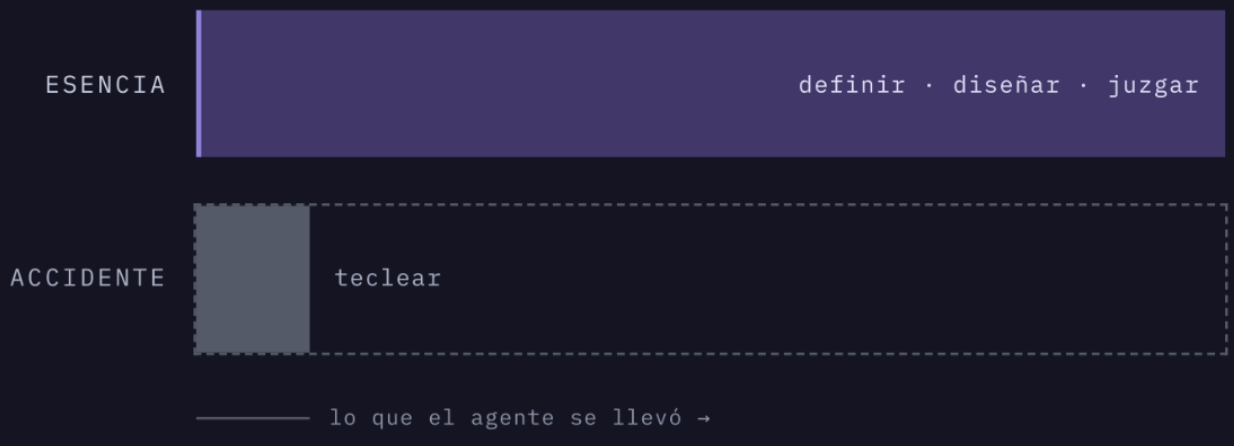
VOLUMEN AVANZADO · COMPLEMENTO DE ARCHITECTURE FOR VIBE CODERS

Vibe Coding para Desarrolladores

Arquitectura, práctica y equipo
en la era del código asistido por IA



Diógenes Alberto Moreira

LA VARA SUBE

Vibe Coding para Desarrolladores

Arquitectura, práctica y equipo en la era del código asistido por IA — volumen avanzado

Índice

Vibe Coding para Desarrolladores Arquitectura, práctica y equipo en la era del código asistido por IA

© 2026, Diógenes Alberto Moreira

ISBN 978-631-01-6918-7

Primera edición.

Todos los derechos reservados. Ninguna parte de esta publicación puede ser reproducida, almacenada o transmitida por ningún medio sin la autorización previa y por escrito del autor, salvo citas breves con fines de reseña o estudio.

Agradecimientos

A Máximo Prieto, que me enseñó a programar pensando en objetos y, sobre todo, a pensar. Su manera de ver el oficio —que todo es un objeto, que todo es un mensaje— me formó de un modo que todavía hoy sostiene cómo escribo software y cómo lo razono. Máximo ya no está, y estas páginas llevan, entre líneas, su memoria. Ojalá algo de su claridad haya quedado en ellas.

A Alan Kay, cuyas ideas sobre los objetos, los sistemas y el valor de un buen punto de vista guiaron mi forma de entender la programación aun desde la distancia. Buena parte de este libro no sería la misma sin haberlo leído y escuchado a lo largo de los años.

A Alan Knight, por lo mismo: un maestro a la distancia, cuyo trabajo y cuyo criterio me acompañaron sin que él lo supiera.

Como siempre a mi familia.

Y a los desarrolladores que dudan, que discuten y que se toman el oficio en serio: este libro es, en el fondo, una conversación con ustedes.

Prólogo

Empiezo a escribir este libro convencido de una cosa: el vibe coding llegó para quedarse.

No lo digo desde la especulación ni desde el entusiasmo fácil que rodea a cada tecnología nueva. Lo digo desde la experiencia. En mi propio trabajo vi aparecer un nivel de productividad que hasta hace muy poco vivía solo en los sueños del visionario más delirante: cosas que antes llevaban semanas resolviéndose en tardes, ideas que morían en el backlog por falta de manos y que ahora se prototipan mientras uno todavía las está pensando. No es magia, y no es gratis —de eso trata buena parte de este libro—, pero es real. Lo suficientemente real como para que valga la pena escribir sobre ello con seriedad.

Sé que esa afirmación no cae bien en todos lados. La comunidad tecnológica tiene, con razón, un músculo escéptico bien entrenado. Vimos pasar suficientes olas como para desconfiar de la próxima que promete cambiarlo todo. Y hay una resistencia que no es capricho: quien dedicó años a dominar un oficio no está dispuesto a aceptar que una herramienta que "adivina" el código lo iguale de un día para el otro. Detrás de ese recelo hay preguntas legítimas —sobre la calidad, sobre la responsabilidad, sobre qué pasa con el criterio que tanto costó construir— y este libro las toma en serio en lugar de barrerlas debajo de la alfombra.

Los que me conocen saben dos cosas de mí: mi distancia con la cátedra y mi fascinación de siempre por la programación orientada a objetos. Ninguna de las dos se movió un milímetro. No perdí la identidad ni perdí la fuerza; sigo pensando exactamente lo mismo que pensaba. La única diferencia es que ahora tengo al lado una asistencia que no se cansa, con manos mucho más rápidas que las mías —y que, además, no afloja nunca. Sigo siendo yo; lo que cambió es el par de manos.

Pero creo que gran parte de la discusión está mal planteada. Se la vive como si hubiera que elegir entre el oficio y la herramienta, entre saber programar y dejar que la máquina programe. Mi experiencia me dice que esa oposición es falsa. El vibe coding no reemplaza al que sabe: lo potencia. Y, hecho sin criterio, es exactamente en manos

del que no sabe donde más daño hace. La destreza técnica, lejos de volverse obsoleta, se vuelve más decisiva que nunca; lo que cambia es dónde se aplica. Menos en teclear cada línea y más en especificar bien, en revisar con rigor y en responsabilizarse por lo que se produce.

Eso es, en el fondo, lo que busca este libro: zanjar esa falsa grieta. No convencer al escéptico a fuerza de eslóganes, sino ofrecerle un terreno común donde su conocimiento siga valiendo —y valga más—, integrado a una forma de trabajar que ya está acá. Quiero sumar capacidad a lo que ya era conocimiento, no sustituirlo. Este es un libro para desarrolladores que conocen su oficio; el complemento avanzado del anterior, que se ocupaba de los fundamentos. Si el primero explicaba qué es esto y por qué importa, este se mete en el cómo: la práctica diaria, las decisiones de arquitectura, las herramientas y, sobre todo, el trabajo en equipo, que es donde estas ideas se ponen a prueba de verdad.

Y quiero ser claro sobre a quién le hablo. Este libro está escrito, sobre todo, para el desarrollador que todavía no incorporó el vibe coding a su forma de trabajar: el que lo mira de reojo, el que probó una vez y no le encontró la vuelta, o el que simplemente decidió esperar a ver. No vengo a apurarte ni a hacerte sentir atrasado. Vengo a acompañarte a aggiornarte —a ponerte al día— en un mundo que, te guste o no, ya llegó y no se va a ir. Tu experiencia no es un lastre que haya que dejar atrás para empezar de nuevo: es, justamente, el capital que va a hacer que uses esto mejor que muchos de los que ya se subieron. Este libro es el puente entre el oficio que ya tenés y la forma de trabajar que se está volviendo el estándar. No te pido que abandones nada de lo que sabés; te pido que lo pongas a jugar en un tablero nuevo.

Lo escribo, además, desde una convicción más personal. Buena parte de lo que se publica sobre estos temas llega desde los grandes centros donde se construyen las herramientas. Yo escribo desde otro lado —desde este humilde rincón del mundo— con la certeza de que el criterio y la experiencia no son patrimonio de ningún hemisferio. Lo que aprendimos resolviendo con recursos acotados, mirando el costo de cada decisión, sirve. Y tiene algo para aportarle a la conversación global.

No pretendo tener la última palabra. Pretendo, sí, dejar por escrito lo que funcionó, lo que salió mal, y las preguntas que todavía no tienen respuesta cerrada. Si al terminar el libro seguís siendo escéptico en algunos puntos, habré cumplido igual: prefiero un lector que discute a uno que asiente. Lo único que pido es que la discusión parta de haber probado.

Empecemos.

Cómo leer este libro

Este es un libro de referencia tanto como de lectura corrida. Está pensado para que puedas leerlo de principio a fin una vez y después volver a capítulos sueltos cuando los necesites.

A quién está dirigido. A desarrolladores que ya conocen el oficio —que escribieron sistemas de verdad, que tomaron decisiones de arquitectura y las pagaron, que saben leer código ajeno— y que todavía no incorporaron el vibe coding a su forma de trabajar. Si lo mirás de reojo, si probaste una vez y no le encontraste la vuelta, o si decidiste esperar a ver, este libro es para vos: un puente entre el oficio que ya tenés y una forma de trabajar que se está volviendo el estándar. No es una introducción al vibe coding —para eso está el volumen anterior, que se resume en las próximas páginas—; es el volumen de ingeniería, el que se mete en el cómo, escrito para ayudarte a aggiornarte sin resignar nada de lo que sabés.

Cómo está armado cada capítulo. Todos siguen la misma estructura: abren con un **epígrafe** de alguien que pensó bien el oficio; desarrollan la idea en prosa; y cierran con un **Taller**, la parte aplicada, con prompts, plantillas o ejemplos concretos que podés copiar y adaptar. Cada capítulo trae al menos una **figura**. Todas las citas de los epígrafes están verificadas y con su fuente completa en la Sección de citas del final.

Convenciones. Hablo en primera persona y en un registro rioplatense, porque este libro es una opinión informada, no un manual neutro. Cuando digo “el agente” me refiero, de forma genérica, a cualquier asistente de código basado en modelos de lenguaje; evito nombres de productos a propósito, porque envejecen rápido y porque las ideas valen para todos. Los términos técnicos propios del libro están reunidos en el Glosario.

Itinerarios de lectura. No todos vienen a lo mismo:

- *Si sos un desarrollador que quiere trabajar mejor ya:* Partes I y II, y volvé a los Talleres. El Kit del director te resume las plantillas.

- *Si liderás un equipo:* después de la Parte I, saltá a la Parte V (equipo y proceso) y a la Parte VI (el factor humano, la carrera). Las preguntas de discusión al final sirven para trabajarlas con el equipo.
- *Si venís desconfiando:* empezá por "Cuándo no usar vibe coding" y "El caso del escéptico", en la Parte I. Si esos dos capítulos no te expulsan, el resto te va a resultar honesto.
- *Si querés lo práctico primero:* los dos casos de estudio (uno construye algo nuevo, el otro transforma código heredado) muestran el método entero en acción; después volvé a la teoría que los sostiene.

El libro anterior, en dos páginas

Este volumen asume el anterior, pero no lo exige. Acá va lo esencial, para que puedas leer este solo.

Qué es el vibe coding. Es la práctica de construir software describiéndole a un agente —en lenguaje natural— qué se quiere, y dejar que el agente produzca el código, en un ida y vuelta conversacional. El nombre, algo informal, describe una experiencia real: el desarrollador deja de teclear cada línea y pasa a expresar intención, ver qué vuelve, y ajustar. No es autocompletado sofisticado; es delegar la escritura y quedarse con la dirección.

Por qué apareció ahora. Porque los modelos de lenguaje cruzaron un umbral: dejaron de completar líneas sueltas y empezaron a producir funciones, módulos y sistemas enteros coherentes, y a usar herramientas —leer archivos, correr comandos, buscar— que los convierten en agentes capaces de actuar, no solo de sugerir. Ese salto cambió la economía del desarrollo: la parte más visible y tediosa del trabajo, escribir la representación, se abarató de manera brutal.

Qué sostenía el primer libro. Tres cosas. Primero, que esto no es una moda: la capacidad de traducir intención en código funcionando llegó para quedarse. Segundo, que baja la barrera de entrada: mucha más gente puede hoy convertir una idea en algo que corre. Y tercero, que eso no elimina la dificultad de fondo —construir buen software sigue siendo difícil—, sino que la corre de lugar. El primer libro era, sobre todo, un libro de fundamentos y de entusiasmo con los pies en la tierra, escrito para un público amplio.

Dónde termina aquel y empieza este. El volumen introductorio celebra —con razón— que la representación se haya abaratado. Este volumen mira la otra cara: cuando escribir código deja de ser el cuello de botella, todo el peso cae sobre lo que la herramienta no hace —definir bien, decidir la arquitectura, revisar con criterio, responsabilizarse—, y eso exige más oficio, no menos. Si el primero abrió la puerta, este se ocupa de lo que hay que saber para cruzarla sin romper nada del otro lado.

Capítulo 1. Del vibe coding conceptual al vibe coding de ingeniería

> «Creo que la parte difícil de construir software es la especificación, el diseño y la prueba de este constructo conceptual, y no el trabajo de representarlo ni de comprobar la fidelidad de esa representación.»
>
> — Fred Brooks, *No Silver Bullet: Essence and Accidents of Software Engineering* (1987)

El volante cambia de manos

El primer libro se ocupó de una pregunta anterior: qué es esto, por qué apareció y por qué no es una moda pasajera. Fue, deliberadamente, un libro de fundamentos, escrito para que cualquiera —viniera o no de la programación— pudiera entender de qué hablamos cuando hablamos de vibe coding. Este libro empieza donde aquel terminó, y da por saldada esa discusión. Acá el lector es otro: alguien que conoce el oficio, que escribió sistemas de verdad, que sabe lo que cuesta una decisión de arquitectura mal tomada. Para esa persona, la pregunta interesante ya no es *qué es*, sino *qué cambia en mi trabajo cuando el volante lo tengo yo*.

La respuesta corta, la que vamos a desarmar durante todo el capítulo, es que cambia menos de lo que el marketing promete y más de lo que el escéptico admite. No cambia la naturaleza del problema de construir software. Cambia dónde ponemos las manos y, sobre todo, dónde se concentra el valor de lo que sabemos hacer.

Para verlo con claridad conviene volver casi cuarenta años atrás, a un texto que la mayoría de nosotros leyó alguna vez y que hoy se lee distinto.

Esencia y accidente

En 1987, Fred Brooks publicó *No Silver Bullet*. Su tesis era pesimista y envejeció asombrosamente bien: no existe una única técnica, lenguaje o herramienta que, por sí sola, produzca un salto de un orden de magnitud en la productividad del desarrollo de software. La razón la dio con una distinción que sigue siendo la herramienta

conceptual más útil para pensar lo que nos está pasando: la diferencia entre dificultades **esenciales** y dificultades **accidentales**.

Las dificultades esenciales son inherentes a la naturaleza del software: entender qué hay que construir, modelar un dominio complejo, diseñar las estructuras y sus relaciones, razonar sobre casos y estados, especificar el comportamiento correcto, probar que el sistema hace lo que debe. Ese trabajo —el del *constructo conceptual*, como lo llamaba Brooks— es la esencia. Es difícil porque el problema es difícil, y ninguna herramienta lo vuelve trivial.

Las dificultades accidentales son las que nos agregamos nosotros en el camino de representar ese constructo: la sintaxis del lenguaje, el boilerplate, la gestión de memoria, la configuración del entorno, la traducción tediosa de una idea clara a las líneas que la máquina entiende. Brooks observó que buena parte de la historia del progreso en software —de los lenguajes de alto nivel a los entornos integrados— fue una guerra ganada contra lo accidental. Y advirtió el límite: por más que redujéramos lo accidental a cero, la esencia seguiría ahí, intacta, marcando el techo.

Guardá esa imagen, porque es la llave de todo el libro. Escribir el código —traducir una idea ya entendida a un lenguaje— es, en el marco de Brooks, trabajo predominantemente accidental. Decidir qué construir, cómo estructurarlo y cómo probar que está bien es la esencia.

Qué automatiza, en realidad, el vibe coding

Acá viene el giro. Cuando un agente genera código a partir de una descripción, lo que ataca —con una eficacia que Brooks no habría imaginado— es precisamente la dificultad accidental. La traducción de intención a representación, esa que nos consumía la mayor parte de las horas visibles del día, se comprime de manera brutal. Lo que antes era una tarde de teclear estructuras conocidas hoy es un párrafo bien escrito y una revisión.

Es tentador leer eso como la refutación de Brooks: al fin, la bala de plata. Es exactamente al revés. El vibe coding es la confirmación más literal de su tesis. Aplastó lo accidental casi hasta el piso, y al hacerlo dejó a la vista, más desnuda que nunca, la esencia. Cuando el costo de representar se vuelve casi cero, lo único que separa un buen sistema

de un desastre es la calidad del constructo conceptual: qué pediste, cómo lo pensaste, cómo verificaste que estuviera bien. La herramienta no toca esa parte. La expone.

Esta es la diferencia entre el vibe coding conceptual —el del primer libro, el que celebra que "ahora cualquiera puede hacer una app"— y el vibe coding de ingeniería, que es del que se ocupa este. El primero mira lo accidental y ve una liberación. El segundo mira la esencia y entiende que la vara, lejos de bajar, subió.

Por qué la vara sube, no baja

Cuando el trabajo accidental era caro, actuaba como un amortiguador involuntario. Escribir todo a mano era lento, y esa lentitud nos obligaba a pensar mientras tecleábamos; nos daba tiempo de detectar, casi por accidente, que una idea no cerraba. El costo de representar era también un filtro de calidad de la concepción.

Ese amortiguador desapareció. Hoy una idea mal pensada se materializa en un sistema que corre —que compila, que pasa una demo— con la misma velocidad que una idea buena. El error ya no se paga en la etapa de escritura; se paga después, más lejos y más caro, cuando el sistema está en producción y nadie entiende bien por qué hace lo que hace. La generación veloz no perdona la concepción floja: la amplifica.

Por eso sostengo que estas herramientas son más peligrosas, no menos, en manos de quien no domina el oficio. Al que no sabe especificar, el agente le entrega con enorme convicción exactamente lo que pidió mal. Al que no sabe leer código, le devuelve algo que parece correcto y que él no puede auditar. La destreza técnica no se volvió opcional; se volvió el factor que decide si toda esta velocidad juega a favor o en contra. Lo que cambió es su punto de aplicación: menos en producir la representación, más en dominar la esencia —especificar con precisión, revisar con criterio, responsabilizarse por el resultado.

El malentendido de la democratización

Conviene desactivar acá un eslogan que hace ruido en las dos veredas. Se dice que el vibe coding "democratiza" la programación, y de esa frase el entusiasta saca que el conocimiento ya no importa y el

escéptico saca que se viene una inundación de basura. Los dos leen mal.

Lo que se democratizó es el acceso a la representación: mucha más gente puede hoy convertir una idea en algo que corre. Eso es real y es valioso. Pero el acceso a la representación no es lo mismo que el dominio de la esencia. Se abarató producir código; no se abarató producir buen software. La brecha, de hecho, se corrió: antes separaba a quien podía escribir de quien no; ahora separa a quien puede *dirigir y juzgar* de quien solo puede *pedir*. Y esa segunda brecha es más difícil de cruzar, porque no se salva con una herramienta: se salva con oficio.

Para un desarrollador con experiencia, esto no es una amenaza a la identidad. Es una revalorización de lo que ya sabía y que a veces quedaba sepultado bajo el trabajo mecánico. El criterio arquitectónico, la intuición para oler un mal diseño, la disciplina de la prueba: todo eso, que era difícil de exhibir cuando el 80% del tiempo se iba en teclear, hoy es el centro de la escena.

De la anécdota a la ingeniería

Hay una distancia enorme entre hacer algo impresionante una vez y hacerlo de manera confiable, en equipo, sobre un sistema que tiene que durar. Esa distancia es, palabra por palabra, la que separa a un hobby de una disciplina de ingeniería. La mayor parte de lo que se escribe hoy sobre vibe coding vive del lado de la anécdota: el fin de semana en que alguien levantó un producto entero, la demo que dejó a todos con la boca abierta. Nada de eso es mentira. Pero tampoco es ingeniería todavía.

Tratar esto en serio significa hacerse las preguntas que nos hacemos con cualquier otra tecnología que entra a un sistema productivo. ¿Cómo especifico lo que quiero de manera que sea verificable? ¿Cómo reviso código que no escribí línea por línea? ¿Cómo evito que la velocidad se transforme en deuda técnica impagable? ¿Quién responde cuando eso que generó un agente falla a las tres de la mañana? ¿Cómo trabaja un equipo cuando varias personas dirigen agentes sobre el mismo repositorio? Ninguna de esas preguntas tiene una respuesta cómoda, y todas son el temario real de este libro. Este capítulo solo instala la premisa: si el trabajo accidental se abarató,

entonces la ingeniería —la esencial— es lo que queda por hacer, y hay que hacerla con método.

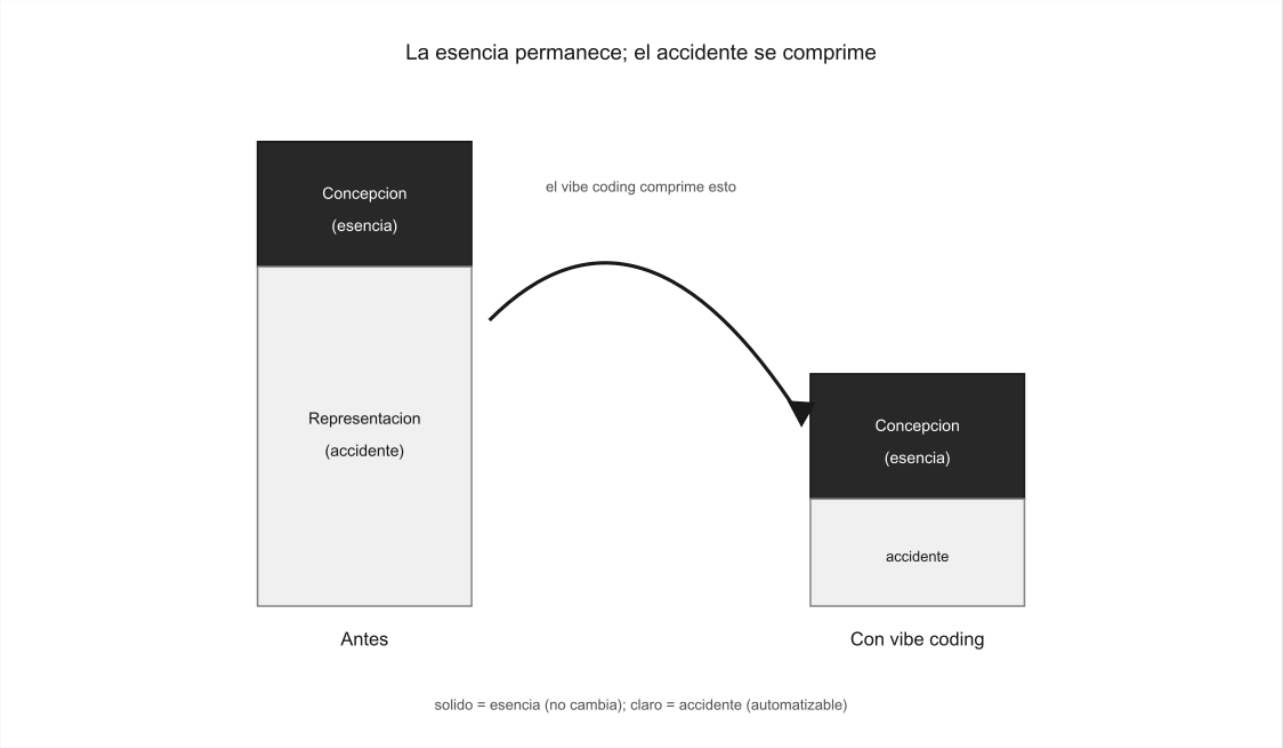
Un modelo de trabajo mínimo

Para no quedarnos en lo abstracto, fijemos desde ya un esqueleto que vamos a refinar a lo largo del libro. No es una metodología cerrada ni depende de ninguna herramienta en particular; es la forma mínima que toma el trabajo cuando uno se lo toma en serio.

Primero, **intención**: antes de pedir nada, escribir qué se quiere y por qué, con las restricciones y los criterios que permitirán decir si está bien. Es el constructo conceptual de Brooks, puesto por escrito. Segundo, **generación**: delegar la representación al agente, en pasos lo bastante chicos como para poder seguirlos. Tercero, **revisión**: leer lo que volvió no solo buscando errores de implementación, sino confrontándolo contra la intención original —¿resuelve el problema correcto?—. Cuarto, **verificación**: pruebas que actúen como contrato, no como decorado. Y quinto, **responsabilidad**: alguien humano firma, integra y responde por el resultado.

Ese ciclo —intención, generación, revisión, verificación, responsabilidad— es la columna vertebral. Todo lo que sigue en el libro es, en el fondo, este esqueleto mirado con lupa desde distintos ángulos: el prompting como especificación, la revisión cuando la escribe un agente, la arquitectura, el equipo. Si te llevás una sola idea de este capítulo, que sea esta: el vibe coding no reemplazó al ingeniero, le sacó de encima el trabajo accidental para dejarlo cara a cara, sin excusas, con el esencial.

(Ver Figura 1.1: dónde ataca el vibe coding en el marco de esencia y accidente.)



El costo del error se mudó de lugar

Hay una consecuencia de todo esto que conviene mirar de frente, porque es la que más plata cuesta cuando no se la ve. En el mundo viejo, el costo de un error estaba distribuido a lo largo de la escritura. Vos tecleabas, y en ese tecleo lento el error tenía mil oportunidades de delatarse: la firma que no cerraba, el tipo que no compilaba, la estructura que se resistía a ser escrita porque en el fondo estaba mal pensada. La representación cara funcionaba como una malla de contención que atrapaba, temprano y barato, buena parte de los errores de concepción. No lo hacía a propósito; lo hacía como efecto secundario de su propia lentitud.

Cuando la representación se abarata a casi cero, esa malla desaparece, y el error no desaparece con ella: se corre hacia adelante. Deja de pagarse en la escritura y pasa a pagarse en la integración, en la revisión, en la prueba y —si nada de eso lo atrapa— en producción. Y acá aparece la ley económica dura del oficio nuevo: cuanto más tarde se detecta un error, más caro sale arreglarlo, porque para entonces ya tiene código construido encima, decisiones que dependen de él, y a veces datos reales contaminados. El vibe coding no cambió esa ley; la volvió más urgente, porque aceleró

justamente la etapa que antes hacía de filtro y dejó intactas las etapas caras de más adelante.

De ahí se desprende algo que a muchos les cuesta aceptar: la velocidad de generación no es, por sí sola, velocidad de entrega. Podés llegar al primer borrador diez veces más rápido y no llegar más rápido al software terminado, si ese borrador arrastra decisiones flojas que vas a pagar en revisión y en mantenimiento. La única forma de que la aceleración sea real es reconstruir, a mano y a conciencia, la malla de contención que la lentitud te daba gratis: los criterios de aceptación escritos antes, la revisión que confronta contra la intención, la prueba que actúa como contrato. Sin esa malla, el vibe coding no te hace más rápido; te hace más rápido en acumular deuda.

Dónde se concentra ahora el valor

Vale detenerse en la pregunta que más le importa a un profesional con experiencia: si el trabajo accidental se abarató, ¿en qué se vuelve valioso lo que sé? La respuesta es que el valor migra, palabra por palabra, hacia los extremos del ciclo. Al principio, en la capacidad de convertir un problema difuso en una especificación precisa y verificable: entender el dominio, anticipar los casos límite, decidir la arquitectura, poner por escrito qué significa "terminado". Y al final, en la capacidad de juzgar: leer lo que volvió, detectar lo que está plausible pero mal, responsabilizarse por integrarlo. El medio —la producción de la representación— es exactamente lo que se abarató, y por eso es lo que menos te distingue hoy.

Esto reordena en silencio el valor de una carrera. Durante años, buena parte de lo que exhibíamos como competencia vivía en el medio: escribir rápido, conocer de memoria la sintaxis, resolver el mecanismo. Eso sigue siendo útil como base —no se puede juzgar bien lo que no se sabría hacer—, pero dejó de ser lo que crea valor diferencial. Lo que crea valor es lo que siempre fue difícil de enseñar y difícil de medir: el criterio para especificar y el criterio para juzgar. El desarrollador que entiende esto deja de competir en velocidad de tecleo, donde la herramienta lo va a superar siempre, y compite donde la herramienta no llega: en la esencia de Brooks.

Ejemplo trabajado: la misma feature, dos concepciones

Dejame bajarlo a tierra con un caso que vi repetirse con variantes. Dos personas reciben el mismo pedido: agregar a una aplicación la posibilidad de que un usuario exporte sus datos. Las dos usan el mismo agente, la misma potencia de generación. La diferencia está entera en la concepción, es decir, en la esencia.

La primera abre el chat y escribe lo primero que se le viene:

```
Agregá un endpoint para que el usuario exporte sus datos a
un archivo.
```

El agente, obediente y literal, le devuelve en segundos algo que corre: un endpoint que junta las tablas del usuario, arma un archivo y lo devuelve. Compila, la demo funciona, el ticket se cierra. Parece un triunfo de la velocidad. El problema es todo lo que esa frase no dijo y el agente completó con lo más probable: ¿exporta también los datos de otros usuarios que aparecen mezclados en registros compartidos? ¿Incluye campos internos que nunca deberían salir del sistema? ¿Qué pasa si el usuario tiene millones de registros y el archivo entero se arma en memoria? ¿Queda algún rastro del archivo generado en un disco temporal? Ninguna de esas preguntas se hizo, así que ninguna se respondió. El sistema anda, y el error —una fuga de datos ajenos, una caída bajo carga— espera agazapado para pagarse más lejos y más caro.

La segunda persona no toca el agente todavía. Primero escribe la intención, y recién después la convierte en pedido:

```
Necesito una exportación de datos personales para un
usuario autenticado.

CONTEXTO: vive en el servicio de cuentas; NO debe acceder
a datos de
otros usuarios ni a campos marcados como internos.

RESTRICCIONES: debe funcionar con usuarios de hasta
millones de registros
sin cargar todo en memoria; el archivo temporal se borra
siempre, también
si la operación falla.
```

```
CRITERIOS DE ACEPTACIÓN:
- Solo incluye datos cuyo dueño es el usuario que pide.
- Excluye la lista de campos internos [enumerada].
- Bajo carga alta, memoria acotada (streaming).
- No deja archivos residuales bajo ningún camino de error.
CASOS LÍMITE: usuario sin datos; operación interrumpida a
la mitad;
registros compartidos con otros usuarios.

FUERA DE ALCANCE: exportación programada o recurrente.

Antes de escribir código, devolveme tu plan y los
supuestos que hacés.
```

El agente tarda lo mismo en generar. Pero ahora genera contra una especificación que ya pensó en las trampas, y la revisión tiene contra qué confrontar: cada criterio de aceptación es una pregunta concreta que hacerle al código que volvió. Lo notable es que la segunda persona no escribió más código que la primera; escribió más *pensamiento* antes del código. Invirtió su esfuerzo donde el vibe coding no la reemplaza —en la esencia— y dejó que la herramienta se ocupara de lo que sí sabe hacer mejor: la representación. Ese es, en una escena, todo el capítulo. La velocidad de las dos fue idéntica en el papel; la distancia entre las dos se va a medir dentro de seis meses, cuando una de las dos exportaciones aparezca en un informe de incidentes y la otra siga funcionando sin que nadie se acuerde de ella.

Representar para pensar, no solo para entregar

Hay un uso del abaratamiento de lo accidental que casi nadie aprovecha, y que a mí me parece el más interesante de todos, porque da vuelta la relación entre representación y concepción que venimos discutiendo. Hasta acá tratamos la generación como el paso que viene después de pensar: primero fijás la intención, después delegás la representación. Es el orden correcto cuando ya sabés qué querés. Pero hay otro orden, igual de legítimo y distinto, para cuando todavía no lo sabés: usar la representación barata como instrumento para descubrir la esencia que no ves.

Cuando representar era caro, no te lo podías permitir. Construir algo era el entregable, así que pensabas primero justamente porque una construcción equivocada costaba días. La lentitud te forzaba al diseño previo, y estaba bien, pero también te cerraba una puerta: la de aprender construyendo. Hoy esa puerta se abrió. Podés generar una cosa entera, mirarla, y tirarla, no porque haya salido mal, sino porque su único propósito era enseñarte algo sobre el problema que no ibas a ver en abstracto. La representación deja de ser solo el destino y pasa a poder ser, también, un telescopio para mirar la esencia.

Acá conviene una distinción que separa el uso maduro del uso ingenuo, y es la misma que en otros oficios separa el boceto de la obra: hay generación para descubrir y hay generación para conservar. La generación para descubrir —lo que en el oficio viejo llamábamos un *spike*, una prueba de concepto sabiendo de antemano que la vas a descartar— tiene una regla sagrada: se borra. No importa que haya quedado prolija, que compile, que la demo funcione. Su valor no está en el código que produjo, sino en las preguntas esenciales que sacó a la superficie. El pecado, el que hunde a mucha gente entusiasta, es dejar que el spike sobreviva y se vuelva el cimiento del sistema solo porque estaba ahí y andaba. Lo que se descubrió construyendo hay que reescribirlo en la intención, y recién entonces construir de nuevo, en serio, contra esa intención afilada.

Por qué esto ataca la esencia y no lo accidental es lo que lo vuelve un tema de este capítulo y no una simple técnica de productividad. Cuando el problema es difuso —no sabés bien cómo modelar un dominio, no sabés qué estructura va a resistir mejor los cambios que vienen, no sabés qué casos límite esconde una interacción que nunca hiciste— pensar en abstracto tiene un techo. Hay preguntas esenciales que no aparecen hasta que las cosas se tocan entre sí, y las cosas se tocan cuando existen, aunque existan para ser tiradas. La representación barata te deja materializar tres versiones distintas de una idea en el tiempo que antes te llevaba dudar sobre una sola, y verlas al lado te revela cuál era la pregunta esencial que ninguna de las tres respondía bien.

Ejemplo trabajado: tres bocetos para encontrar la pregunta

Bajemos esto a tierra con un caso donde el orden "pensar y después generar" se traba, porque el que tiene que pensar todavía no sabe contra qué. Tenés que agregar a un sistema la posibilidad de notificar a los usuarios cuando pasan ciertas cosas —un pago que se acredita, un pedido que cambia de estado, un límite que se está por alcanzar—. En abstracto, la pregunta se siente resuelta: "mando una notificación cuando pasa el evento". Pero esa frase esconde una decisión de modelado que no vas a ver hasta que la toques: ¿el sistema que genera el evento sabe *cómo* notificar, o solo *que algo pasó* y otro se encarga de decidir a quién y por qué canal? Esa es una pregunta esencial —define acoplamientos que vas a arrastrar por años— y en el papel se siente prematura.

En vez de agonizar en abstracto, usás la representación barata como sonda. No pedís "hacé el sistema de notificaciones". Pedís bocetos desechables, y aclarás que son desechables:

```
Necesito EXPLORAR, no construir. Generá tres versiones
mínimas y
desechables de cómo un pago acreditado dispara una
notificación al
usuario, cada una con un acoplamiento distinto:

A) el servicio de pagos llama directo al de
notificaciones.
B) el servicio de pagos publica un evento
"pago.acreditado" y un
   suscriptor decide qué notificar.
C) un proceso aparte lee los pagos nuevos y notifica, sin
que pagos
   sepa que las notificaciones existen.

Para cada una, mostrame solo el esqueleto y decime: qué
pasa si mañana
agrego un segundo canal (SMS), qué pasa si notificar
falla, y quién se
entera del error. NO estoy eligiendo todavía. Voy a tirar
las tres.
```

Lo que volvió no te importa como código; te importa como espejo. Al ver las tres al lado, la pregunta esencial que estaba escondida

aparece sola: en la versión A, agregar un segundo canal obliga a tocar el servicio de pagos, que no tiene por qué enterarse de que existen los canales; en la versión A, además, si notificar falla, el pago queda en un limbo raro. Recién ahí entendés que la decisión no era "cómo notifico" sino "cuánto tiene que saber el que origina el evento sobre lo que pasa después", y que la respuesta correcta para tu caso —donde los canales van a multiplicarse y una notificación caída no puede arrastrar al pago— está entre la B y la C. Esa comprensión no la tenías antes de generar, y no la ibas a tener rumiándola en la cabeza.

El paso que separa esto de un desastre es el que casi nadie da: cerrás la exploración, borrás los tres bocetos, y escribís en la intención lo que aprendiste —"el origen del evento no conoce los canales; la entrega es asíncrona y su fallo no afecta al evento origen"—. Recién con esa restricción esencial ya descubierta, arrancás la construcción en serio, con el ciclo de siempre. Fijate el movimiento completo: usaste lo accidental barato para iluminar una decisión esencial que no se dejaba pensar en seco, pero no dejaste que lo accidental se quedara a vivir como si fuera esencial. El agente no pensó por vos; te construyó, rápido y gratis, los tres muñecos contra los cuales por fin pudiste pensar. Esa es una forma de trabajar que el oficio viejo, por el precio de la representación, sencillamente no se podía permitir.

Taller

La parte práctica de este capítulo no es escribir código, sino lo que viene antes: ejercitar la disciplina de la intención. Dos herramientas concretas.

1. Plantilla de intención (el constructo conceptual, por escrito)

Antes de pedirle una sola línea a un agente, completá esta plantilla. La regla es simple: si no podés llenarla, todavía no estás listo para generar.

```
OBJETIVO
  Qué problema resuelve esto, en una frase. Para quién.

CONTEXTO
```

```
  Dónde vive (módulo, servicio, capa). Con qué interactúa.
  Qué NO debe tocar.

RESTRICCIONES
  Técnicas (rendimiento, compatibilidad, dependencias
permitidas).
  De negocio o de dominio (reglas que no se negocian).

CRITERIOS DE ACEPTACIÓN
  Lista verificable de "esto está bien si...".
  Casos límite conocidos que deben quedar cubiertos.

FUERA DE ALCANCE
  Lo que explícitamente no se pide, para que el agente no
invente.
```

Un prompt de generación bien formado no es más que esta plantilla convertida en pedido:

```
Necesito [OBJETIVO]. Vive en [CONTEXTO] y no debe tocar
[X].
Respetá estas restricciones: [...]. Consideralo terminado
solo si
cumple estos criterios: [...]. No implementes nada de
[FUERA DE ALCANCE].
Antes de escribir código, devolveme tu plan y los
supuestos que estás
haciendo, para que los confirme.
```

Esa última línea —pedir el plan y los supuestos antes del código— es la que convierte la generación en un acto de ingeniería y no de fe.

2. Reglas de la casa (infraestructura de intención en el repositorio)

La intención no debería reescribirse en cada prompt: buena parte es estable y pertenece al proyecto. Conviene versionarla junto al código, en un archivo de convenciones que el equipo mantiene y que sirve de contexto permanente. Un esqueleto mínimo, en texto plano, agnóstico de herramienta:

```
# Convenciones del proyecto

## Arquitectura
- Estilo y límites: [p. ej. capas, qué puede depender de
```

```
qué].
- Decisiones que NO se revierten sin discusión (enlazar a
los registros
  de decisión de arquitectura).

## Estándares de código
- Convenciones de nombres, formato, manejo de errores.
- Bibliotecas permitidas y bibliotecas prohibidas (y por
qué).

## Pruebas
- Qué se prueba siempre. Qué cuenta como cobertura
suficiente.

## Reglas para trabajo asistido
- Nunca exponer secretos ni datos sensibles en los
pedidos.
- Todo cambio generado se integra vía revisión humana.
- Cada cambio deja registro de su intención (spec) y sus
supuestos.
```

Este archivo es deliberadamente aburrido, y esa es la idea: es el lugar donde el criterio del equipo deja de vivir en la cabeza de cada uno y se vuelve infraestructura compartida. Volveremos sobre él —bastante— en la Parte V.

Epígrafe: la cita de Brooks se consigna con su fuente completa en la Sección de citas.

Capítulo 2. Un nuevo modelo mental: el desarrollador como director técnico

> «Las herramientas que usamos tienen una influencia profunda (¡y solapada!) sobre nuestros hábitos de pensamiento y, por lo tanto, sobre nuestra capacidad de pensar.»
>
> — Edsger W. Dijkstra, *How do we tell truths that might hurt?* (EWD498, 1975)

La herramienta que reordena la cabeza

Dijkstra escribió esa frase para advertir sobre los lenguajes de programación, pero describe con una precisión incómoda lo que nos pasa hoy. Una herramienta nueva no se limita a hacer más rápido lo que ya hacíamos: reordena, sin pedir permiso, la forma en que pensamos el trabajo. Y ahí está el riesgo silencioso del vibe coding. No es que la herramienta sea mala; es que si la operamos con el modelo mental viejo —el del programador que produce cada línea— la vamos a usar mal, y encima no nos vamos a dar cuenta.

El capítulo anterior estableció *qué* cambió: el trabajo accidental se abarató y quedó expuesta la esencia. Este capítulo se ocupa de *cómo* tenemos que cambiar nosotros para trabajar en ese mundo. La tesis es simple de enunciar y difícil de habitar: el desarrollador deja de ser, principalmente, quien escribe el código, y pasa a ser quien lo dirige. De autor a director técnico. No es un ascenso ni una degradación; es un cambio de oficio dentro del mismo oficio.

Qué significa dirigir

Conviene desconfiar de las metáforas fáciles, así que fijemos una con cuidado. Un director técnico no es un gerente que dejó de saber. Es más parecido a un director de orquesta o al responsable técnico de un rodaje: alguien que no toca todos los instrumentos ni opera todas las cámaras, pero que entiende cada oficio lo suficiente como para pedir con precisión, para escuchar cuando algo desafina y para hacerse responsable del resultado final. La distinción es la clave de todo el capítulo: dirigir no es dejar de saber, es aplicar lo que se sabe en otro punto de la cadena.