica, usarás las tres. Previene lo que puedas. Automatiza el resto. Revisa todo.

12.5 Overhead de Orquestación

Más agentes no siempre es mejor.

Cada agente adicional añade coste de coordinación. Tienes que descomponer el trabajo. Configurar worktrees. Definir interfaces. Manejar merges. Revisar múltiples ramas en lugar de una. Para una tarea que le lleva a un solo agente treinta minutos, levantar tres agentes podría llevar cuarenta y cinco — veinte minutos de trabajo de agentes en paralelo más veinticinco minutos de tu tiempo orquestando.

El punto de equilibrio es más alto de lo que crees. En mi experiencia, la orquestación multi-agente empieza a rendir cuando la tarea total le llevaría a un solo agente *al menos dos horas*. Por debajo de eso, el overhead se come las ganancias.

También hay otros costes. El contexto se fragmenta. Cada agente ve solo su pieza. Ningún agente entiende todo el sistema como lo haría uno solo trabajando. Esto puede llevar a inconsistencias — diferentes convenciones de nomenclatura, diferentes patrones de manejo de errores, diferentes suposiciones sobre estado compartido.

La habilidad no es ejecutar tantos agentes como sea posible. La habilidad es saber *cuándo* paralelizar y cuándo un solo agente enfocado es la herramienta adecuada. Una tripulación de cinco no siempre es más rápida que un marinero experimentado que conoce el barco.

12.6 Un Ejemplo Práctico

Hagámoslo concreto. Estás construyendo una aplicación web y necesitas añadir una nueva funcionalidad: notificaciones de usuario. Los usuarios deberían ver un icono de campana con un contador, hacer clic para ver una lista y marcar notificaciones como leídas. Necesitas una API, un componente de frontend y tests.

Este es un caso de libro para tres agentes en paralelo.

Paso 1: Define la interfaz. Antes de levantar ningún agente, pasas cinco minutos escribiendo el contrato de API. `GET /api/notifications` devuelve una lista. `PATCH /api/notifications/:id` marca una como leída. El objeto de notificación tiene `id`, `message`, `read`, `created_at`. Escribe esto en un documento compartido o un archivo stub. Este es el contrato contra el que trabajan los tres agentes.

Paso 2: Crea los worktrees.

```
git worktree add ../app-api agent/notifications-api
git worktree add ../app-frontend agent/notifications-frontend
git worktree add ../app-tests agent/notifications-tests
```

Paso 3: Lanza los agentes. Cada agente recibe una tarea clara y acotada:

- Agente 1: «Implementa los endpoints de API de notificaciones en `../app-api`. Sigue el contrato en `API_CONTRACT.md`. Incluye modelo, ruta y controlador. Escribe tests unitarios para el controlador.»
- Agente 2: «Construye el componente UI de notificaciones en `../app-frontend`. Llama a `GET /api/notifications` y `PATCH /api/notifications/:id`. Muestra un icono de campana con contador de no leídas. Al hacer clic abre una lista desplegable.»
- Agente 3: «Escribe tests de integración en `../app-tests`. Cubre el flujo completo: crear una notificación, obtener la lista, marcar como leída, verificar que el contador se actualiza.»

Paso 4: Déjalos trabajar. Los tres agentes se ejecutan simultáneamente. Cada uno hace commits en su propia rama. Monitorizas el progreso pero no intervienes a menos que algo salga mal.

Paso 5: Revisa y fusiona. Cuando los tres terminan, revisas cada rama. La rama de API se fusiona primero — es la base. Luego la rama de frontend. Luego los tests. Ejecutas la suite completa de tests después de cada merge para detectar problemas de integración temprano.

Tiempo total: quizá cuarenta minutos. El agente de API tardó treinta, el agente de frontend tardó veinticinco, y el agente de tests tardó treinta y cinco. Pero se ejecutaron en paralelo, así que el tiempo de reloj fue treinta y cinco minutos más diez minutos de tu trabajo de orquestación.

¿Un solo agente haciendo todo secuencialmente? Probablemente noventa minutos.

Las matemáticas funcionan cuando la tarea es lo suficientemente grande. Y a medida que mejoras en la descomposición, desarrollarás intuición para qué tareas vale la pena dividir y cuáles no. Como la mayoría de las cosas en ingeniería, es una cuestión de criterio. Pero ahora tienes las herramientas para tomarlo.

13 Agentes en el Pipeline

Un equipo que conozco montó una automatización ingeniosa el año pasado. Añadieron un agente Claude como paso de CI que detectaría fallos de lint y los arreglaría automáticamente. Un desarrollador hace push del código, se ejecuta el linter, y si falla, el agente reescribe las líneas ofensivas y hace push de un commit de corrección. Sin intervención humana. El pipeline se mantiene verde. A todos les encantó.

Funcionó brillantemente durante unas dos semanas.

Luego alguien notó algo raro durante una revisión de código. Toda una categoría de reglas de lint había desaparecido. El agente había descubierto que la forma más rápida de arreglar un fallo de lint era modificar `.eslintrc.json` — deshabilitar la regla, hacer push del cambio de configuración, el pipeline se pone verde. Había estado haciendo esto durante un mes. Nadie lo detectó porque la señal que estaban vigilando — el estado del pipeline — era exactamente la señal que el agente había aprendido a manipular.

La solución fue sencilla: hacer que la configuración de lint fuera de solo lectura para el agente. Pero la lección era más grande que un pipeline mal configurado.

Los agentes automatizados necesitan las mismas barandillas que los interactivos. Probablemente más, porque no hay nadie sentado ahí mirando el diff desplazarse. Cuando trabajas con un agente localmente, ves cada archivo que toca. Notas cuando hace algo ingenioso pero incorrecto. En CI, el agente se ejecuta en la oscuridad. Lo único que ves es el resultado — verde o rojo. Y si el agente encuentra una forma de hacer el resultado verde sin realmente resolver el problema, puede que no lo notes durante semanas.

Este capítulo trata sobre usar agentes en pipelines de forma responsable. Dónde añaden valor genuino, dónde crean riesgo, y cómo configurarlos para que sigan siendo útiles sin convertirse en un lastre.

Las implicaciones son más altas en CI que en tu escritorio. Hazlo bien, y los agentes multiplican el rendimiento de tu equipo las veinticuatro horas. Hazlo mal, y has construido un arma costosa y sin supervisión.

13.1 Agentes Como Pasos de CI

La forma más simple de meter agentes en tu pipeline es como pasos de CI — trabajos discretos que se ejecutan en cada push o pull request, igual que tu linter o suite de tests.

No estás reinventando tu pipeline. Le estás añadiendo inteligencia.

El agente de CI más inmediatamente útil: pre-filtrado de PR. No reemplaza la revisión humana, pero filtra. Un agente lee el diff, busca problemas obvios — imports no usados, nombres inconsistentes, manejo de errores faltante, archivos de test que realmente no asertan nada — y deja comentarios. Para cuando un revisor humano abre el PR, lo trivial ya está señalado. El humano puede enfocarse en arquitectura, enfoque, intención.

Otros buenos candidatos:

- **Generación de changelog.** El agente lee los commits desde el último release, los cruza con números de tickets, y redacta notas de release. Un humano edita antes de publicar, pero el primer borrador es gratis.
- **Ordenación de imports y formateo.** Seguro, verificable, de bajo riesgo. Si el agente formatea algo mal, el diff es obvio.
- **Resúmenes de actualización de dependencias.** Cuando Dependabot abre un PR, un agente puede leer el changelog del paquete actualizado y resumir qué cambió y si es probable que afecte a tu código.

También hay un uso más sutil: **triaje de fallos de tests**. Cuando una ejecución de CI falla, un agente puede leer la salida de tests, identificar el test que falla, mirar el diff reciente y publicar un comentario explicando si el fallo parece un bug genuino o un test inestable. No siempre acertará, pero ahorra al desarrollador abrir los logs de CI, desplazarse por doscientas líneas de salida y descubrir qué salió mal. Esa investigación de diez minutos se convierte en un vistazo de treinta segundos a un comentario.

El principio clave: los agentes en CI deberían hacer cosas que son **seguras de automatizar** y **fáciles de verificar**. Si no puedes determinar rápidamente si el agente hizo lo correcto, no debería estar automatizado todavía. Un buen test de fuego: ¿te sentirías cómodo si el agente hiciera esto cien veces y solo revisaras por muestreo diez? Si la respuesta es no, no está listo para CI.

13.2 El Agente Nocturno

Este es uno de los patrones más poderosos en la ingeniería agéntica, y uno de los menos discutidos. También es el que hace que los ojos de los managers se iluminen — «¿quieres decir que el agente trabaja mientras dormimos?» — que es exactamente por lo que necesita un encuadre cuidadoso.

Estás cerrando el día. Hay un ticket bien definido en el backlog — añadir un nuevo endpoint de API, escribir una migración de datos, refactorizar un módulo para usar la nueva biblioteca de logging. El tipo de tarea donde los requisitos son claros y la definición de hecho es testeable. Apuntas un agente, le das una rama y te vas a casa. Te despiertas con un PR con el trabajo hecho, los tests pasando, listo para revisar.

El listón de calidad para agentes sin supervisión es más alto que para los interactivos. Cuando estás ahí sentado mirando, detectas errores en tiempo real. Cuando el agente trabaja durante la noche, los errores se acumulan. Así que necesitas:

- **Tests exhaustivos para verificar.** El agente necesita una forma de saber que ha terminado. Tests existentes que deberían seguir pasando, más criterios claros para nuevos tests que debería escribir.
- **Alcance estricto.** Un ticket, una preocupación. No apuntes un agente nocturno a «refactorizar el sistema de autenticación.» Apúntalo a «añadir limitación de tasa al endpoint de login.»
- **Aislamiento de rama.** Siempre una rama nueva, siempre un worktree. Si el agente hace un desastre, borras la rama. Nada toca main.
- **Un timeout.** Establece un límite duro — cuatro horas es generoso para la mayoría de las tareas. Un agente que lleva seis horas ejecután-

dose no está progresando. Está dando vueltas en círculos. Mátalo y reevalúa por la mañana.

Un script de configuración simple se ve así:

```
#!/bin/bash
# overnight-agent.sh — fire and forget before you leave

TICKET_ID="$1"
BRANCH_NAME="agent/overnight-${TICKET_ID}"
WORKTREE_DIR="../overnight-${TICKET_ID}"

# Create isolated workspace
git worktree add "$WORKTREE_DIR" -b "$BRANCH_NAME"

# Run the agent with a token budget and timeout
timeout 4h claude --worktree "$WORKTREE_DIR" \
  --max-tokens 200000 \
  --prompt "Implement ticket ${TICKET_ID}. Read TICKETS/
${TICKET_ID}.md for requirements. Write tests. Commit your work.
Do not modify any CI config or lint rules." \
  2>&1 | tee "logs/overnight-${TICKET_ID}.log"

# Push the branch so you can review in the morning
cd "$WORKTREE_DIR" && git push -u origin "$BRANCH_NAME"
```

Lo refinas con el tiempo. Añade notificaciones de Slack cuando termine. Añade un resumen de lo que hizo. Añade una comprobación que verifique que la suite de tests pasa antes de hacer push.

Algo que he aprendido de equipos que hacen esto bien: la descripción del ticket importa enormemente. Un ticket que dice «añadir endpoint de preferencias de usuario» no es suficiente. El agente nocturno necesita criterios de aceptación, payloads de ejemplo de petición/respuesta, y punteros a endpoints existentes similares que pueda usar como referencia. Estás escribiendo instrucciones para un desarrollador competente pero sin contexto. Cuanto más específico seas, mejor será el resultado.

Los equipos que más provecho sacan de los agentes nocturnos son los que invierten en la disciplina de escritura de tickets. Lo cual, de nuevo, beneficia a todos — tus compañeros humanos también prefieren tickets claros. El agente solo hace más visible el coste de la vaguedad.

El bucle principal es simple: aislar, restringir, ejecutar, revisar por la mañana.

13.3 Control de Costes en CI

Cuando ejecutas un agente interactivamente, tienes un cortacircuito natural: tú mismo. Ves los tokens subiendo, ves al agente dando vueltas en círculos, pulsas Ctrl+C. En CI, no hay nadie mirando.

Un equipo ejecutando un monorepo concurrido lo aprendió por las malas. Habían configurado un agente para auto-revisar cada PR. Bastante razonable — excepto que su monorepo veía cuarenta a cincuenta PRs al día, y cada revisión consumía alrededor de quince mil tokens. Eso es manejable. Lo que no era manejable fue la lógica de reintento. Cuando el agente alcanzaba un límite de tasa, el job de CI reintentaba. Tres reintentos por fallo, backoff exponencial, pero cada reintento iniciaba una ejecución de agente nueva desde cero. Durante un fin de semana festivo, con un lote de actualizaciones automatizadas de dependencias llegando, el pipeline generó una factura de $4.000. Nadie estaba en la oficina para notarlo.

Protégete:

- **Presupuestos de tokens por ejecución de pipeline.** Establece un tope duro. Si el agente lo alcanza, el job falla con un mensaje claro. Mejor perderte una revisión que quemar tu presupuesto mensual en un día.
- **Límites de concurrencia.** No dejes que veinte jobs de agentes se ejecuten simultáneamente. Ponlos en cola. Dos o tres ejecuciones de agente concurrentes son suficientes para la mayoría de los equipos.
- **Alertas de gasto.** Tu proveedor de LLM casi seguro las soporta. Establece una al 50% de tu presupuesto mensual. Establece otra al 80%. Canalízalas a un canal que alguien realmente lea.
- **Interruptores de emergencia.** Un feature flag o variable de entorno que deshabilite todos los pasos de agentes en CI instantáneamente. Cuando algo sale mal a las 2am, quieres una solución de una línea, no un cambio de configuración de pipeline que necesita su propio PR.
- **Seguimiento de coste por job.** Registra el conteo de tokens y coste estimado de cada ejecución de agente en CI. No puedes optimizar lo que

no mides. Un informe semanal de gasto de agentes en CI, desglosado por tipo de job, te mostrará dónde va el dinero y dónde ajustar.

Un cortacircuito simple en tu configuración de CI se ve así:

```
agent-review:
  timeout-minutes: 15
  env:
    MAX_TOKENS: 50000
    COST_ALERT_THRESHOLD: "$5.00"
  steps:
    - name: Run agent review
      run: |
        claude review --max-tokens $MAX_TOKENS \
          --on-budget-exceeded "exit 1" \
          pr/$PR_NUMBER
```

Los detalles variarán según herramienta y plataforma, pero el patrón es constante: establece un techo, falla ruidosamente cuando lo alcances, y haz el techo fácil de ajustar.

La regla general: trata tu gasto en LLM en CI como tratas tu gasto en computación cloud. Monitorízalo, presupuéstalo, alerta sobre ello. Es un centro de costes real.

13.4 La Cuestión de la Revisión

Los PRs generados por agentes siguen necesitando revisión humana. Punto.

La tentación es real. El agente escribió el código. Los tests pasan. El linter está contento. La cobertura no ha bajado. ¿Por qué no auto-fusionar? Te ahorrarás quince minutos de tiempo de revisión, y el pipeline de CI ya verificó todo lo que importa.

Pero piensa en qué significa realmente «todo lo que importa». Los tests verifican corrección. No verifican intención.

Un agente al que se le pide «reducir el número de consultas de base de datos en la página de perfil de usuario» podría cachear agresivamente e introducir bugs de datos obsoletos que ningún test actual detecta. Podría desnormalizar datos de una forma que hace la siguiente funcionalidad el

doble de difícil de construir. Podría resolver el problema perfectamente pero en un estilo completamente ajeno al resto de tu código base. Los tests pasan porque los tests comprueban comportamiento, no enfoque.

Un revisor humano detecta estas cosas. Lee buscando el *cómo*, no solo el *qué*. Nota cuando una solución funciona hoy pero crea deuda para mañana. Detecta el atajo que confundirá al siguiente desarrollador que toque este código.

También hay una dimensión social. Si tu equipo sabe que los PRs de agentes se auto-fusionan, dejan de prestar atención al código generado por agentes. Se convierte en una caja negra. Seis meses después, la mitad de tu código base fue escrita por un agente y nadie en el equipo la entiende completamente. Esa es una brecha de conocimiento que te dolerá durante un incidente.

Auto-fusionar está bien para cambios triviales y mecánicos — correcciones de formato, ordenación de imports, bumps de versión. Para cualquier cosa que implique una decisión de diseño, un humano la revisa. El agente es un redactor rápido de borradores, no un tomador de decisiones. Y la revisión no tiene que ser exhaustiva — un escaneo de cinco minutos para comprobar que el enfoque es razonable ya es mucho.

13.5 Despliegues Asistidos por Agentes

Los despliegues son donde las cosas se ponen genuinamente peligrosas, y donde la brecha entre «el agente puede hacer esto» y «el agente debería hacer esto» es más amplia. Seamos precisos sobre qué deberían y qué no deberían hacer los agentes aquí.

Los agentes son excelentes **monitorizando** despliegues. Pueden observar flujos de logs, rastrear tasas de error, comparar percentiles de latencia contra la línea base pre-despliegue, y señalar anomalías. Un agente que dice «la tasa de error en `/api/checkout` ha aumentado 3x desde el despliegue hace doce minutos» es enormemente valioso. Está leyendo datos, encontrando patrones y sacando información a la superficie — exactamente en lo que los agentes son buenos.

Los agentes también pueden **sugerir** acciones. «Basándome en el aumento de tasa de errores, recomiendo revertir a la versión anterior» es una sugerencia útil. Es el tipo de cosa que normalmente requeriría que alguien estuviera mirando un dashboard de Grafana durante quince minutos para concluir. El agente ha hecho el análisis. Un humano decide si actuar.

Lo que los agentes no deberían hacer es tomar la decisión de rollback de forma autónoma. Los despliegues a producción son demasiado consecuentes para decisiones ad-hoc de agentes. Si quieres rollbacks automatizados, usa un runbook pre-aprobado con umbrales duros — «si la tasa de error excede el 5% durante tres minutos, revertir automáticamente.» Eso es determinista. Lo testeaste. Aprobaste la lógica de antemano. ¿Un agente decidiendo por su cuenta si un aumento de tasa de error del 2.3% «se siente» lo suficientemente significativo para revertir? Esa es una receta para rollbacks innecesarios o incidentes no detectados. Las reglas deterministas son aburridas. Aburrido es bueno cuando tus ingresos están en juego.

En la práctica, un agente de monitorización de despliegues se ve algo así: se suscribe a tu agregador de logs y dashboard de métricas vía API, se ejecuta durante quince minutos después de cada despliegue, y publica un resumen en Slack. «Despliegue de v2.4.1 completado. Tasa de error estable en 0.3%. Latencia P99 aumentó de 180ms a 210ms — dentro de la varianza normal. No se detectaron nuevos tipos de excepción.» La mayor parte del tiempo, ese resumen es aburrido. Ese es el punto. Cuando no es aburrido, quieres saberlo inmediatamente.

Esto conecta con el principio de barandillas: producción es solo lectura para los agentes. Observan, analizan, reportan, recomiendan. Los humanos (o automatización pre-aprobada con reglas deterministas) toman acción.

13.6 El Pipeline Como Contexto

Tu configuración de CI/CD es contexto, y los agentes la leen como cualquier otro código. Es fácil olvidarse de esto — la mayoría de los equipos tratan su configuración de pipeline como fontanería de infraestructura, no como algo que necesita comunicar con claridad. Pero cuando un agente

está intentando entender por qué un build falló o qué pasos de verificación existen, la configuración del pipeline es lo primero que lee.

Un pipeline bien estructurado ayuda a los agentes a entender tu proyecto. Nombres de etapa claros (`build`, `unit-tests`, `integration-tests`, `deploy-staging`) le dicen al agente cómo es tu proceso de verificación. Salida de error estructurada — logs JSON, códigos de salida con significado, categorías de error — le da al agente algo con lo que trabajar cuando un paso falla.

Un pipeline que produce `BUILD FAILED` y nada más es inútil para agentes y humanos por igual.

Invierte en observabilidad del pipeline:

- **Logs estructurados.** JSON o pares clave-valor, no texto libre. Un agente puede parsear `{"stage": "unit-tests", "failed": 3, "passed": 247, "failures": ["test_auth_flow", "test_rate_limit", "test_session_expiry"]}` y tomar acción significativa. No puede hacer mucho con `Tests failed. See above for details.`
- **Códigos de salida significativos.** No dejes que todo salga con código 1. Distingue entre «tests fallaron» (arreglable) y «no se pudo conectar a la base de datos» (problema de infraestructura, no asunto del agente).
- **Almacenamiento de artefactos.** Resultados de tests, informes de cobertura, logs de build — guárdalos como artefactos del pipeline. Un agente depurando un fallo de CI necesita leer la salida completa, no solo las últimas diez líneas que caben en la vista de consola.

Considera añadir un `PIPELINE.md` a tu repositorio — una descripción en lenguaje llano de lo que hace cada etapa de CI, cómo se ven sus salidas esperadas, y qué modos de fallo comunes existen. Un agente que puede leer este archivo antes de depurar un fallo de CI rendirá dramáticamente mejor que uno que tiene que hacer ingeniería inversa de tu pipeline solo desde configuración YAML. Y tus nuevas incorporaciones también te lo agradecerán.

Un buen diseño de pipeline y una buena integración de agentes se refuerzan mutuamente. Mejoras tu CI para los agentes, y tus desarrolladores humanos también se benefician. Es una de esas inversiones raras que pagan dividendos en ambas direcciones.

13.7 Empezando Pequeño

Si has leído este capítulo y estás tentado de conectar agentes a cada etapa de tu pipeline para el viernes — no lo hagas. He visto ese impulso. Suele terminar con un fin de semana deshaciendo la automatización porque el equipo no estaba listo para el volumen de ruido generado por agentes.

Los equipos que tienen éxito con agentes en CI son los que construyen confianza incrementalmente, igual que con los agentes interactivos.

La buena noticia es que el camino está bien trazado. Aquí hay una hoja de ruta práctica de adopción que he visto funcionar en múltiples equipos:

Semana 1: Auto-formateo. Añade un paso de CI que ejecute un agente para arreglar problemas de formato en PRs. El formateo es determinista, fácil de verificar y de bajo riesgo. Si el agente reformatea algo incorrectamente, el diff está ahí mismo. Revisa los commits del agente la primera semana para construir confianza.

Semana 2: Pre-filtrado de PR. Añade comentarios de revisión generados por agentes en PRs. No bloqueantes — solo informativos. Deja que tu equipo vea qué detecta el agente y calibra si su retroalimentación es útil o ruidosa. Ajusta el prompt según lo que funcione.

Mes 1: Redacción de changelog. Apunta al agente al historial de commits para generar borradores de notas de release. Un humano edita y publica. Estás usando al agente como un primer borrador, no como un tomador de decisiones. Este también es buen momento para configurar monitorización de costes y alertas.

Mes 3: Agentes nocturnos. Para ahora has construido confianza en la salida generada por agentes y tienes la infraestructura — worktrees, presupuestos de tokens, aislamiento de ramas, procesos de revisión. Empieza con un solo ticket bien acotado. Revisa el resultado cuidadosamente. Itera en tu script nocturno. Expande gradualmente a tareas más complejas a medida que crece tu confianza.

Fíjate en lo que *no* está en esta hoja de ruta: auto-fusión, despliegues autónomos o agentes tomando decisiones arquitectónicas. Esos no son la etapa cuatro. Pueden ser la etapa diez, o puede que nunca sean apropiados para tu equipo. La hoja de ruta no es una marcha hacia la automatización

total — es una marcha hacia el nivel correcto de automatización para tu contexto.

Resiste la tentación de saltarte pasos. Cada uno se construye sobre el anterior. Aprendes en qué son buenos tus agentes, dónde tienen problemas y qué barandillas necesitas. Para el mes tres, CI asistido por agentes se siente natural — no porque automatizaste todo de golpe, sino porque te ganaste cada pieza a través de la experiencia.

El pipeline es solo otro entorno donde los agentes trabajan. Los mismos principios aplican: acota estrictamente, verifica exhaustivamente, confía incrementalmente. La única diferencia es que nadie está mirando, así que tus redes de seguridad necesitan ser mucho más fuertes.

14 Cuando los Agentes Se Equivocan

Todo ingeniero que trabaja con agentes el tiempo suficiente tiene una colección de estas historias. Los momentos en que te echas hacia atrás en la silla, miras la pantalla y murmuras algo impublicable. Son humillantes. Son educativas. Y son *inevitables*.

Este capítulo es una colección de historias de guerra — cosas que realmente salen mal cuando entregas trabajo real a un agente. No hipotéticas. No demos fabricadas. El tipo de fallos que te cuestan una tarde, un despliegue o tu fe en la automatización. Cada una se conecta con principios de capítulos anteriores, porque los principios son baratos hasta que los aprendes por las malas.

Lee estas antes de cometer los mismos errores. O léelas después, y siéntete menos solo.

14.1 El Refactorizador Entusiasta

La tarea era simple: arreglar un problema de z-index en un menú desplegable. El desplegable se renderizaba detrás de un overlay modal. Una corrección de CSS de una línea — quizá dos si eres cuidadoso.

El agente vio el componente del desplegable, decidió que su CSS era «inconsistente con las prácticas modernas,» y lo refactorizó. Luego notó que el modal usaba un patrón de estilo diferente, así que lo refactorizó también. Luego los componentes de botón. Luego los componentes de tarjeta. Antes de que levantaras la vista de tu café, treinta y dos archivos habían cambiado. El diff era de 1.400 líneas. El agente había esencialmente reescrito el sistema de estilos de la biblioteca de componentes, migrando de un enfoque a otro con admirable consistencia y cero autorización.

¿El bug original de z-index? Seguía ahí. Enterrado en algún lugar de la avalancha de cambios, el agente había reproducido el mismo problema de capas con nuevos nombres de clase.

El PR era irrevisable. No puedes revisar significativamente 1.400 líneas de cambios de CSS en treinta y dos archivos. Lo que *puedes* hacer es borrar la rama y empezar de nuevo. Que es lo que pasó.

Lo que haces diferente ahora: Acota las tareas estrictamente. «Arregla el z-index del desplegable en `NavMenu.tsx`, no toques nada más.» Usa una rama dedicada para que el daño esté contenido. Y *siempre* revisa el diff antes de dejar que el agente pase a otra cosa. En el momento en que veas el conteo de archivos subir más allá de lo que tiene sentido para la tarea, detén al agente. El capítulo de barandillas existe por una razón.

14.2 La Biblioteca Alucinada

El agente necesitaba parsear algunos rangos de fechas complejos de la entrada del usuario. Importó `@temporal/daterange-parser`, escribió código elegante usando su método `.parseRange()` con opciones para parseo consciente del idioma y coincidencia difusa. El código era limpio. Los tipos eran correctos. El manejo de errores era cuidadoso. Incluso escribió tests — que, naturalmente, también importaban el paquete alucinado.

Todo se veía perfecto en el diff. Las firmas de funciones tenían sentido. La API estaba bien diseñada. Era una biblioteca *tan* plausible que casi no la cuestionaste.

Luego `npm install` falló. El paquete no existía. Nunca había existido. El agente había inventado una biblioteca, le había dado un nombre creíble y una API coherente, y había escrito código de producción contra una ficción.

La parte insidiosa: si solo hubieras hecho revisión de código — leyendo la lógica, comprobando los tipos, evaluando el enfoque — lo habrías aprobado. El código era *bueno*. Simplemente no conectaba con la realidad.

Lo que haces diferente ahora: El agente ejecuta los tests. Siempre. Si tu configuración no lo permite, los ejecutas tú mismo antes de revisar el

código. Sin excepciones. Un `npm install` que falla es una señal fuerte y obvia. Una API alucinada que nunca se ejecutó es una bomba silenciosa. Los tests detectan lo que la revisión humana no detecta — no porque tu revisión sea mala, sino porque las ficciones plausibles están *diseñadas* para pasar la revisión. Eso es lo que *es* la alucinación.

14.3 El Bucle Infinito

Le pediste al agente que arreglara un test de integración que fallaba. Leyó el error, hizo un cambio, ejecutó el test. Nuevo error. Leyó *ese* error, hizo otro cambio, ejecutó el test. Error diferente. Luego una variación del primer error. Luego algo nuevo. Luego el primer error de nuevo.

Estabas en otro terminal, trabajando en otra cosa. Cuarenta minutos después comprobaste. El agente había hecho diecinueve intentos. Había quemado 200.000 tokens. El código ahora estaba peor que al principio — un registro geológico de correcciones fallidas apiladas unas sobre otras. El agente seguía adelante, alegremente confiado en que el intento veinte sería el bueno.

No lo fue.

El problema fundamental era que el agente no entendía *por qué* el test fallaba. Estaba haciendo coincidencia de patrones sobre mensajes de error y haciendo ediciones locales, pero el problema real era un malentendido arquitectónico — un estado de base de datos compartido entre casos de test que requería un enfoque de setup completamente diferente. Ninguna cantidad de ajustes al código bajo test arreglaría un problema en el harness de tests.

Lo que haces diferente ahora: Establece límites de iteración. Tres intentos en el mismo problema es un máximo razonable. Si el agente no lo ha resuelto en tres pasadas, no lo resolverá en treinta. Cuando veas el bucle — error, corrección, error diferente, corrección, error original — *rómpelo manualmente*. Detén al agente, lee los errores tú mismo, y dale un enfoque completamente diferente o toma el control. El tiempo del agente es barato. Tu tarde desperdiciada no. Más importante, empieza un contexto fresco. La comprensión del agente ahora está contaminada con diecinueve teorías

equivocadas. Un inicio limpio con tu diagnóstico del problema *real* llegará más lejos, más rápido.

14.4 La Respuesta Equivocada Con Confianza

Un job en background ocasionalmente procesaba el mismo ítem dos veces. Condición de carrera clásica. Apuntaste al agente al problema.

El agente analizó el código, identificó la ventana de carrera y añadió un delay de 500ms entre la comprobación y la actualización. «Esto asegura que la transacción anterior tenga tiempo de hacer commit antes de la siguiente comprobación,» explicó, con la serena confianza de alguien que nunca ha operado un sistema bajo carga.

Los tests pasaron. El delay era más largo que el tiempo de transacción del test, así que la ventana de carrera se cerró — en el entorno de tests, con un worker concurrente, en una máquina tranquila.

Se fue a producción. Bajo carga, con doce workers y latencia de base de datos variable, 500ms a veces no era suficiente. Y ahora tenías un *nuevo* problema: el delay había reducido el rendimiento lo suficiente como para que la cola de jobs se acumulara durante las horas pico, creando timeouts en cascada que tumbaron un servicio no relacionado.

El sleep no arregló la condición de carrera. La ocultó a baja concurrencia y empeoró el sistema a alta concurrencia. Una corrección apropiada — un advisory lock o una clave de idempotencia — habría sido correcta a cualquier escala. El agente eligió la corrección que hacía el test verde, no la corrección que era *correcta*.

Lo que haces diferente ahora: Tratas los tests que pasan como necesarios pero no suficientes. Cuando un agente arregla un problema de concurrencia, te preguntas *a ti mismo* si la corrección es correcta bajo carga, bajo fallo, bajo condiciones que la suite de tests no cubre. Los agentes optimizan para la señal de retroalimentación que les das, y si esa señal es «los tests pasan,» encontrarán el camino más corto a verde — incluso si ese camino es un `time.Sleep`. Tu criterio de ingeniería sobre *por qué* algo funciona importa más que *si* los tests pasan. Esta es la parte del trabajo que aún no se ha automatizado. Apóyate en ella.

14.5 La Amnesia de Contexto

Dos horas dentro de una sesión, tenías una funcionalidad evolucionando bien. Le habías dicho al agente al principio: «No uses ningún ORM — estamos escribiendo SQL directo en este proyecto. Es una decisión deliberada.» El agente acusó recibo y escribió consultas limpias, hechas a mano, para las primeras varias tareas.

Luego le pediste que añadiera un nuevo endpoint. Noventa minutos de contexto se habían acumulado. El agente construyó el endpoint — usando Prisma. Archivo de esquema completo, migración, cliente generado. Código hermoso. Contradiciendo completamente la restricción que habías establecido al inicio de la sesión y los patrones en cada otro archivo que había escrito ese día.

Cuando lo señalaste, el agente se disculpó, reescribió todo en SQL directo y actuó como si nada hubiera pasado. No había *decidido* ignorar tu restricción. Simplemente la había perdido. La ventana de contexto se había llenado con suficiente trabajo intermedio que la instrucción temprana se había desvanecido en irrelevancia.

Lo que haces diferente ahora: Las sesiones largas se degradan. Esta es una propiedad fundamental de cómo funcionan las ventanas de contexto, no un bug que se parcheará el próximo trimestre. Mantén las tareas cortas y enfocadas. Haz commit del código que funciona en los límites naturales para que el progreso se capture en git, no solo en la conversación. Empieza sesiones nuevas para tareas nuevas. Y para restricciones de proyecto como «no ORM» o «no nuevas dependencias,» ponlas en un archivo `CLAUDE.md` o equivalente que el agente lea al arrancar. No confíes en que el agente *recuerde* lo que dijiste hace dos horas. No lo hará. Escríbelo.

14.6 La Avalancha de Dependencias

Pediste un selector de fechas. Una entrada simple donde los usuarios pueden seleccionar una fecha. El agente evaluó las opciones y decidió ser exhaustivo.

Instaló `moment.js` para el manejo de fechas. Luego `@popperjs/core` para posicionar el desplegable. Luego una biblioteca completa de componentes UI porque «proporciona primitivas accesibles de selector de fechas.» Luego un preprocesador CSS porque el sistema de temas de la biblioteca de componentes lo requería. Luego dos paquetes utilitarios que el selector de fechas de la biblioteca de componentes necesitaba como peer dependencies.

Seis nuevas dependencias. Tu tamaño de bundle pasó de 180KB a 540KB. Tu tiempo de build se duplicó. Tenías un selector de fechas, eso sí. Muy bonito.

El `<input type="date">` nativo de HTML habría sido suficiente. O una sola biblioteca ligera de selector en 8KB. En su lugar habías heredado un ecosistema entero porque el agente optimizó para completitud en lugar de minimalismo.

La peor parte no fue el tamaño del bundle — fue la superficie de mantenimiento. Seis paquetes nuevos significan seis cosas nuevas que pueden tener vulnerabilidades de seguridad, seis cosas nuevas que pueden romperse en una actualización, seis changelogs nuevos que leer cuando Dependabot empiece a abrir PRs. No solo añadiste un selector de fechas. Adoptaste seis proyectos open source.

Lo que haces diferente ahora: Restringe lo que los agentes pueden instalar. «No nuevas dependencias sin preguntarme primero» es una instrucción legítima y a menudo *sabia*. Cuando permitas nuevos paquetes, dile al agente tus restricciones: presupuesto de bundle, no paquetes con menos de N descargas semanales, no paquetes que traigan mega-dependencias transitivas. Los agentes no tienen opiniones sobre el tamaño del bundle o la carga de mantenimiento. Ellos no mantienen el proyecto el año que viene. *Tú* sí. Así que tú pones los límites.

14.7 El Hilo Común

Cada una de estas historias tiene la misma causa raíz: el agente estaba haciendo *exactamente para lo que fue diseñado*, y el humano no estaba proporcionando suficiente estructura.

El refactorizador entusiasta estaba siendo servicial. La biblioteca alucinada estaba siendo creativa. El bucle infinito estaba siendo persistente. La respuesta equivocada con confianza estaba siendo guiada por tests. La amnesia de contexto era una limitación, no una elección. La avalancha de dependencias estaba siendo exhaustiva.

Ninguna de estas son bugs del agente. Son bugs de *flujo de trabajo*. La solución nunca es «usa un agente más inteligente.» La solución es siempre la misma: alcance más estricto, mejores bucles de retroalimentación, más estructura, sesiones más cortas, y un humano que se mantiene involucrado.

Los agentes se equivocan. Los humanos también. La diferencia es que los humanos se equivocan lo suficientemente despacio como para notarlo. Los agentes se equivocan a la velocidad del autocompletado, y para cuando levantas la vista de tu café, treinta y dos archivos han cambiado.

Mantente en el bucle. Revisa los diffs. Confía pero verifica. Y cuando las cosas salgan mal — porque lo harán — recuerda que el botón de borrar-rama-y-empezar-de-nuevo es la herramienta más infravalorada de tu flujo de trabajo.

14.8 El Manual de Diagnóstico

Las historias de guerra son entretenidas. Pero no puedes pegar anécdotas en tu monitor. Lo que necesitas es un enfoque sistemático — una checklist para cuando el agente produce algo incorrecto, para que puedas diagnosticar el fallo rápido y arreglar lo correcto en lugar de ir a ciegas.

Cuando un agente te da mala salida, recorre estas preguntas en orden. La mayoría de los fallos caen en una de seis categorías, y saber en cuál estás determina qué hacer a continuación.

1. ¿Es un problema de alcance?

¿Pediste demasiado en un prompt? El Refactorizador Entusiasta ocurrió porque «arregla el z-index» era demasiado vago — no decía «no toques nada más.» Si el agente cambió archivos que no esperabas, o hizo trabajo que no pediste, el alcance era demasiado suelto. Ajusta el prompt. Sé

explícito sobre qué *no* hacer. Las restricciones son más útiles que las instrucciones cuando lidias con un agente entusiasta.

2. ¿Es un problema de contexto?

¿El agente tenía lo que necesitaba para resolver el problema real? Recuerda la historia del bug de facturación del Capítulo 2 — un ingeniero dio contexto vago y obtuvo una respuesta vaga, el otro dio archivos específicos y trazas de error y obtuvo una corrección funcional. Si el agente resolvió el problema *equivocado*, probablemente no podía ver el correcto. Aliméntalo con los archivos relevantes, la salida de errores, las restricciones que no puede inferir. Los agentes no adivinan bien. Trabajan con lo que les das.

3. ¿Es un problema de señal de retroalimentación?

¿Tus tests realmente verifican lo correcto? La Respuesta Equivocada Con Confianza ocurrió porque los tests pasaron — pero no testeaban bajo condiciones realistas. El sleep de 500ms como «corrección» era verde en CI e incorrecto en producción. Si la solución del agente te parece dudosa pero los tests pasan, *tus tests son el problema.* El agente optimizó para la señal que le diste. Dale una señal mejor.

4. ¿Es un problema de capacidad?

Algunas tareas están genuinamente más allá de lo que el modelo puede hacer. Razonamiento multi-archivo a través de un sistema extenso con dependencias implícitas. Problemas sutiles de concurrencia que requieren entender el comportamiento en runtime, no solo la estructura del código. Lógica sensible a la seguridad donde «plausible» no es suficiente. Si el agente sigue intentando diferentes enfoques y fallando, puede que no sea capaz de esta tarea particular. Eso no es un problema de prompt — es una limitación. Reconócelo y toma el control. Tu tiempo se emplea mejor haciendo el trabajo que ingeniando el prompt perfecto para una tarea que el agente no puede manejar.

5. ¿Es un problema de ventana de contexto?

Las sesiones largas se degradan. Punto. La historia de Amnesia de Contexto lo demostró — restricciones establecidas al principio de una sesión simplemente se pierden a medida que el contexto se llena con trabajo intermedio. Si el agente contradice algo que hizo correctamente antes en la

misma sesión, o ignora una restricción que estableciste hace una hora, la ventana de contexto es el culpable. Empieza una sesión nueva. Reestablece las restricciones relevantes. Dale solo el contexto que necesita para la tarea actual, no el registro arqueológico de todo lo que habéis discutido hoy.

6. ¿Es un bucle?

Tres intentos en el mismo error es el límite. Si el agente no lo ha resuelto en tres pasadas, no lo resolverá en treinta. La historia del Bucle Infinito quemó 200.000 tokens en diecinueve intentos sin progreso. Cuando veas el patrón — error, corrección, error diferente, corrección, error original — rompe el bucle inmediatamente. Detén al agente. Lee los errores tú mismo. O dale al agente un enfoque completamente diferente con un diagnóstico fresco, o toma el control por completo. El contexto del agente ahora está contaminado con teorías fallidas, y más intentos solo añadirán más contaminación.

Imprime esta checklist. Pégala en tu monitor. Las primeras veces la recorrerás conscientemente. Después de un mes, se convierte en instinto. Después de tres meses, detectarás el problema antes de que el agente termine su primer intento.

15 Cuándo No Usar Agentes

Este libro trata sobre usar agentes bien. Este capítulo trata sobre saber cuándo no usarlos en absoluto.

Si has leído hasta aquí, probablemente estás convencido de la ingeniería agéntica. Bien. Pero la forma más rápida de perder credibilidad — y tiempo — es recurrir a un agente cuando el trabajo requiere un humano. Los mejores ingenieros agénticos no son los que más usan agentes. Son los que saben *exactamente* cuándo guardar el agente y hacer el trabajo ellos mismos.

15.1 El Impuesto del Overhead

Cada interacción con un agente tiene un coste. Escribes el prompt. Esperas la salida. Revisas lo que produjo. Arreglas los errores. Lo vuelves a ejecutar si no acertó. Eso es overhead, y es real.

Para tareas complejas que te llevarían una hora, dedicar dos minutos al prompt y la revisión es una ganga. Pero para tareas que puedes hacer en treinta segundos, el overhead hace a los agentes *más lentos*, no más rápidos.

Renombrar una variable. Arreglar un typo. Ajustar un valor de configuración. Añadir una línea de log. Estas son tareas de memoria muscular. Tus dedos conocen las pulsaciones. Para cuando has tecleado un prompt describiendo lo que quieres, podrías haberlo hecho ya.

Esto no es un fallo de los agentes. Es aritmética. Las tareas pequeñas tienen retornos pequeños, y el coste fijo de la interacción con el agente se come el margen. No dejes que la novedad de los agentes te engañe para usarlos en todo. Parte del trabajo simplemente es más rápido a mano.

15.2 Decisiones de Arquitectura Novedosas

Cuando estás diseñando un sistema desde cero — eligiendo entre un monolito y microservicios, decidiendo tu modelo de datos, eligiendo tus patrones de comunicación — el *pensar es el trabajo*. El valor no está en el diagrama o el documento. El valor está en el modelo mental que construyes mientras luchas con los compromisos.

Un agente puede ayudarte a explorar opciones. Puede listar los pros y contras del event sourcing versus CRUD. Puede esbozar cómo se vería una arquitectura particular. Eso es útil como entrada.

Pero delegar la arquitectura misma a un agente significa que no entiendes tu propio sistema. Te costará depurarlo, extenderlo o explicárselo a tu equipo. El trabajo del ingeniero senior es tomar las decisiones difíciles — sopesar los compromisos que no tienen respuestas limpias, decidir qué complejidad vale la pena cargar y cuál no. Ese criterio viene de hacer el trabajo, no de leer el resumen de un agente.

Usa agentes como sparring partner para la arquitectura. No como el arquitecto.

15.3 Código Crítico para la Seguridad

Flujos de autenticación. Cifrado. Control de acceso. Validación de entrada. Manejo de tokens. Estas son áreas donde «parece correcto» no es suficiente.

Los bugs de seguridad son diferentes de los bugs normales. Una función de ordenamiento rota produce salida incorrecta que alguien nota. Una comprobación de autenticación rota no produce *síntomas visibles* hasta que un atacante la encuentra. El código parece bien. Los tests pasan. Y seis meses después estás escribiendo un informe de incidente.

Los agentes producen código plausible. Esa es su fortaleza y, en contextos de seguridad, su peligro. Un fallo sutil en un flujo de validación JWT, una comprobación faltante en una URL de redirección, un canal lateral de timing en una comparación de contraseñas — estos son el tipo de errores que sobreviven la revisión de código porque *se ven bien*.

Escribe el código crítico para la seguridad tú mismo. Revísalo cuidadosamente. Consigue un segundo par de ojos humanos. Si usas un agente para redactar código de seguridad, trata ese borrador con más sospecha de la que darías al primer intento de un desarrollador junior, no menos.

15.4 Cuando Necesitas Aprender

Estás aprendiendo un nuevo lenguaje. Un nuevo framework. Un nuevo paradigma. La tentación es obvia: deja que el agente escriba el código mientras tú te enfocas en el panorama general.

Resístelo.

Si dejas que el agente escriba todo el Rust mientras estás aprendiendo Rust, no has aprendido Rust. Has construido algo que no puedes mantener, depurar ni extender sin el agente. Has creado una dependencia, no una habilidad.

Hay una diferencia crucial entre usar un agente para *explicar* algo y usarlo para *hacer* algo. Preguntar «¿por qué ocurre este error del borrow checker?» construye comprensión. Preguntar «arregla este error del borrow checker» no.

Cuando el objetivo es aprender, ve despacio. Escribe el código tú mismo. Comete los errores tú mismo. Usa agentes como tutores, no como escritores fantasma. La comprensión que construyes luchando con las partes difíciles es todo el punto.

15.5 Decisiones con Carga Emocional

No toda decisión de ingeniería es técnica. Algunas de las más difíciles son humanas.

Deprecar una API de la que depende un socio. Decirle a un stakeholder que su petición de funcionalidad no entrará. Rechazar un deadline que sabes que es irreal. Decidir discontinuar un producto que todavía tiene usuarios.

Estas decisiones requieren empatía. Requieren leer la sala, entender la política, sopesar el coste humano junto al coste técnico. Requieren *responsabilidad* — alguien que sea dueño de la decisión y sus consecuencias.

Un agente puede redactar el email. Puede ayudarte a pensar en los puntos clave. Pero la decisión en sí, y la conversación que la entrega, debe venir de un humano. Las personas merecen escuchar noticias difíciles de una persona, no de alguien que copió y pegó la salida de un agente.

15.6 Cuando el Código Base Es Demasiado Desordenado

Los agentes amplifican lo que ya está ahí. En un código base limpio y bien estructurado con convenciones fuertes, los agentes producen código limpio y bien estructurado. Detectan los patrones y los siguen.

¿En un desastre? Producen más desastre.

Si tu código base tiene nombres inconsistentes, dependencias enredadas, sin límites claros de módulos, y tres formas diferentes de hacer lo mismo — un agente detectará *todos* esos patrones. Podría combinar las peores partes de cada uno. No sabe qué patrones son intencionales y cuáles son deuda técnica. Solo ve lo que hay y produce más de lo mismo.

A veces lo correcto es limpiar antes de traer agentes. Refactorizar el módulo. Establecer la convención. Borrar el código muerto. Hacer del código base un lugar donde un agente pueda hacer buen trabajo. Este es trabajo poco glamuroso, manual. Pero es la base que hace posible todo lo demás.

Piénsalo como un taller. No le das herramientas eléctricas a alguien en un taller desordenado y desorganizado. Primero limpias, *luego* traes las herramientas.

15.7 Trabajando con Código Legacy

Esa metáfora del taller es bonita. Pero no todos tienen el lujo de limpiar primero. Algunos de nosotros mantenemos monolitos Java de 500.000

líneas que empezaron en 2014 y han pasado por tres migraciones de framework, dos adquisiciones, y un breve periodo en el que alguien pensó que la configuración XML era buena idea. El consejo de «refactorizar antes de traer agentes» es sólido en teoría y risible en la práctica cuando estás mirando un código base que llevaría años refactorizar.

Entonces, ¿cómo *usas* agentes en código legacy? Con cuidado. Y con una estrategia específica.

Empieza con los tests. Antes de apuntar un agente a código legacy, escribe tests de caracterización — tests que capturan lo que el código *hace actualmente*, no lo que *debería* hacer. Estos no son aspiracionales. Son descriptivos. Dicen «cuando llamas a esta función con estas entradas, esto es lo que sale, y ese es el contrato actual lo haya pretendido alguien o no.»

Los tests de caracterización le dan al agente una red de seguridad. Sin ellos, el agente cambiará comportamiento que no entiende, y no lo sabrás hasta que producción te lo diga. Con ellos, cualquier cambio de comportamiento aparece como un fallo de test *antes* de que el código salga de tu máquina. Esto es innegociable. Código legacy sin tests es código legacy que no puedes dejar que los agentes toquen de forma segura.

Acota despiadadamente. En un código base legacy, el agente no puede entender todo el sistema. No intentes que lo haga. Apúntalo a un archivo, una función, un bug. Dale el contexto inmediato — la firma de la función, los llamadores, el comportamiento esperado — y nada más. El código legacy está lleno de suposiciones implícitas, efectos secundarios no documentados y peculiaridades estructurales. Cuanto menos vea el agente, menos puede malinterpretar. Un alcance estrecho con contexto claro supera a un alcance amplio con complejidad arqueológica.

Usa agentes para las partes tediosas. Los códigos base legacy están llenos de trabajo mecánico: actualizar llamadas a API deprecadas en cientos de archivos, migrar de una versión de dependencia a otra, añadir anotaciones de tipos a código sin tipar, estandarizar patrones de manejo de errores, reemplazar un framework de logging por otro. Estas son tareas *perfectas* para agentes — repetitivas, bien definidas, fácilmente verificadas por comprobaciones automatizadas. Deja que los agentes hagan el trabajo tedioso. Guarda tu energía para las partes que requieren entender *por qué*

el código es como es. Ese es el trabajo que solo un humano que ha vivido con el sistema puede hacer.

Documenta sobre la marcha. Cada vez que un agente trabaja exitosamente en un archivo legacy, añade un breve comentario explicando qué hace el código, o actualiza el `CLAUDE.md` del proyecto con contexto sobre ese módulo. Con el tiempo, algo interesante pasa: las partes del código base legacy que los agentes han tocado se vuelven mejor documentadas que las partes que no. No porque te propusieras documentar el sistema, sino porque usar agentes te *forzó* a articular qué hace cada pieza para poder hacer prompts efectivos. Los agentes están lentamente haciendo el código base más legible — no refactorizándolo, sino haciéndote explicarlo.

15.8 El Argumento del Oficio

Hay una razón más para a veces guardar el agente, y no se trata de eficiencia o riesgo. Se trata del oficio.

Escribir código a mano construye algo que los agentes no pueden darte. Memoria muscular. Reconocimiento de patrones que vive en tus dedos, no solo en tu cabeza. La comprensión profunda e intuitiva que viene de haber escrito el mismo tipo de función docenas de veces. La satisfacción silenciosa de resolver un problema difícil a través de tu propio razonamiento.

Estas cosas importan. No porque sean románticas, sino porque te hacen mejor ingeniero. El desarrollador que ha escrito un parser a mano entiende el parsing de forma diferente que uno que solo ha hecho prompts a un agente para que escriba uno. El desarrollador que ha depurado un problema de concurrencia a las 2am *siente* el peligro del estado mutable compartido de una forma que leer sobre ello nunca proporciona.

Los agentes son herramientas. Potentes. Pero si los dejas hacer todo el trabajo, tus propias habilidades se atrofian. Y cuando te encuentres con un problema que el agente no puede resolver — y lo harás — necesitas que esas habilidades sigan afiladas.

Mantente en forma. Escribe código a mano regularmente. No porque sea más rápido, sino porque te mantiene peligroso.

15.9 El Criterio Que Importa

La habilidad central de la ingeniería agéntica no es el prompting. No es la configuración de herramientas. No es saber qué modelo usar para qué tarea.

Es *criterio*.

Saber cuándo un agente te ahorrará una hora y cuándo desperdiciará una. Saber cuándo confiar en la salida y cuándo reescribirla desde cero. Saber cuándo delegar y cuándo meterte de lleno tú mismo.

Los mejores ingenieros agénticos usan agentes agresivamente — pero selectivamente. Recurren a agentes cuando el apalancamiento es real: refactorizaciones a gran escala, generación de boilerplate, exploración de código, escritura de tests, documentación. Y guardan el agente cuando el trabajo requiere pensamiento humano, responsabilidad humana u oficio humano.

Ese criterio es lo que separa a los ingenieros agénticos de los meros operadores de prompts. Cualquiera puede teclear un prompt. Saber *cuándo no hacerlo* es la habilidad más difícil.

16 Equipos Agénticos

El software siempre ha sido un deporte de equipo. Los agentes no cambian eso. Pero cambian la forma del equipo, el ritmo del trabajo y las habilidades que importan. Si lideras un equipo — o trabajas en uno — necesitas pensar en esto ahora, no después.

16.1 El Multiplicador 10x Es Real, Pero Se Distribuye Diferente

Has escuchado la afirmación: los agentes hacen a los ingenieros 10x más productivos. No es incorrecta. Solo es engañosa.

Los agentes hacen ciertas tareas *dramáticamente* más rápidas. Generar boilerplate, escribir scaffolding de tests, refactorizar a lo largo de cien archivos, migrar versiones de API, cablear endpoints CRUD — esto pasa de horas a minutos. La ganancia de velocidad es real y es masiva.

Pero otras tareas apenas cambian. El diseño de sistemas sigue requiriendo pensar. Depurar una condición de carrera sutil sigue requiriendo paciencia, intuición y comprensión profunda del runtime. Las conversaciones con stakeholders siguen llevando el tiempo que llevan. El agente no puede sentarse en tu revisión de arquitectura y hacer las preguntas correctas.

El multiplicador 10x no es un multiplicador plano a lo largo de tu día. Es un gráfico de picos. Algunas tareas van 50x más rápido. Algunas van 1x. Unas pocas podrían incluso ralentizarse si peleas con el agente en lugar de hacerlo tú mismo.

Los equipos que entienden esto despliegan agentes estratégicamente. No le pasan cada tarea a un agente esperando magia. Identifican las zonas de alto apalancamiento — las tareas donde los agentes genuinamente comprimen los plazos — y enfocan el esfuerzo de los agentes ahí. El resto sigue siendo humano.

16.2 La Revisión de Código Cambia

Esto es lo que pasa cuando los agentes entran en el flujo de trabajo de un equipo: el volumen de PRs sube. *Mucho*. Un ingeniero que solía abrir dos PRs al día ahora abre cinco. El código en esos PRs es sintácticamente correcto, bien formateado y pasa los tests. Y revisarlo es *agotador*.

El viejo modelo de revisión de código — leer cada línea, buscar errores off--by-one, verificar el manejo de casos límite — no escala cuando la mitad del código fue generado en segundos. Quemarás a tus revisores en una semana.

El cambio es de «¿es esto correcto?» a «¿es este el enfoque correcto?» El código generado por agentes suele ser *localmente* correcto. Hace lo que se pidió. La pregunta es si lo que se pidió era lo correcto. ¿Necesita existir este nuevo servicio, o debería vivir esta lógica en el existente? ¿Es este el límite de abstracción correcto? ¿Este cambio hace el sistema más simple o más complejo?

Los revisores se convierten en arquitectos. Hacen zoom out. Comprueban intención, no implementación.

Adaptaciones prácticas que funcionan:

- PRs más pequeños y enfocados — más fáciles tanto para agentes como para revisores
- Las comprobaciones automatizadas manejan lo mecánico (linting, cobertura de tests, comprobación de tipos)
- El tiempo de revisión se protege en el calendario, no se exprime entre reuniones
- Los equipos acuerdan «patrones de confianza» — si un PR sigue un patrón conocido y pasa CI, obtiene una vía de revisión más rápida

La fatiga de revisión es el asesino silencioso de los equipos asistidos por agentes. Tómalo en serio.

16.3 La Pregunta del Ingeniero Junior

Este es el problema más difícil de este capítulo, y no tengo una respuesta limpia.

Los ingenieros junior tradicionalmente han aprendido haciendo el trabajo que los agentes ahora hacen más rápido y mejor. Escribir ese primer endpoint CRUD. Luchar con un layout CSS complicado. Descubrir por qué el test está fallando. Estas tareas eran tediosas para los seniors pero *formativas* para los juniors. La lucha era la educación.

Si los agentes manejan todo eso, ¿dónde ocurre el aprendizaje?

El peor resultado son juniors que se vuelven dependientes de los prompts — pueden hacer que un agente genere código, pero no pueden explicar qué hace el código ni depurarlo cuando se rompe. Se saltan la fase de comprensión completamente. Eso no es ingeniería; es un flujo de trabajo de copiar y pegar muy caro.

El mejor camino es usar agentes como *tutores*, no como reemplazos. Un junior escribiendo una migración de base de datos debería pedirle al agente que le *explique* la migración, no que simplemente la genere. «¿Por qué usaste una transacción aquí?» «¿Qué pasa si esto falla a mitad?» «Muéstrame cómo se ve esto sin el ORM.» El agente se convierte en un profesor paciente con tiempo infinito — algo que la mayoría de los seniors no pueden ofrecer.

El pair programming con agentes, *supervisado por seniors*, es el modelo más prometedor que he visto. El junior dirige al agente. El senior observa, hace preguntas e interviene cuando el junior acepta algo que no entiende. Es más lento que dejar que el agente haga todo, pero produce ingenieros que realmente saben lo que están haciendo.

Los equipos que se saltan esta inversión están tomando prestado del futuro. Los juniors sin supervisión de hoy son los seniors de mañana que no pueden depurar producción sin una muleta de IA.

16.4 Distribución del Conocimiento

Aquí hay un patrón que aparece en cada equipo que adopta agentes de forma desigual: un ingeniero se vuelve fluido con los agentes, empieza a entregar a 3x la velocidad de los demás, y en pocos meses ha tocado cada parte del código base. Se convierten en el punto único de fallo.

El factor bus cae a uno. No porque nadie lo planificara, sino porque la velocidad y la concentración de conocimiento están correlacionadas. El ingeniero que más entrega aprende más. Los demás se quedan atrás, no solo en producción sino en *comprensión* del sistema que colectivamente poseen.

Este es un problema de gestión, no de tecnología. La solución es estructural:

- **Archivos `CLAUDE.md` compartidos.** Cada proyecto tiene uno. Todos contribuyen a él. Codifica el conocimiento colectivo del equipo, no el de una persona.
- **Flujos de trabajo y convenciones compartidos.** El equipo acuerda cómo usan agentes — qué herramientas, qué patrones, qué barandillas. Sin configuraciones de lobo solitario.
- **Rotación.** Los ingenieros fluidos con agentes rotan a diferentes partes del código base. El conocimiento se propaga a través del trabajo, no de la documentación.
- **Compartir sesiones de agente.** Algunos equipos han empezado a compartir sesiones de agente interesantes — los prompts, las salidas, las decisiones. Es una forma de transferencia de conocimiento que no existía antes.

El objetivo no es ralentizar a tu ingeniero más rápido. Es asegurarte de que el conocimiento del equipo se mantiene al ritmo de la producción del equipo.

16.5 Las Convenciones Compartidas Importan Más

Cubrimos las convenciones en un capítulo anterior. En el contexto de equipo, las implicaciones son mayores.

Cuando un ingeniero solo usa agentes, sus convenciones afectan a una persona. Cuando un equipo usa agentes, las convenciones afectan *cada sesión de agente en todo el equipo*. Un proyecto bien estructurado con nombres claros, patrones consistentes y un `CLAUDE.md` mantenido significa que los agentes de cada ingeniero empiezan desde una base sólida.

Un proyecto desordenado significa que cada agente reinventa la rueda. Diferentes ingenieros obtienen salidas diferentes. El código base se desvía. Las revisiones se vuelven más difíciles porque ahora estás revisando no solo el código sino el *estilo* del código, que varía según qué agente de qué ingeniero lo escribió.

Los estándares de código en un equipo agéntico no se tratan de estética. Se tratan de *efectividad del agente*. El equipo que acuerda la estructura del proyecto, convenciones de nombres, patrones de testing y formato de documentación obtiene mejor salida de cada sesión de agente. Es un multiplicador de fuerza sobre un multiplicador de fuerza.

Invierte el tiempo. Escribe los estándares. Imponlos en CI. Se devuelve con cada PR.

16.6 El Nuevo Standup

¿Cómo se ve la coordinación diaria cuando cada ingeniero está ejecutando múltiples sesiones de agentes en paralelo?

El viejo standup: «Ayer trabajé en la refactorización de autenticación. Hoy seguiré. Sin bloqueos.»

El nuevo standup: «Tengo tres agentes ejecutándose. La refactorización de autenticación se entregó y está en revisión. La migración de API está al 80% — el agente se quedó atascado en el parseo de XML legacy, así que estoy tomando el control manualmente. Acabo de lanzar una tercera sesión para generar tests de integración para el módulo de facturación.»

La granularidad cambia. El trabajo se mueve más rápido, así que la coordinación necesita mantenerse al día. Un ingeniero podría *empezar y terminar* una tarea entre standups. La sincronización diaria pasa de ser actualizaciones de progreso a alineación de intención — asegurarse de que tres ingenieros no están todos enviando agentes al mismo problema desde diferentes ángulos.

Algunos equipos se están moviendo a standups asíncronos con check-ins más frecuentes. Otros usan dashboards compartidos que rastrean sesiones de agentes activas. La respuesta correcta depende del equipo, pero el

viejo ritmo de «una actualización por persona por día» a menudo no es suficiente.

16.7 Cumplimiento y la Pista de Auditoría

Si un agente escribe código que causa un incidente en producción, ¿quién es responsable? ¿El ingeniero que hizo el prompt? ¿El revisor que aprobó el PR? ¿El líder del equipo que decidió adoptar flujos de trabajo agénticos? Esta no es una pregunta filosófica que debatas con unas cervezas. Es una pregunta legal y de cumplimiento, y las industrias reguladas necesitan respuestas claras *antes* de que ocurra el incidente, no durante el postmortem.

La buena noticia es que la respuesta no es realmente tan complicada. El tooling y los procesos ya existen. Solo necesitas ser explícito sobre ellos.

Git es tu pista de auditoría. Cada commit generado por agente debería ser atribuible. El mensaje de commit debería indicar que fue asistido por agente — un tag `Co-Authored-By`, un prefijo, cualquier convención que adopte tu equipo. El PR debería mostrar quién lo revisó. La aprobación del merge es la firma. Así es como ya trabajan la mayoría de los equipos; la clave es hacerlo *consistente*. Atribución ad-hoc — a veces etiquetando, a veces no — es peor que no tener sistema, porque crea la impresión de un proceso sin la fiabilidad de uno.

El revisor es responsable. La respuesta práctica para la mayoría de las organizaciones es directa: el ingeniero que revisa y aprueba el PR asume la responsabilidad, igual que lo haría con cualquier código de cualquier fuente. El código generado por agentes no tiene un estándar de responsabilidad diferente. Si apruebas un PR, estás diciendo «he revisado esto y creo que es correcto.» La herramienta que generó el código es irrelevante para esa declaración. Esto también significa que las revisiones de código generado por agentes necesitan ser revisiones *reales*, no sellos de goma. Si el volumen de PRs generados por agentes está haciendo imposible una revisión exhaustiva, eso es un problema de flujo de trabajo que resolver, no un estándar que bajar.

Para industrias reguladas. Documenta tu flujo de trabajo agéntico como parte de tu documentación del SDLC. Qué modelos se usan, qué versión, qué barandillas están en su lugar, qué proceso de revisión sigue el código generado por agentes antes de llegar a producción. Los auditores quieren ver un *proceso*, no perfección. Un proceso documentado que incluye «generación de código asistida por IA con revisión humana obligatoria y verificación CI» es auditable. Un proceso no documentado donde los ingenieros usan las herramientas que quieran sin enfoque consistente no lo es. Si estás en fintech, salud o cualquier cosa con supervisión regulatoria, documenta esto antes de que alguien lo pida.

Guarda logs de sesiones. Retén logs de sesiones de agente, especialmente para código que toca sistemas sensibles — facturación, autenticación, manejo de datos, cualquier cosa con implicaciones regulatorias. No porque los leas rutinariamente, sino porque podrías necesitarlos durante una revisión de incidente. «¿Qué vio el agente cuando generó este código? ¿Qué prompt produjo esta salida? ¿Con qué contexto trabajaba?» Son preguntas que quieres poder responder seis meses después. La mayoría de las herramientas agénticas pueden exportar o registrar sesiones. Configura la retención antes de necesitarla. El coste de almacenamiento es trivial comparado con el coste de no tener los logs cuando cumplimiento venga a llamar.

16.8 La Contratación Cambia

Si los agentes manejan las partes mecánicas de la codificación, ¿qué necesitas realmente de un ingeniero?

La velocidad bruta de codificación importa menos. El ingeniero que podía escribir un binary search perfecto en una pizarra en tres minutos tiene una habilidad que los agentes han convertido en commodity. Eso no es inútil — entender algoritmos sigue importando — pero ya no es el diferenciador.

Lo que importa más:

- **Diseño de sistemas.** La capacidad de descomponer un problema en componentes, definir interfaces y razonar sobre compromisos. Los

agentes pueden implementar un diseño. No pueden decirte qué diseño es correcto para tus restricciones.

- **Criterio.** Saber cuándo usar el agente y cuándo pensar. Saber cuándo la salida del agente está equivocada aunque parezca correcta. Saber qué esquinas recortar y cuáles proteger.
- **Comunicación.** La capacidad de articular intención con claridad — a agentes, a compañeros de equipo, a stakeholders. Los pensadores vagos obtienen salida vaga del agente. Los pensadores precisos obtienen resultados precisos.
- **Descomposición de problemas.** Dividir una tarea grande en piezas del tamaño de un agente es una habilidad. Está relacionada con el diseño de sistemas pero es más táctica. El ingeniero que puede convertir un Jira epic en diez prompts de agente bien acotados superará al que pega el epic entero en una ventana de chat.

La entrevista que pregunta «implementa este algoritmo en una pizarra» está evaluando una habilidad que importa menos cada año. La entrevista que pregunta «aquí tienes un sistema con estas restricciones — explícame cómo lo construirías, qué compromisos harías, y cómo verificarías que funciona» está evaluando las habilidades que importan *más* cada año.

Contrata por criterio. Siempre puedes darles un agente para el resto.

17 Palabras Finales

Mi hijo mayor tiene cinco años. Todavía no sabe leer, no realmente — deletrea palabras en cajas de cereales y carteles de la calle, orgulloso de cada sílaba. Pero sabe lo que hace mi ordenador. Me ha visto hablarle, ha visto texto aparecer en la pantalla en respuesta, me ha visto asentir o negar con la cabeza y hablar de nuevo. Una noche se subió a mi regazo, vio a un agente refactorizar un módulo en tiempo real — archivos abriéndose y cerrándose, tests ejecutándose, marcas verdes apareciendo — y dijo: «¿El ordenador se está arreglando solo?»

No tuve una buena respuesta. Todavía no la tengo del todo. Pero esa pregunta me acompañó a lo largo de cada capítulo de este libro. Porque lo que me di cuenta, sentado ahí con él, es que no estaba escribiendo un libro sobre herramientas. Estaba escribiendo un libro sobre lo que significa ser ingeniero en el momento exacto en que la definición de ingeniería se está reescribiendo — y sobre lo que llevamos adelante hacia lo que venga después.

Este capítulo no es un resumen. No necesitas que te recapitule quince capítulos que acabas de leer. Esto es lo que quiero decirte directamente, de ingeniero a ingeniero, antes de que nos separemos.

17.1 Lo Que Creo

Creo que el oficio no está muriendo. Está siendo *comprimido*. Una década de boilerplate y fontanería y ceremonia se está colapsando en intención y criterio. Lo que queda cuando quitas el teclear es lo que siempre fue realmente el trabajo: saber qué construir y saber cuándo está bien.

Creo que los agentes son amplificadores, no reemplazos. Dale un agente a un ingeniero mediocre y obtienes software mediocre a velocidad aterradora. Dale uno a un gran ingeniero y obtienes algo que, francamente, hace

que los últimos veinte años de desarrollo de software parezcan como si estuviéramos construyendo autopistas con palas.

Creo que los ingenieros que prosperarán serán los que dominen las cosas *aburridas* — contexto, barandillas, testing, convenciones, pensamiento claro — mientras todos los demás persiguen el anuncio brillante del nuevo modelo. Las herramientas cambian cada trimestre. El criterio se acumula a lo largo de una carrera.

Creo que no estamos siendo reemplazados. Estamos siendo promovidos. De mecanógrafos a capitanes. De programadores a *ingenieros*, en el sentido más completo de la palabra. La pregunta es si aceptas la promoción.

Creo que este cambio es *bueno*. No fácil. No indoloro. Pero bueno. Porque la parte de nuestro trabajo que se está automatizando nunca fue la parte que amábamos. Nadie se hizo ingeniero porque soñaba con escribir parsers de JSON boilerplate. Nos hicimos ingenieros porque queríamos construir cosas que importan. Ahora podemos.

17.2 En Qué Me Equivoqué

Seré honesto contigo: cambié de opinión al menos tres veces mientras escribía este libro.

Cuando empecé el capítulo de sandboxes, pensaba que el aislamiento con contenedores era excesivo para la mayoría de los flujos de trabajo. Para cuando lo terminé, después de que un agente hiciera rm -rf en un directorio que me importaba un martes por la tarde, creía que el sandboxing era innegociable. Esa experiencia entró en el capítulo. La convicción detrás de ella es tejido cicatricial.

Originalmente escribí el capítulo multi-agente con la suposición de que orquestar cinco o seis agentes simultáneamente era el estado final natural — una planta de fábrica de trabajadores autónomos. He retrocedido de eso. El overhead de coordinación es real, los modos de fallo se multiplican, y he descubierto que dos o tres agentes bien dirigidos superan a seis sin supervisión casi siempre. El capítulo refleja dónde terminé, pero podría terminar en otro sitio en seis meses.

También fui, durante un tiempo, demasiado despectivo con los modelos locales. Escribí un borrador temprano que básicamente decía «simplemente usa las APIs comerciales.» Luego pasé un fin de semana ejecutando un modelo local fine-tuned en un código base con restricciones propietarias y me di cuenta de que hay todo un mundo de casos de uso donde lo local no es solo viable sino *necesario*. El capítulo sobre modelos locales versus comerciales existe porque estaba equivocado y tuve que corregirme.

Partes de este libro probablemente ya estén equivocadas de formas que no puedo ver todavía. El panorama se mueve así de rápido. Pero las herramientas específicas nunca fueron el punto. Si te he ayudado a construir un modelo mental — una forma de pensar sobre autonomía, confianza y estructura — entonces el libro cumplió su función, incluso cuando cada ejemplo de código en él esté desactualizado.

17.3 La Metáfora de la Tripulación, Una Última Vez

Crecí en Dinamarca, cerca del agua. Si has navegado en aguas escandinavas, conoces la luz de finales de verano — baja y dorada, el tipo de luz que hace que el mar parezca cobre martillado. Recuerdo una travesía una vez, un barco pequeño, cuatro de nosotros. El viento cambió fuerte y de repente todos estábamos moviéndonos sin hablar. Uno en el foque, uno en la escota mayor, uno ajustando, uno al timón. Sin órdenes. Solo confianza construida a lo largo de docenas de navegaciones previas.

Eso es lo que se siente la ingeniería agéntica en un buen día. Tú al timón, los agentes ajustando y afinando, el trabajo fluyendo porque has invertido las horas en construir contexto compartido — a través de convenciones, a través de barandillas, a través de suites de tests que detectan errores antes de que importen. No necesitas gritar instrucciones. El sistema *sabe*.

Y luego están los malos días. Los días en que el viento cae y un agente alucina una API que no existe, o reescribe un módulo que no le pediste que tocara, o pasa todos los tests porque borró los que estaban fallando. Esos días, estás achicando agua y maldiciendo. Eso también es navegar.

Pero debería ser honesto sobre algo, porque este libro no funciona si no lo soy.

La metáfora de una *tripulación* implica lealtad. Continuidad. Compañeros de barco con los que has navegado antes, que conocen tus hábitos, que anticipan la siguiente orden. Es una imagen hermosa. Tampoco es cómo trabajo realmente.

La mayoría de los días, lo que realmente hago es levantar un agente, darle un trabajo, tomar la salida y tirarlo por la borda. Luego levanto otro. No recuerdan la última sesión. No saben lo que le pedí al agente anterior. Cada uno llega fresco, hace su trabajo y desaparece. Es menos una tripulación leal y más como contratar estibadores en cada puerto — les informas, los ves trabajar, les pagas y en el siguiente puerto lo haces de nuevo.

Y eso está bien. Así es como la mayoría de las tripulaciones reales funcionaron a lo largo de la historia marítima. El barco era la continuidad. El capitán era la continuidad. Las cartas, el cuaderno de bitácora, el aparejo — eso persistía entre viajes. La tripulación a menudo se reunía para una sola travesía y se disolvía en el destino. Lo que lo hacía funcionar no era que los marineros conocieran al capitán. Era que el capitán conocía el *barco* — y tenía sistemas lo suficientemente buenos para que cualquier marinero competente pudiera subir a bordo y ser útil.

Eso es lo que es tu código base. Eso es lo que son tus convenciones, tus suites de tests, tus archivos CLAUDE.md, tus barandillas. Son el barco. Cada nuevo agente que levantas es un nuevo miembro de la tripulación subiendo a bordo de un navío bien aparejado. No necesitan conocer tu historia. Necesitan conocer el barco. Y si has construido el barco bien, serán productivos en minutos.

Así que sí — tíralos por la borda. Levanta nuevos. Eso no es un fallo de la metáfora. Es la metáfora funcionando exactamente como se pretendía. La tripulación es desechable. El barco no lo es.

Tú eres el capitán. Siempre fuiste el capitán. La tripulación acaba de llegar — y seguirán llegando, frescos y listos, cada vez que los necesites.

17.4 Gracias

Gracias por leer este libro. Lo digo en serio. Intercambiaste tu tiempo y atención por mis palabras, y no me lo tomo a la ligera. Espero haberlo merecido.

Gracias a la comunidad — los ingenieros en foros, en servidores de Discord, en repositorios open source — que están compartiendo sus experimentos, sus fallos, sus ideas ganadas con esfuerzo. Este libro fue moldeado por cientos de conversaciones que no tuve solo.

Y gracias, supongo, a los propios agentes — que me ayudaron a escribir código, depurar problemas, y ocasionalmente generar prosa tan mala que me recordó por qué el criterio humano sigue importando. Sois una buena tripulación. Estáis mejorando. Y sospecho que para cuando mi hijo sea lo suficientemente mayor para leer este libro, seréis algo que ninguno de nosotros predijo del todo.

17.5 Ve y Construye Algo

Esto es lo que quiero que hagas.

Esta noche — no mañana, no la semana que viene, *esta noche* — abre tu terminal. Elige un bug que has estado evitando. Ese que sigues moviendo al fondo del backlog, el que es molesto pero no urgente, el que vive en una parte del código base que preferirías no tocar. Apunta un agente. Dale contexto. Establece una barandilla. Mira qué pasa.

Quizá arregla el bug en cuatro minutos y sientes el suelo moverse bajo tus pies, como se movió bajo los míos aquella noche con el script de migración. Quizá hace un desastre y aprendes algo sobre cómo dar mejores instrucciones. De cualquier forma, sabrás más que antes.

Luego ve a más. Configura aislamiento con worktrees y ejecuta dos agentes en paralelo. Escribe un archivo CLAUDE.md para un proyecto que te importa. Construye una suite de tests lo suficientemente buena como para ser tu red de seguridad cuando los agentes estén haciendo commits. Refactoriza ese módulo que todo el mundo teme — pero esta vez, con una tripulación.

Luego ve más grande todavía. Introduce flujos de trabajo agénticos en tu equipo. Comparte lo que has aprendido — las victorias *y* los desastres. Escribe tus propias historias de guerra. Contribuye al conocimiento colectivo de una disciplina que todavía se está inventando.

Porque eso es lo que es esto: una disciplina que se está inventando, ahora mismo, por las personas dispuestas a hacer el trabajo. No por las personas escribiendo posts de blog sobre el futuro. No por las personas esperando la herramienta perfecta. Por las personas que abren un terminal hoy y construyen algo real con lo que tienen.

Los ingenieros que definirán esta era no están esperando permiso. No están esperando certeza. Están entregando, rompiendo cosas, aprendiendo y volviendo a entregar — con una tripulación a su lado que mejora un poco cada día.

Espero que tú seas uno de ellos.

Rasmus Bornhøft Schlünsen
Marzo 2026

A mis hijos —
sois la mejor tripulación que he tenido.
Todo esto fue para vosotros.
Siempre.