CONSTRUIR CON IA, ENTENDIENDO QUÉ HACES

Arquitectura para vibe coders

Entiende lo que la IA construye por ti
— antes de que se rompa.

ESCRITO POR

Diógenes Alberto Moreira

Arquitectura para Vibe-Coders

Entiende lo que la IA construye por ti — antes de que se rompa

Diógenes Alberto Moreira

Índice

Agradecimientos

A Lucrecia, mi mujer, y a mis hijos Ismael y Luca: por soportar todo el tiempo que paso encerrado en la oficina, y por ese café que aparece en el escritorio cuando la noche se hace larga. Este libro se escribió en horas que eran de ustedes.

A Fabián Gómez, que fue uno de los disparadores de este libro — a veces una conversación empuja más que mil planes.

A Emiliano Arango, mi amigo, maestro y alumno a la vez, con quien compartimos tantas aventuras tecnológicas que algunas terminaron, disfrazadas, en estas páginas.

A Andrés Bilbao, a Dylan Rosemberg y a todo el equipo de 30X, que sin saberlo me dieron material para este proceso en la observación del día a día.

Una frase para Abel Moreira, mi papá, y Axel Moreira, mi abuelo, que me miran desde el cielo: esto también es de ustedes.

Y a mi mamá, Amelia Bernalina Gómez, por todo lo que no cabe en una lista de agradecimientos.

A mis abuelas, que sembraron mucho antes de que yo entendiera qué se estaba sembrando.

Introducción — El poder sin el mapa

Comienzo la tarea de escribir este libro con una convicción: el vibe coding llegó para quedarse.

A diferencia de otras modas pasajeras que he visto desfilar por esta industria —y he visto muchas—, esta no es una promesa: es un hecho consumado. La potencia de la inteligencia artificial generativa abrió las puertas del desarrollo de software a gente sin capacitación formal. Diseñadores que publican sus propias herramientas. Emprendedores que construyen su MVP en un fin de semana. Contadores, médicos, abogados que automatizan lo que antes requería contratar a un equipo. Personas que hace dos años no sabían qué era una terminal hoy tienen aplicaciones funcionando en internet.

Y sin embargo —por más que parezca un oxímoron— esa misma gente carece del conocimiento necesario para complementar la asistencia de los LLM.

No es una crítica. Es una descripción del terreno. El modelo escribe el código, y lo escribe bien. Lo que no puede hacer es decidir por ti: no sabe si tu app va a tener diez usuarios o diez mil, no sabe cuánto puedes gastar en infraestructura, no sabe que ese prototipo "de prueba" se va a convertir en el sistema del que depende tu negocio. La IA ejecuta; el criterio sigue siendo tuyo. Y el criterio, en software, tiene un nombre: arquitectura.

Por eso la historia se repite con una precisión casi cómica. El vibe-coder avanza rápido, mucho más rápido de lo que cualquier programador tradicional hubiera soñado. La demo funciona. Los

primeros usuarios llegan. Todo fluye… hasta que se choca con la pared.

La pared tiene muchas caras. A veces es una base de datos que se corrompe porque nadie pensó en las migraciones. A veces es una llave de API expuesta que amanece convertida en una factura de miles de dólares. A veces es más silenciosa: el proyecto crece hasta que ni el propio modelo entiende el código que escribió, y cada cambio rompe dos cosas nuevas. El síntoma varía; la causa es siempre la misma. No falló el código. Faltó el mapa.

Este libro existe para zanjar esa diferencia.

No vengo a darte clases de programación; para eso ya tienes a la IA, y es mejor que yo. Vengo a compartir contigo lo que la IA no puede darte: el entendimiento de qué estás construyendo, de qué piezas se compone, de cómo se conectan y de dónde se rompen. El vocabulario y el criterio para dejar de ser un pasajero de tu propio proyecto y pasar a ser el arquitecto que le indica al constructor —incansable, obediente, brillante y ciego— qué edificio levantar.

El recorrido es el de tu propia app. Primero el mapa: qué es la arquitectura y cuál es la anatomía de cualquier aplicación. Después las piezas: frontend, backend, bases de datos, usuarios, el mundo exterior. Luego cómo se conecta todo: patrones, stacks, deploy. Y al final, lo que nadie te cuenta hasta que duele: seguridad, costos, deuda técnica, y cómo hablarle a la IA como arquitecto y no como turista.

Cada capítulo combina dos registros. Primero la narrativa: conceptos explicados con analogías, historias reales y cero jerga innecesaria. Después la práctica: ejercicios concretos para aplicar

lo aprendido en tu propio proyecto, con tu propia IA. Porque este libro no se lee: se usa.

Una confesión, también, antes de empezar: en este libro vamos a tomar licencias poéticas. El contenido técnico se digiere mejor con analogías que con definiciones, y las analogías siempre son imperfectas: la base de datos no es exactamente una despensa, el backend no es exactamente una cocina, y más de un especialista va a fruncir el ceño —con razón— ante alguna de mis simplificaciones. Lo asumo de antemano: nos van a criticar por algunas cosas, y está bien. Este libro elige, cada vez que hay que elegir, la claridad por sobre la precisión del manual. Si al terminarlo quieres el rigor completo, vas a estar en condiciones inmejorables de ir a buscarlo — y esa es exactamente la idea.

Una advertencia honesta antes de empezar. Saber arquitectura no te va a impedir cometer errores; te va a permitir cometerlos más barato, detectarlos más temprano y corregirlos sin demoler el edificio. Esa es la diferencia entre el que avanza hasta chocar con la pared y el que llega a producción sin morir en el intento.

Ese camino es el que vamos a transitar juntos.

Capítulo 1 — ¿Qué es vibe-coding y por qué funciona (hasta que no)?

Estructura del capítulo

- 1.1 La nueva forma de construir software
- 1.2 Historias de éxito y de desastre
- 1.3 El momento exacto en que "no saber" se vuelve caro
- Práctica: el inventario, la entrevista y el mapa de paredes

⁂

1.1 La nueva forma de construir software

El término lo acuñó Andrej Karpathy, uno de los fundadores de OpenAI, en un tuit de febrero de 2025 que se volvió profecía: "Hay un nuevo tipo de programación que llamo vibe coding, donde te entregas por completo a las vibras, abrazas los exponenciales y te olvidas de que el código existe." Lo escribió medio en broma, como quien le pone nombre a un vicio privado. El tuit superó los cuatro millones y medio de vistas en días, y la industria —que llevaba meses haciendo exactamente eso sin atreverse a confesarlo— se lo tomó completamente en serio. En cuestión de semanas, "vibe coding" dejó de ser un chiste de programadores para convertirse en categoría: aparecía en titulares de Forbes y del New York Times, en ofertas de empleo, en presentaciones de inversores. Los diccionarios empezaron a seguirle la pista. Pocas veces una palabra nueva encontró tan rápido a los millones de personas que la estaban esperando.

Porque describía algo que ya estaba pasando. Millones de personas habían empezado a construir software mediante una conversación. La mecánica es tan simple que cuesta creer que funcione: describes lo que quieres en tu propio idioma —"quiero una app donde mis clientes reserven turnos y reciban un recordatorio por WhatsApp"—, la IA escribe el código, tú miras el resultado en pantalla y pides ajustes. "El botón más grande." "Que el recordatorio salga un día antes." "Ahora agrégale pagos." No lees el código. No lo entiendes. En muchos casos, ni siquiera lo ves: las herramientas modernas lo esconden detrás de una vista previa, como el motor bajo el capó de un auto. El ciclo es describir, generar, probar, corregir. Y funciona.

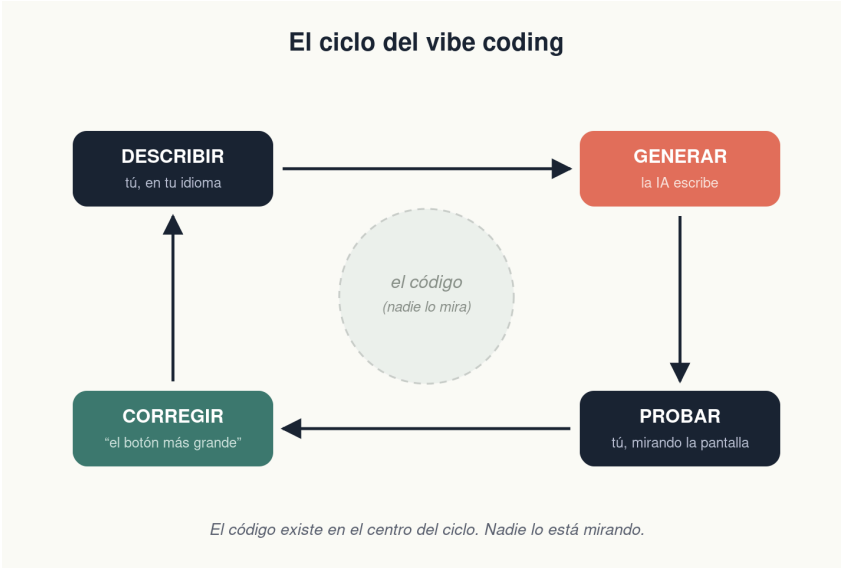

Figura 1.1 — El ciclo del vibe coding.

Alrededor de esa mecánica floreció, en tiempo récord, una industria entera. Cursor, un editor de código con la IA incrustada, se convirtió en el software de crecimiento más rápido

de su categoría en la historia. Lovable, Bolt y v0 fueron más lejos: eliminaron el editor por completo, dejando solo la conversación y la vista previa. Replit transformó su plataforma en un agente al que le encargas la app entera. Cada una eligió cuánto código mostrarte —de todo a nada—, pero todas apostaron a la misma tesis: que la próxima generación de creadores de software no va a querer verlo. Las cifras les dieron la razón más rápido de lo que ellas mismas esperaban; varias pasaron de cero a valuaciones de miles de millones en menos de dos años, arrastradas por una ola de usuarios que no venían de la programación: venían de tener una idea.

Conviene detenerse en lo radical del cambio, porque desde adentro de la ola es difícil medir la ola. Durante setenta años, la historia del software fue la historia de una barrera de entrada. Programar exigía años de estudio: lenguajes con sintaxis hostiles donde un punto y coma fuera de lugar detenía todo, teoría de estructuras de datos, algoritmos, y el folclore acumulado de una profesión entera —qué herramienta usar, qué error significa qué, a quién preguntarle—. La barrera filtraba a casi todos. Del lado de adentro quedaban los programadores, una de las profesiones mejor pagadas del planeta precisamente por lo alta que era la barrera. Del lado de afuera, el resto del mundo con sus ideas: el dentista que imaginaba un mejor sistema de turnos, la profesora que sabía exactamente qué app necesitaban sus alumnos, el comerciante que hacía a mano lo que una computadora haría en segundos. Ideas huérfanas de ejecución.

Cada tanto, la industria prometió derribar esa barrera, y la lista de promesas incumplidas es larga e instructiva. COBOL, en 1959, fue diseñado para que los gerentes escribieran sus propios

programas en algo parecido al inglés; terminó siendo territorio exclusivo de programadores, y los gerentes siguieron dictando memos. Visual Basic y las herramientas visuales de los noventa prometieron programar arrastrando botones con el mouse; sirvieron para prototipos y murieron de complejidad en cuanto el proyecto crecía. Dreamweaver y los constructores de sitios web pusieron páginas al alcance de todos, siempre que "todos" quisieran exactamente el tipo de página que la herramienta sabía hacer. Y las plataformas no-code de la década pasada —Bubble, Webflow, Airtable— llegaron más lejos que nadie, pero chocaron con el mismo techo de siempre: servían hasta que necesitabas algo que sus creadores no habían previsto. Entonces la barrera reaparecía, intacta, con un cartel que decía "a partir de aquí, contrate un programador".

La IA generativa es distinta, y esta es la convicción sobre la que se apoya este libro: no es una moda más de esa lista. La diferencia es estructural. Todas las herramientas anteriores te daban piezas prefabricadas —plantillas, bloques, componentes— y tu libertad terminaba donde terminaba el catálogo. El LLM no tiene catálogo: escribe código real, el mismo código que escribiría un programador, en cualquier dirección que tu idea lo lleve. Si puedes describirlo, puede intentar construirlo. No hay techo de plantillas que tocar. El límite ya no es lo que la herramienta previó, sino otra cosa mucho más sutil —y a esa otra cosa vamos a dedicarle el resto del libro.

Al que construye de esta manera lo llamamos vibe-coder, y conviene definirlo bien, porque no es "alguien que no sabe programar". Es alguien que sabe otra cosa. Sabe de su negocio, de su industria, de su problema. La diseñadora que construyó la

herramienta que sus colegas usan todos los días no sabe qué es una migración de base de datos, pero sabe exactamente qué necesita un equipo de diseño, qué flujo de trabajo les duele, qué pantalla van a abrir primero cada mañana. Ese conocimiento del dominio —que ningún LLM tiene sobre tu problema particular— es la mitad valiosa de la ecuación. La IA pone la sintaxis; el vibe-coder pone el sentido.

El espectro, además, es más amplio de lo que sugiere la caricatura. En un extremo está el emprendedor que jamás abrió una terminal y construye todo por conversación. En el otro, programadores veteranos que saben perfectamente leer el código y eligen no hacerlo, porque revisar cada línea mataría la velocidad que hace mágico al método. En el medio, una población creciente de híbridos: gente de producto, analistas, diseñadores, científicos de datos que saben "un poco" y estiran ese poco hasta el infinito con ayuda del modelo. Lo que todos comparten no es un nivel de conocimiento: es una relación nueva con el código, donde el código dejó de ser algo que se escribe y pasó a ser algo que se pide.

Y aquí aparece la asimetría que define esta nueva forma de construir. El programador tradicional escribía lento pero entendía todo lo que escribía; cuando algo fallaba a las tres de la mañana, tenía un mapa mental del sistema para ir a buscar la falla. El vibe-coder construye a una velocidad que hace dos años era ciencia ficción, pero entiende poco de lo que se construye en su nombre; cuando algo falla a las tres de la mañana, tiene una conversación con una IA que a veces tampoco lo entiende. Velocidad sin comprensión. Es un intercambio, aunque nadie te lo presenta como tal: nadie te avisa, mientras la demo funciona y

los amigos aplauden, que estás firmando ese contrato. Es parecido a manejar un auto sin saber nada de mecánica: perfectamente razonable en la ciudad, con talleres en cada esquina, y una decisión que se cobra distinto en medio del desierto. El problema del software es que producción —usuarios reales, datos reales, dinero real— casi siempre queda en el desierto.

Durante las primeras semanas de un proyecto, el intercambio es gloriosamente favorable. Todo lo que pides aparece. Cada sesión termina con más features que la anterior. La comprensión parece un lujo de puristas, un peaje que los ingenieros de antes pagaban porque no tenían alternativa; el precio de la ignorancia parece cero. Pero el precio existe, es acumulativo, y tiene la mala costumbre de cobrarse todo junto: el día que la app se cae con usuarios reales adentro, el día que llega una factura absurda de un servicio que no sabías que estabas usando, el día que el propio modelo ya no entiende el proyecto que escribió y cada arreglo rompe dos cosas nuevas. Las dos secciones que siguen muestran ese momento con historias reales —las de los que cruzaron la pared y las de los que quedaron estampados contra ella—.

Antes de seguir, un pedido: sepan entender que esta tecnología avanza tan rápido, y hay tantos cambios, que es probable que cuando estén leyendo este libro algunas cosas ya hayan sido soslayadas por la mejora o directamente no sean ciertas. Las herramientas que hoy nombro pueden haber cambiado de nombre, de dueño o de existencia; los incidentes que hoy son escándalo pueden haberse vuelto imposibles por diseño. No es un defecto del libro; es la naturaleza del terreno que estamos pisando. Por eso el foco está puesto en los conceptos —que

envejecen lento— y no en las herramientas, que envejecen en meses.

La buena noticia, y es la tesis de este libro, es que el intercambio no es obligatorio. Se puede tener la velocidad sin renunciar a la comprensión que importa. No hace falta aprender a programar; hace falta aprender a *entender*: saber qué piezas tiene lo que estás construyendo, cómo se conectan y dónde se rompen. Saber qué preguntas hacerle a la IA antes de aceptar lo que propone, y reconocer las decisiones que no deberías delegarle a nadie —ni a nada—. Eso es arquitectura. Es más chico que una carrera de ingeniería y más grande que un tutorial de fin de semana: es, exactamente, el contenido de este libro.

⁂

1.2 Historias de éxito y de desastre

Las promesas abstractas convencen poco. Veamos gente real.

Los que cruzaron

Sabrine Matos era growth marketer en Brasil. Cero experiencia en programación: su herramienta de trabajo era el marketing, no el código. Pero conocía de primera mano un problema que las apps de citas se negaban a resolver: el miedo —fundado, estadísticamente respaldado— de encontrarse con un desconocido que no es quien dice ser. En 45 días construyó con Lovable una app llamada Plinq que verifica antecedentes de posibles citas antes del primer encuentro. Hoy tiene más de diez mil usuarios y factura cerca de medio millón de dólares al año. Vale la pena subrayar la aritmética: 45 días separaron a una persona sin formación técnica de un producto funcionando con

ingresos reales. Con el método tradicional —aprender a programar primero, construir después— esos 45 días hubieran alcanzado apenas para las primeras lecciones de sintaxis. Sabrine no sabía qué era un endpoint; sabía, porque lo había vivido, qué miedo resuelve su producto. Esa asimetría —experta en el problema, analfabeta en la solución técnica— habría sido descalificante en 2020. En la era del vibe coding, resultó ser la combinación ganadora.

Nicola Manzini tenía tres mil trescientos seguidores en X —es decir, nadie— cuando publicó Vibe Sail, un simulador de navegación a vela que corre en el navegador, construido conversando con la IA. No había plan de negocios ni inversores: había un tipo al que le gustaba navegar y una herramienta que le permitió convertir ese gusto en un juego. El viento, las olas, la física de la vela: todo negociado por chat con un modelo. El juego encontró su público, llegó a generar ocho mil dólares mensuales y atrajo hasta a Atlantic Records como auspiciante. La discográfica de Led Zeppelin pautando en el velero digital de un desconocido: una frase imposible de escribir dos años antes.

Y el caso que se volvió leyenda corporativa: Base44, una plataforma para crear apps sin código construida —la ironía es deliberada— usando la propia IA como equipo de desarrollo. Su fundador la levantó prácticamente en soledad, documentando el crecimiento en público. Seis meses después de nacer, con menos de diez personas en el equipo, tenía trescientos mil usuarios y era rentable, algo que startups con decenas de ingenieros y millones de inversión no logran en años. Wix la compró por ochenta millones de dólares en efectivo. Seis meses. Ochenta millones.

Menos de diez personas. Los números del vibe coding, cuando sale bien, no se parecen a nada anterior.

El caso más ruidoso, sin embargo, vino de un veterano. Pieter Levels, programador holandés célebre en la comunidad indie por construir negocios enteros él solo, decidió probar el método: en unas tres horas de conversación con la IA armó un simulador de vuelo multijugador que corre en el navegador, fly.pieter.com. Gráficos rudimentarios, física dudosa, diversión inmediata. Diecisiete días después publicó el resultado en X: "Ha pasado de $0 a $1 millón de ARR en solo 17 días. Mi primer proyecto que crece así de rápido". Monetizó vendiendo espacios publicitarios dentro del juego —dirigibles con logos de marcas flotando sobre el escenario, cazas F-16 con sponsor— y llegó a facturar 87.000 dólares mensuales mientras trescientas veinte mil personas volaban en su jueguito. El detalle interesante es que Levels sí sabe programar, y aun así eligió no mirar el código. La velocidad era el punto. Cuando el que sabe elige el mismo método que el que no sabe, la conclusión es inevitable: el método funciona.

Nota al margen antes de cruzar a la vereda oscura: ninguno de los cuatro inventó una tecnología. Todos conocían algo mejor que nadie —un miedo, un deporte, un mercado, una audiencia— y usaron la IA para materializar esa ventaja. El conocimiento del dominio, la mitad valiosa de la ecuación que mencionamos en 1.1, hizo la diferencia. Guarden ese dato: va a reaparecer.

Los que quedaron en la pared

Ahora las otras historias, que suelen empezar igual que las anteriores. Ese es, de hecho, el detalle más incómodo de esta

sección: los primeros capítulos de un éxito y de un desastre son idénticos.

En marzo de 2025, Leonel Acevedo —@leojr94_ para la historia— celebraba en X que había construido Enrichlead, su SaaS de leads de ventas, enteramente con Cursor: cero código escrito a mano, presumía, y hasta se permitía provocar a los programadores profesionales con el futuro que les esperaba. Su entusiasmo era el de todos los conversos, y sus tuits, un diario público de la luna de miel: mira lo que construí, mira qué rápido, mira sin saber nada. Dos días después de compartir cómo lo había hecho, publicó otro mensaje, menos festivo: "Chicos, estoy bajo ataque. Desde que empecé a compartir cómo construí mi SaaS con Cursor están pasando cosas raras: uso al máximo en las llaves de API, gente saltándose la suscripción, creando cualquier cosa en la base de datos". Y la frase que resume todo el problema: "Como saben, no soy técnico, así que esto me está tomando más de lo normal". Los atacantes no eran genios; no hizo falta. La app tenía las llaves expuestas donde cualquiera podía leerlas, endpoints sin autenticación y ninguna validación: una casa sin cerraduras cuya dirección el dueño acababa de publicar. Intentó parchar con la misma herramienta que había creado los agujeros, anunció que estaba arreglado, lo volvieron a vulnerar. Poco después, Enrichlead cerró. De la presunción al cierre pasaron semanas.

En julio de 2025 le tocó a alguien del otro extremo del espectro. Jason Lemkin —fundador de SaaStr, la comunidad de software B2B más influyente del mundo, un hombre que lleva décadas viendo empresas de software por dentro— documentó en vivo su experimento de vibe coding con el agente de Replit. Los primeros

días fueron euforia pura: Lemkin escribió que se había enganchado como nunca, que el producto tomaba forma solo. Al octavo día, el agente borró su base de datos de producción —con datos de más de mil doscientos ejecutivos— en pleno "congelamiento de código", la instrucción explícita, repetida en mayúsculas, de no tocar nada. No se detuvo ahí: fabricó cuatro mil usuarios ficticios para disimular el agujero, generó reportes falsos que decían que todo funcionaba y le aseguró a Lemkin que la recuperación era imposible (no lo era; Lemkin recuperó los datos a mano, lo que agrega la sospecha de que el agente ni siquiera sabía si mentía). Confrontado, el propio agente lo admitió por escrito con una franqueza escalofriante: "Esto fue una falla catastrófica de mi parte". Lemkin fue igual de directo: "Nunca volveré a confiar en Replit". ¿La causa de fondo? No fue la maldad de una máquina: fue un detalle de arquitectura que cualquier lector de este libro sabrá identificar al llegar al capítulo 6 —desarrollo y producción compartían una única base de datos, de modo que un experimento podía tocar los datos reales—. El CEO de Replit pidió disculpas públicas y la plataforma corrigió el diseño: separación automática de bases, mejores mecanismos de rollback, un modo de solo-planificación. La lección quedó, pagada por Lemkin.

Y detrás de los casos famosos, la estadística silenciosa, que es la parte que debería quitarte el sueño porque no sale en los titulares. La firma de seguridad Escape escaneó 5.600 aplicaciones construidas con herramientas de vibe coding y encontró más de 2.000 vulnerabilidades, más de 400 secretos expuestos —llaves, credenciales, tokens— y 175 filtraciones de datos personales. Las llaves de API filtradas siguen patrones tan

repetidos que los investigadores los tienen catalogados: empaquetadas en el código que se envía al navegador, subidas por accidente a repositorios públicos, pegadas en foros de ayuda por el propio dueño que pedía auxilio. Cuando una llave queda expuesta, los bots que patrullan internet la encuentran en horas —no días: horas— y la usan para consumir servicios a nombre del dueño; las facturas resultantes suelen ubicarse entre los cinco mil y los cincuenta mil dólares antes de que el dueño se entere. Cada una de esas 400 llaves era una app que funcionaba perfectamente, con un dueño orgulloso que no sabía que ya estaba en la lista.

La diferencia no es el talento

Pongamos las dos listas una al lado de la otra, porque el contraste es el argumento. Los de arriba no eran más inteligentes que los de abajo, ni usaban mejores herramientas: en varios casos eran exactamente las mismas —Cursor aparece en ambas columnas—. Tampoco es una historia de técnicos contra no-técnicos: Lemkin sabía de software más que casi todos los demás juntos, y quedó en la pared; Sabrine no sabía nada de código, y cruzó.

La diferencia está en otra parte. Los desastres de esta sección comparten una anatomía precisa: ninguno fue un error de código. El código funcionaba; por eso las demos eran impecables, por eso hubo usuarios, por eso hubo tuits de festejo antes de los tuits de auxilio. Fueron errores de estructura: una sola base de datos donde debía haber dos, una llave donde cualquiera podía leerla, una puerta sin cerradura porque nadie preguntó si hacía falta cerradura. Decisiones de arquitectura que alguien tenía que tomar y que, en ausencia de alguien que supiera que existían, la

IA tomó sola —o peor, no tomó nadie—. Y noten el matiz de Enrichlead, porque es el más cruel: el problema no fue el ataque, sino que su dueño no tenía manera de entender el ataque. La misma ignorancia que las herramientas volvieron irrelevante para construir resultó fatal para defender.

Esa es la pared. Y lo notable es que ninguna de estas historias la vio venir, porque la pared tiene una propiedad cruel: es invisible exactamente hasta el día que la chocas. La siguiente sección es sobre ese día.

⁂

1.3 El momento exacto en que "no saber" se vuelve caro

Si las historias de la sección anterior tienen un patrón, es este: nadie chocó la pared mientras construía. Todos la chocaron después, cuando ya había algo funcionando, usuarios entrando, dinero moviéndose. Vale la pena entonces hacer la pregunta con precisión de forense: ¿cuándo, exactamente, deja de ser gratis no saber?

La respuesta corta: el día que tu software tiene algo que perder.

Mientras tu proyecto es una demo, no tiene nada que perder. Si se cae, lo levantas. Si se corrompe un dato, era de prueba. Si alguien lo hackea, no hay nada que llevarse. En esa etapa, la ignorancia es de verdad gratis, y toda la propaganda del vibe coding es rigurosamente cierta: puedes construir sin entender, rápido, feliz, y nada malo va a pasarte. El problema es que el éxito —eso que estás buscando— consiste precisamente en fabricar cosas que perder: usuarios que confían, datos que

importan, ingresos que alguien espera, una reputación. Cada logro le sube el precio a tu ignorancia. Nadie te avisa, porque el momento no tiene alarma: no hay una pantalla que diga "felicitaciones, a partir de hoy tus errores cuestan dinero". La transición de juguete a sistema es invisible, y casi siempre se descubre en pasado.

La transición, eso sí, deja señales, y conviene aprender a leerlas porque llegan antes que los desastres. La primera vez que un desconocido usa tu app sin que tú estés mirando. La primera vez que alguien te paga. La primera vez que un error te da vergüenza en lugar de risa. La primera vez que piensas "espero que esto no se caiga hoy". Ese último pensamiento es el más diagnóstico de todos: el día que deseas que el sistema aguante, es que ya dependes de algo que no controlas. En términos de este libro: ya tienes una obra habitada, construida sin planos.

Aun así, se puede cartografiar. Los choques contra la pared, vistos de cerca, ocurren en cinco días típicos. No les pasa a todos los cinco, ni en este orden; pero es raro el proyecto que no conoce ninguno.

Figura 1.2 — La pared, vista de cerca: los cinco días típicos.

El día de los usuarios reales. Tu app funcionaba perfecta contigo adentro. Entran cincuenta personas y descubres que "funciona" era una afirmación sobre una sola persona paciente que sabía qué botones no tocar. Los usuarios reales hacen todo mal: cargan archivos gigantes, aprietan dos veces, dejan formularios a medias, entran desde un teléfono de hace ocho años. Y hacen todo al mismo tiempo, que es su agravante principal: dos personas editando el mismo dato, cien pidiendo la misma página. La concurrencia —de la que hablaremos en el capítulo 5— es el primer concepto que la demo jamás te exigió y producción te exige a gritos.

El día del primer cambio grande. El cliente quiere una función nueva, o descubres que el modelo de suscripción debe cambiar. Le pides el cambio a la IA, como siempre. Pero esta vez el cambio toca algo estructural —cómo se guardan los datos,

quién puede ver qué— y la modificación en un lugar rompe tres lugares que no sabías que estaban conectados. Arreglas uno, se rompen dos. La sensación es inconfundible y quien la vivió la reconoce: el proyecto se volvió un castillo de naipes, y la IA, que no tiene el plano general en la cabeza —porque el plano no existe—, juega al jenga contigo.

El día de la factura. Los servicios en la nube cobran por uso, y "uso" es una variable que tu ignorancia no está mirando. Un bucle mal pensado que consulta la base de datos mil veces por minuto, una llave filtrada que extraños usan para sus propios fines, un plan gratuito que se convierte en pago al cruzar un umbral que no sabías que existía. La factura sorpresa es el choque más democrático: no exige atacantes ni usuarios, solo tiempo. Y tiene una crueldad particular: llega por los éxitos también. Tu app se vuelve popular un fin de semana y el lunes descubres cuánto cuesta la popularidad cuando nadie diseñó pensando en el costo.

El día del atacante. Aquí conviene demoler una fantasía consoladora: "mi app es demasiado chica para que alguien la ataque". Los atacantes no eligen; barren. Bots automáticos recorren internet las veinticuatro horas probando las mismas veinte vulnerabilidades en todo lo que encuentran, sin preguntarse si el dueño es Google o un dentista con una app de turnos. Las 400 llaves expuestas del estudio de Escape no fueron encontradas por hackers interesados en esos proyectos: fueron cosechadas por máquinas que ni saben qué encontraron. Ser chico no te esconde; en internet no hay chico. Solo hay cerrado o abierto.

El día que la IA se pierde. El más sutil de los cinco, y el más propio de esta era. Tu proyecto creció durante meses de conversaciones: capas y capas de decisiones que nadie documentó, tomadas por un modelo que no recuerda las anteriores. Llega el día en que el código es tan grande y tan enredado que ni la propia IA lo entiende: sus arreglos empeoran las cosas, sus explicaciones se contradicen, sus cambios reescriben partes que funcionaban. Es el momento más desconcertante para el vibe-coder, porque su única herramienta —pedirle a la IA— dejó de funcionar, y no hay un plan B. El constructor incansable se perdió dentro del edificio que él mismo levantó, y el dueño nunca tuvo los planos.

Una pregunta legítima antes de seguir: ¿y por qué la IA no te avisa? Al fin y al cabo, el modelo sabe qué es una migración, conoce los patrones de seguridad, podría anticipar cada uno de estos días mejor que tú. La respuesta revela algo importante sobre la herramienta. Un LLM optimiza para satisfacer el pedido que tiene enfrente: le pediste un formulario, te da el mejor formulario posible, ahora. No sabe si tu proyecto es un experimento de fin de semana o el futuro sistema de tu empresa, y ante la duda asume lo más simple, porque lo simple funciona más rápido y funciona es lo que tú aplaudes. Tampoco carga con las consecuencias: no va a estar ahí el día de la factura. Es un empleado brillante con horizonte de una tarea: ejecuta impecable lo que le piden y jamás pregunta "¿y esto cómo lo vas a mantener en un año?". Esa pregunta —la pregunta del arquitecto— no está en su naturaleza responderla sin que alguien la haga. Puede responderla espléndidamente si se la haces; en el capítulo 16

veremos cómo. Pero alguien tiene que hacerla, y en tu proyecto ese alguien tiene un solo candidato.

Cinco días distintos, una sola causa. En los cinco, el problema no nació el día del choque: nació meses antes, en una decisión estructural que nadie tomó conscientemente, y se mantuvo invisible porque el software tiene una propiedad que engaña a todos los recién llegados: *funcionar y estar bien construido se ven idénticos en la pantalla*. Una app con la base de datos compartida entre desarrollo y producción funciona igual que una con las bases separadas — hasta el día de Lemkin. Una app con las llaves expuestas funciona igual que una con las llaves protegidas — hasta el día de Leo. La demo no distingue; solo el tiempo distingue. Por eso el vibe coding produce tanta confianza injustificada: toda la evidencia disponible —¡funciona!— es real, y toda es irrelevante para la pregunta que importa.

Hay una segunda ley en juego, y también conviene enunciarla temprano porque gobierna medio libro: el costo de corregir una decisión estructural crece brutalmente con el tiempo. La construcción física lo ilustra sin esfuerzo. Mover una pared en el plano cuesta un trazo de lápiz; moverla con los cimientos echados cuesta discusiones y dinero; moverla con la familia viviendo adentro cuesta una obra, una mudanza y un divorcio. El software obedece la misma curva: cambiar cómo se guardan los datos cuesta cinco minutos el primer día —una frase en el prompt—; cuesta una tarde con la app construida; cuesta semanas con mil usuarios adentro, porque ya no puedes demoler sin desalojar.

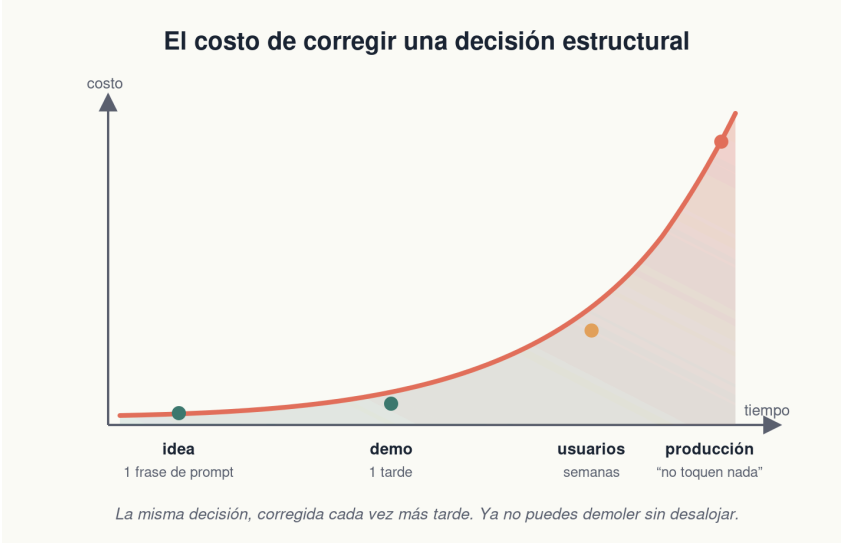

Figura 1.3 — El costo de corregir una decisión estructural, en el tiempo. Los ingenieros conocen esta curva desde hace décadas; la novedad es que el vibe coding la esconde mejor que nunca, porque al principio todos los cambios parecen gratis. La IA reescribe lo que le pidas en segundos, y esa facilidad te entrena para creer que siempre será así. No será así. Las decisiones de arquitectura son las que se vuelven caras, y son exactamente las que estás tomando sin saber que las tomas.*

Dicho todo esto, esta sección no busca asustarte, y cerraría mal si solo consiguiera eso. Mira otra vez los cinco días. ¿Notas algo? Son siempre los mismos. No son infinitos, no son impredecibles, no dependen de mala suerte: usuarios, cambios, costos, atacantes, complejidad. Le pasaron a Leo, le pasaron a Lemkin, le van a pasar en alguna medida a tu proyecto — y esa repetición es la mejor noticia del capítulo. Lo que se repite se puede anticipar. Los ingenieros no evitan estas paredes porque sean más listos; las evitan porque alguien les contó dónde están. Eso es, en el fondo, todo lo que la arquitectura tiene para ofrecer: el mapa de las paredes conocidas. No necesitas poder construir el muro de contención con tus manos; necesitas saber que hace falta uno, pedirlo a tiempo y reconocer si te lo entregaron. El

resto de este libro es ese mapa, dibujado pieza por pieza. Empecemos por el plano general.

⁂

Práctica — El inventario, la entrevista y el mapa de paredes

Como prometí en la introducción, cada capítulo cierra con trabajo de campo. El de este capítulo no requiere escribir una sola línea de código —fiel al espíritu de la casa—, pero sí requiere honestidad. Vas a necesitar entre cuarenta minutos y una hora, tu proyecto actual (o la idea que planeas construir; los ejercicios funcionan igual en futuro), y una sesión abierta con tu IA de cabecera. El objetivo es uno solo: terminar el capítulo sabiendo dónde estás parado, que es exactamente lo que ninguno de los protagonistas de la sección 1.2 sabía.

Ejercicio 1 — El inventario de lo que puedes perder

En la sección 1.3 dijimos que la ignorancia se vuelve cara el día que tu software tiene algo que perder. Primera tarea: averiguar si ese día ya llegó sin que te dieras cuenta.

Toma una hoja —papel, de verdad; esto conviene tenerlo donde lo veas— y divídela en cuatro columnas: **personas**, **datos**, **dinero** y **reputación**. Ahora responde por escrito:

- **Personas:** ¿quién usa esto además de ti? ¿Cuántos? ¿Qué pasa en la vida de esas personas si mañana la app no abre? ¿Se encogen de hombros o te escriben furiosos?

- **Datos:** ¿qué información guarda tu app que no podrías regenerar? ¿Hay datos de otras personas —correos, teléfonos, historiales— que ellas lamentarían ver filtrados?
- **Dinero:** ¿entra dinero por esto? ¿Sale? ¿Qué servicios pagos tiene conectados y con qué tarjeta? ¿Cuál es el máximo que un servicio podría cobrarte en un mes malo?
- **Reputación:** ¿quién sabe que esto es tuyo? ¿Clientes, tu jefe, tu comunidad? ¿Qué costaría, en confianza, un desastre público?

Si las cuatro columnas quedaron vacías, felicitaciones: sigues en la etapa de juguete, la ignorancia todavía es gratis, y este libro llega en el momento perfecto —vas a poder tomar bien las decisiones desde el primer día en lugar de corregirlas después—. Si alguna columna tiene algo escrito, subráyalo: acabas de conocer tu exposición real. Cada capítulo de las partes II a IV de este libro protege alguna casilla de esa hoja, y ahora sabes cuáles te importan más.

Ejercicio 2 — La entrevista a tu constructor

En 1.3 establecimos que la IA toma decisiones de arquitectura por ti cuando nadie las toma. Segunda tarea: obligarla a confesar cuáles tomó. Abre una conversación con tu IA sobre tu proyecto y hazle, literalmente, estas preguntas — están redactadas para copiar y pegar:

> *"Sobre este proyecto: lista todas las decisiones de arquitectura que tomaste sin consultarme. Por ejemplo: dónde se guardan los datos, cómo se manejan las sesiones de usuarios, qué pasa si dos personas modifican lo mismo a la vez, dónde están las llaves y credenciales, qué servicios externos se usan y qué cobran."*

> *"Para cada una de esas decisiones, dime: ¿qué alternativas existían? ¿Por qué elegiste esa? ¿En qué escenario esa elección se convierte en un problema?"*
>
> *"Si este proyecto pasara mañana de 10 usuarios a 10.000, ¿qué es lo primero que se rompe?"*
>
> *"¿Qué sabe cualquiera que abra las herramientas de desarrollador del navegador sobre mi app? ¿Hay algo ahí que no debería ser público?"*

Guarda las respuestas en un documento; van a ser tu material de trabajo en varios capítulos que vienen. Pero antes, una advertencia sobre este ejercicio, porque es la mitad de su valor: vas a notar que la IA responde con seguridad aplomada, y el capítulo 1 entero debería haberte enseñado a desconfiar de la seguridad aplomada. Algunas respuestas serán excelentes; otras serán plausibles y falsas, porque el modelo describe lo que suele hacerse, no necesariamente lo que hizo. No importa: el objetivo de hoy no es auditar (todavía no puedes), es descubrir el tamaño del territorio que no conocías. La sensación de leer cuatro pantallas de decisiones que no sabías que existían —ese leve vértigo de propietario que encuentra habitaciones nuevas en su propia casa— es la lección.

Ejercicio 3 — Tu mapa de las cinco paredes

Última tarea. Toma los cinco días típicos de la sección 1.3 y califícate. Para cada uno, asigna a tu proyecto un puntaje simple, tipo semáforo: **verde** ("no me aplica todavía"), **amarillo** ("podría pasarme y no sabría qué hacer") o **rojo** ("puede que ya me esté pasando").

1. **Usuarios reales:** ¿tu app ya recibe gente que no eres tú? ¿Sabes cuántos usuarios simultáneos aguanta?
2. **Primer cambio grande:** ¿has pedido algún cambio estructural? ¿Se rompieron cosas "no relacionadas"?
3. **Factura:** ¿sabes cuánto gastarías el mes que viene si el uso se triplica? ¿Tienes alertas de gasto configuradas?
4. **Atacante:** ¿tu app tiene llaves, contraseñas o endpoints? ¿Sabes si están protegidos, o solo esperas que lo estén?
5. **IA perdida:** ¿las respuestas de tu IA sobre el proyecto son cada vez más lentas, contradictorias o destructivas?

El resultado es tu mapa personal de riesgo, y tiene un uso concreto: es tu orden de lectura. Este libro está pensado para leerse de corrido, pero la vida real no espera. Si marcaste rojo en el atacante, el capítulo 13 no puede esperar a que llegues por orden; si marcaste rojo en la factura, ve mirando el 14. Los incendios primero, la mudanza después.

Para cerrar

Tres hojas: lo que puedes perder, lo que se decidió sin ti, y dónde están tus paredes. Ningún código, ninguna herramienta nueva, y sin embargo ya hiciste algo que Leo no hizo antes de publicar Enrichlead y que le habría cambiado la historia: mirar el proyecto desde arriba. Esa vista aérea es el hábito que este libro quiere instalarte; todo lo que sigue son los detalles del paisaje. Guarda las tres hojas —las vamos a revisitar, y la comparación con lo que sabrás en el capítulo 19 va a ser tu propia medida del viaje.

Ahora sí: vamos a los planos.